Netra™ CP2000 and CP2100 Series CompactPCI Boards Programming Guide

for the Solaris Operating Environment

Sun Microsystems, Inc.
www.sun.com

Part No. 816-2485-14
October 2004, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Please
Recycle

Contents

Figures

Tables

Code Samples

Preface

The Netra™ CP2040, Netra CP2060 and Netra CP2080, Netra CP2140 and Netra CP2160 CompactPCI boards are a crucial building block that network equipment providers (NEPs) and carriers can use when scaling and improving the availability of next-generation, carrier-grade systems.

The *Netra CP2000 and CP2100 Series cPCI Boards Programming Guide* is written for program developers and users who want to program these products in order to design original equipment manufacturer (OEM) systems, supply additional capability to an existing compatible system, or work in a laboratory environment for experimental purposes.

In the *Netra CP2000 and CP2100 Series cPCI Boards Programming Guide*, references are made to the Netra CP2000 board series and the Netra CP2100 board series. For the purpose of this book, the CP2000 board series refers to CP2040, CP2060 and CP2080 boards and the CP2100 board series currently includes the CP2140 and CP2160 boards.

Before You Read This Book

You are required to have a basic knowledge of computers and digital logic programming , in order to fully use the information in this document.

How This Book Is Organized

Chapter 1 provides details on the Netra CP2000 board and the CP2100 board series watchdog timer driver and its operation.

Chapter 2 describes the user flash driver for the Netra CP2000 board series and the CP2100 board series onboard flash PROMs and how to use it.

Chapter 3 describes the specific Advanced System Management (ASM) functions of the Netra CP2000 board series and the CP2100 board series.

Chapter 4 describes how to program the User LED on the Netra CP2100 board series.

Chapter 5 describes how to create applications that can identify and control hardware devices connected to Netra CP2100 series board-controlled systems.

Chapter 6 describes how to use Reconfiguration Coordination Manager scripts to automate certain dynamic reconfiguration processes for the Netra CP2100 board series.

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- Solaris Handbook for Sun Peripherals
- AnswerBook2™ online documentation for the Solaris™ operating environment
- Other software documentation that you received with your system

Typographic Conventions

Typeface*	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

* The settings on your browser might differ from these settings.

Shell Prompts

Shell	Prompt
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	`#`

Related Documentation

Application	Title	Part Number
Reference and Installation	*Netra CP2060/CP2080 Technical Reference and Installation Manual*	806-6658-xx
Reference and Installation	*Netra CP2040 Technical Reference and Installation Manual*	806-4994-xx
Reference and Installation	*Netra CP2140 Technical Reference and Installation Manual*	816-4908-xx
Reference and Installation	*Netra CP2160 CompactPCI Board Installation and Technical Reference Manual*	816-5772-xx

Accessing Sun Documentation

You can view, print, or purchase a broad selection of Sun documentation, including localized versions, at:

`http://www.sun.com/documentation`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number (816-2485-13) of your document in the subject line of your email.

CHAPTER 1

Watchdog Timer

The System Management Controller (SMC) on the Netra CP2000/CP2100 board, implements a two-level watchdog timer. The watchdog timer is used to recover the central processing unit (CPU) in case the CPU freezes.

This chapter provides detailed information on the SMC-based watchdog timer driver and its operation for the Netra CP2000/CP2100 boards. This chapter also describes the user-level application programming interface (API) and behavior of the Netra CP2000/CP2100 board watchdog timer. For functional details of the watchdog timer, see the technical reference and installation guide for your board product. See "Accessing Sun Documentation" on page xvi for information on accessing this documentation.

This chapter includes the following sections:

Watchdog Timers

There are two watchdog timers:

- 16-bit timer
- 8-bit pre-timeout timer

This section described one of the many different options the user can select regarding the actions for WD1 and WD2.

16-bit Timer (WD1)

Each tick represents 100 ms. This timer, set to a nonzero number, counts down first. When the timer reaches `zero`, a warning is sent to the host CPU through EBus and the WD2 pre-timeout counter is set to a nonzero value when interrupt option is enabled. Otherwise the SMC resets the host CPU immediately. The reset action takes place when the reset option is enabled

8-bit Pre-timeout Timer (WD2)

Each tick represents one second. This timer is started when the countdown timer reaches zero (if WD1 is set to zero, WD2 starts right away). When the value of this counter reaches zero, the host is reset. If the hard reset option is enabled, no warning is issued prior to reset

Watchdog Timer Driver

The watchdog driver is a loadable STREAMS pseudo driver layered atop the Netra CP2000/CP2100 series service processor hardware. This driver implements a standardized *watchdog timer* function that can be used by systems management software for a number of systems timeout tasks.

The systems management software that uses the watchdog driver has access to two independent timers, the *WD1* timer and the *WD2* timer. The WD2 is the main timer and is used to detect conditions where the Solaris operating environment hangs. Systems management software starts and periodically restarts the WD2 timer before it expires. If the WD2 timer expires, the watchdog function of the WD2 timer can force the SPARC™ processor to reset. The maximum range for WD2 is 255 seconds. Or the WD2 timer could be set to take no action.

The WD1 timer is typically set to a shorter interval than the WD2 timer. User applications can examine the expiration status of the WD1 timer to get advance warning if the main timer, WD2, is about to expire. The system management software has to start WD1 before it can start WD2. If WD1 expires, then WD2 starts only if enabled. The maximum range for WD1 is 6553.5 seconds.

The applications programming interface exported by the watchdog driver is input output control-based (IOCTL-based). The watchdog driver is an exclusive-use device. If the device has already been opened, subsequent opens fail with `EBUSY`.

Operations on the Watchdog Timers

Operations on the watchdog timers require a call to `ioctl(2)` using the parameters appropriate to the operation. The watchdog driver exports Input Output Controls (IOCTLs) to start, stop, and get the current status of the watchdog timers.

When the device is initially opened, both the watchdog timers, WD1 and WD2, are in `STOPPED` state. To start either timer, an application program must use the `WIOCSTART` command. Once started, the WD1 timer can be stopped by using the `WIOCSTOP` command. Once started, the WD2 timer cannot be stopped—it can only be restarted. Each watchdog timer takes the default action when it expires.

If the WD1 timer expires and the default action is enabled, WD1 interrupts the SPARC processor. This interrupt is handled and the status of the WD1 timer queried shows the `EXPIRED` condition. If the default action is disabled, then the WD1 timer is in `FREERUN` state and no interrupt is delivered to the SPARC processor on expiration.

If the WD2 timer expires and the default action is enabled, WD2 resets the SPARC processor. If the default action is disabled, the WD2 timer is put in `FREERUN` state and its expiration does not affect the SPARC processor.

In the Netra CP2000/CP2100 series board, the SMC-based watchdog timers are not independant. The WD2 timer is a continuation of the WD1 timer. There are some behavioral consequences to this implementation that result in the Netra CP2000/CP2100 series watchdog timer having different semantics. The most obvious difference is that starting one timer when the other timer is active causes the other timer to be restarted with its programmed timeout period.

Parameters Transfer Structure

The IOCTL-based watchdog timer application programming interface (API) uses a common data structure to communicate all requests and responses between the watchdog timer driver and user applications.

Along with other API definitions, this structure is defined in the include file `sys/wd_if.h`. The structure, called `watchdog_if_t,` is provided below for reference.

CODE EXAMPLE 1-1 Include File wd_if.h

```
#ifndef _SYS_WD_IF_H
#define _SYS_WD_IF_H

#pragma ident   "@(#)wd_if.h    1.3     01/12/17 SMI"

/*
 * wd_if.h
 * watchdog timer user interface header file.
 */

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * handy defines:
 */
#define WD1             1               /* wd level 1 */
#define WD2             2               /* wd level 2 */
#define WD3             3               /* wd level 3 */

/*
 * state of the counters:
 */
#define FREERUN         0x01            /* counter is running, no intr */
#define EXPIRED         0x02            /* counter has  expired */
#define RUNNING         0x04            /* counter is running, intr is on */
#define STOPPED         0x08            /* counter not started at all */
#define SERVICED        0x10            /* intr was serviced */

/*
 * IOCTL related stuff.
 */
/*
 * TIOC ioctls for watchdog control and monitor
 */
#if (!defined(_POSIX_C_SOURCE) && !defined(_XOPEN_SOURCE)) || \
        defined(__EXTENSIONS__)
#define wIOC    ('w' << 8)
#endif /* (!defined(_POSIX_C_SOURCE) && !defined(_XOPEN_SOURCE))... */

#define WIOCSTART       (wIOC | 0)      /* start counters */
#define WIOCSTOP        (wIOC | 1)      /* inhibit interrupts (stop) */
#define WIOCGSTAT       (wIOC | 2)      /* get status of counters */
```

CODE EXAMPLE 1-1 Include File `wd_if.h` *(Continued)*

```
typedef struct {
        int             thr_fd;         /* wd fd, used in the thread */
        uint8_t         thr_lock;       /* lock for the thread */
        uint8_t         level;          /* wd level */
        uint16_t        count;          /* value to be loaded into limit reg */
        uint16_t        next_count;     /* next lev timer count */
        uint8_t         restart;        /* timer to restart, 0 = stop */
        uint8_t         status[3];      /* status filled in ioctl() */
        uint8_t         inhibit;        /* inhibit timers, bit field */
} watchdog_if_t;

/*
 * Bit field defines for the user interface
 * inhibit.
 */
#define WD1_INHIBIT     0x1             /* inhibit timer 1 */
#define WD2_INHIBIT     0x2             /* inhibit timer 2 */
#define WD3_INHIBIT     0x4             /* inhibit timer 3 */

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_WD_IF_H */
```

The following fields are used by the IOCTL interface. The watchdog timer driver does not use the `thr_fd` and `thr_lock` fields.

`level`	Select timer to perform operations on: WD1 or WD2
`count`	The period for the timer specified by `level` to run before it expires. Legal values lie in the range from 1 to 65534. If the value of `count` is equal to 0 or -1, the timer is set to its default value. The default value for WD1 is 10 seconds and for WD2 it is 15 seconds.
`restart`	(Optional) Select a timer to start automatically when the timer specified by `level` expires. Legal values are WD1 or WD2. This timer can be the same or different from that specified by `level`.

next_count	(Optional) The period for the timer specified by restart to run before it expires. The next_count parameter is subject to the same range and default value rules as count, described above.
inhibit	This is a mechanism for controlling the action taken by a timer when it expires. The inhibit flag is a mask to control the default actions taken on the expiration of each timer. A bit corresponding to each timer determines whether the timer's default action is enabled or disabled. If the corresponding bit in inhibit is zero, then the default action occurs on expiration of that timer; if the bit is set to one, then the default action is disabled. The symbolic names for the control masks, defined in sys/wd_if.h, are WD1_INHIBIT for timer WD1, and WD2_INHIBIT for timer WD2.
status	After a call to ioctl(2) with the WIOCGSTAT command, the status vector reflects the state of each watchdog timer (WD1 and WD2) available on the system. The status vector element status[0] corresponds to the state of WD1 and status[1] corresponds to the state of WD2.

The states that each watchdog timer can assume are listed below. These states are exclusive of each other.

STOPPED	The counter is not running.
RUNNING	The counter is running, and its associated action (interrupt or system reset) is enabled.
FREERUN	The counter is running, but no associated action is enabled.

In addition to these states, the following modes can become attached to a timer, based on its state:

EXPIRED	This mode is applicable only to the WD1 timer. This mode indicates that the WD1 timer interrupt has expired.
SERVICED	This mode is also applicable only to the WD1 timer. This mode indicates that an expiration interrupt has occurred and been serviced by the driver. This mode is cleared once it is reported to the user through WIOCGSTAT. Thus, if two consecutive IOCTL calls using WIOCGSTAT are made by a user program, the driver might return SERVICED for the first IOCTL call, but not for the second.

Input/Output Controls

The watchdog timer driver supports the following input/output control (IOCTL) requests:

`WIOCGSTAT`	Get the state of all the watchdog timers. If the `level` field of the `watchdog_if_t` structure is a valid value (either `WD1` or `WD2`), the `WIOCGSTAT` IOCTL returns the status of both timers in the `status` vector or the structure. Getting the status of the timers clears the `EXPIRED` bit if set for the timer specified by the `level` field of the `watchdog_if_t` structure, so that each timer expiration event is reported.
`WIOCSTART`	A few behavioural consequences are associated with the `WIOCSTART` command that arise from the fact that WD1 and WD2 timers are not independent in the Netra CP2000/CP2100 series board implementation. When a `WIOCSTART` command is issued, the other timer, if already running, will be restarted from its current initial value. In addition, since the WD2 timer is in a sense an extension of the WD1 timer, it is not permissible to set the count value for WD1 to a value greater than that of an active WD2 timer. Similarly, it is not permissible to set the count value for WD2 to a value greater than that of an active WD1 timer. The following rules are applied when setting a timer if the other timer is already active: When WD1 is active, lowering WD2 to a value less than that of WD1 will cause WD1 to be lowered to be equal to WD2. When WD2 is active, raising WD1 to a value greater than that of WD2 will raise the value of WD2 to be the same as WD1.
`WIOCSTOP`	The `WIOCSTOP` command disables timer expiration actions. The `inhibit` mask parameter of the `watchdog_if_t` structure determines which timer is being controlled by `WIOCSTOP`. The `level` parameter of the `watchdog_if_t` structure passed with this command must be a valid watchdog level: either WD1 or WD2. If the watchdog level is not valid, you will receive an error message indicating that the device is not valid. It is possible to stop the WD1 timer if it is running. However, once started, the WD2 timer cannot be stopped and resets the system unless it is prevented from expiration by being periodically restarted.

Errors

EBUSY	An application program attempted to perform an open(2) on /dev/wd but another application already owned the device.
EFAULT	An invalid pointer to a watchdog_if_t structure was passed as a parameter to ioctl(2).
EINVAL	The IOCTL command passed to the driver was not recognized. OR The level parameter of the watchdog_if_t structure is set to an invalid value. Legal values are WD1 or WD2. OR The restart parameter of the watchdog_if_t structure is set to an invalid value. Legal values are WD1, WD2, or zero.
ENXIO	The watchdog driver has not been plumbed to communicate with the SMC device driver.

Example

This code example retrieves the status of the watchdog timers, then starts both timers:

CODE EXAMPLE 1-2 Status of Watchdog Timers and Starting Timers

```
#include                  sys/fcntl.h
#include                  sys/wd_if.h
       .
       .
       .
int                       fd;
watchdog_if_t             wdog1;
watchdog_if_t             wdog2;
int                       rperiod = 5;

          /*
           * open the watchdog driver
           */

          if ((fd = open("/dev/wd", O_RDWR)) < 0) {
              perror("/dev/wd open failed");
              exit(0);
          }

          /*
```

CODE EXAMPLE 1-2 Status of Watchdog Timers and Starting Timers *(Continued)*

```
             * get the status of the timers

             */
            wdog1.level = WD1;
                 /* must be a valid value */
            if (ioctl(fd, WIOCGSTAT, &wdog1) < 0) {
                perror("WIOCGSTAT ioctl failed");
                exit(0);
            }

            printf("Status WD1: 0x%x WD2: 0x%x\n",
                wdog1.status[0], wdog1.status[1]);

            /*
             * Start WD1 to give advance warning if we don't
             * respond in 10 seconds. Also, when WD1 expires,
             * restart it automatically.
             */

        #define RES(sec) (10 * (sec))
               /* convert to 0.1 sec resolution */
            wdog1.level = WD1;
            wdog1.count = RES(10);
                /* 10 sec, resolution of 0.1 sec */
            wdog1.restart = WD1;
            wdog1.next_count = RES(10);
                /* 10 sec, resolution of 0.1 sec */

            /*
             * start the timers ticking...
             */
            if (ioctl(fd, WIOCSTART, &wdog1) < 0) {
                perror("WIOCSTART ioctl failed");
                exit(0);
            }

            /*
             * Start WD2 to reset the SPARC processor if we don't
             * kick it again within 20 seconds.
             */
            wdog2.level = WD2;
            wdog2.count = RES(20);
               /* 20 sec, resolution of 0.1 sec */
            wdog2.restart = 0;

            if (ioctl(fd, WIOCSTART, &wdog2) < 0) {
                perror("WIOCSTART ioctl failed");
```

CODE EXAMPLE 1-2 Status of Watchdog Timers and Starting Timers *(Continued)*

```
            exit(0);
        }

        /*
         * loop, restarting the timers to prevent RESET
         */

        for (;;) {
            watchdog_if_t                wstat;

            /*
             * first sleep for the desired period
             * before restarting the timer(s)
             */
            sleep(rperiod);

            /*
             * setup to get the status of the timers
             */
            wstat.level = WD1;/* must be a valid value */
            if (ioctl(fd, WIOCGSTAT, &wstat) < 0) {
                perror("WIOCGSTAT ioctl failed");
                exit(0);
            }
            /*
             * If the WD1 timer has expired, take
             * appropriate action.
             */
            if (wstat.status[0] & EXPIRED) {
                /* timer expired. shorten sleep? */
                puts("WD1: <EXPIRED>");
            }

            /*
             * restart the timers
             */
            if (ioctl(fd, WIOCSTART, &wdog2) < 0) {
                perror("WIOCSTART ioctl failed");
                exit(0);
            }
        }
```

Configuration

The watchdog device driver runs only on the following implementations:

- SUNW, UltraSPARCengine_CP-40 (for Netra CP2040 and CP2140)
- SUNW, UltraSPARCengine_CP-60 (for Netra CP2060 CP2160)
- SUNW, UltraSPARCengine_CP-80 (for Netra CP2080)

By rule, the watchdog driver and its configuration file must reside in the platform-specific driver directory, /platform/*implementation*/kernel/drv. The value of *implementation* for a given Netra CP2000/CP2100 board system can be obtained by running the uname(1) command on that machine with the -i option:

```
# uname -i
SUNW, UltraSPARCengine_CP-60
```

This directory contains the wdog.conf driver configuration file. This file controls the boot-time configuration of the watchdog timer driver. The driver is configured through a directive to send a notice to syslog when the WD1 timer interrupt is serviced. The Netra CP2000/CP2100 board implementation requires that the appropriate control directive be placed in wdog.conf.

The format for this directive is as follows:

```
        #

        # control to enable syslog notification when a WD1

        # interrupt is handled.

        # handler-message="on" enables syslog notice.

        # handler-message="off" disables syslog notice.

        #

        handler-message="on"
```

OpenBoot PROM Interface

The OpenBoot™ PROM provides two environmental parameters, settable at the ok prompt, that control the behavior of the SMC watchdog timer.

These parameters are watchdog-enable? and watchdog-timeout?. The watchdog-enable? parameter is a logical switch with two possible values: true or false.

If `watchdog-enable?` is set to `false`, the watchdog timer is disabled at boot time,. Once the kernel is booted, applications have the option to start the watchdog timer.

If `watchdog-enable?` is set to `true`, the watchdog timer is enabled at boot time with its default actions: The WD1 timer is controlled by the value in *watchdog-timeout* variable. When WD1 expires it sends an asynchronous message to the local CPU. It also starts the WD2 timer. The default value for the WD2 timer is 1 second. If the WD2 timer expires, it resets the CPU board.

If the watchdog timer is enabled at boot time, it is your responsibility to ensure that an application program is run to periodically restart the WD1 timer. If you fail to do so, the timer expires. The system could be reset when the watchdog timer expires.

Data Structure

Refer to CODE EXAMPLE 1-1 for details on the data structure that is used with watchdog timer programs.

Watchdog Operation

The watchdog operation (the *local watchdog*) is the watchdog that works between the host CPU and System Management Controller (SMC).

Commands at OpenBoot PROM Prompt

TABLE 1-1 lists the commands at OpenBoot prompt.

TABLE 1-1 OpenBoot PROM Prompt Commands

Command	**Description**
`smc-get-wdt`	Gets the current timers values, and other watchdog state bits.
`smc-set-wdt`	Sets the timers values and other flags. This command is also used to stop watchdog operations.
`smc-reset-wdt`	Starts timer countdown and is often referred to as the "heartbeat".

Corner Cases

When watchdog reset occurs, the power module is toggled. Thus, the state of the CPU, except those stored in nonvolatile memory, will be lost. Once watchdog reset occurs after the host CPU is restarted, the host CPU must restart the watchdog timer.

The host CPU must perform a corner case. After the SMC resets the host CPU, the output buffer full (OBF) bit and OEM1 bit in the EBus status register remain set. Since this is a read-only bit, the SMC cannot reset the bit. The host must ignore the status bits and clear the OBF bit by reading one byte of data from EBus. This action must be performed after watchdog reset. Otherwise, the host CPU can inadvertently restart watchdog. For example, if the timer's values are set to very low numbers, the board can never boot to the Solaris operating system.

The SMC manages the race condition by putting interlock. The SMC does not start pre-timeout timer unless the warning is dispatched to the host CPU. The code is set up on the host side after watchdog warning is issued. Use a Keyboard Controller Style (KCS) command to clear the watchdog interrupt. Using this command is the only way to avoid the selected pre-timeout action such as hard reset. This command rewinds the watchdog timer. The host code internally manages the warning, along with the command being sent to the SMC.

If `diag-switch?` is set to true, the timing for watchdog can be affected.

Setting the Watchdog Timer at OpenBoot PROM

▼ To Set the Watchdog Timer Without Running the Pre-Timeout Timer

The examples below are at the OpenBoot PROM level. AFter Level 1 expires the local CPU is put into reset.

1. **Set the timer to 10 minutes = 600 sec = 600,000/10 msec = 0x1770.**
2. **Set the reload values inside the SMC:**

```
ok 17 70 ff 0 31 4 smc-set-wdt
```

3. **Start the watchdog timer:**

```
ok smc-reset-wdt
```

▼ To Set the Watchdog Timer With Pre-Timeout Time

This procedure sets the reload values of countdown timer and pre-timeout timer. Following the Level 1 expiry, there are 80 seconds before the reset action.

1. **Set the timer to 80 seconds = 0x50.**

   Set the countdown value to 10 minutes, as in the previous procedure, and set the pre-timout timer to 80 seconds.

```
ok 17 70 ff 50 31 4 smc-set-wdt
```

2. **Start the watchdog timer:**

```
ok smc-reset-wdt
```

▼ To Stop the Watchdog Timer

```
ok ff ff ff 0 31 4 smc-set-wdt
```

CHAPTER 2

User Flash

This chapter describes the user flash driver for the onboard flash PROMs and how to use it. The Netra CP2000/CP2100 series boards are equipped with user flash memory. This chapter includes the following sections:

User Flash Usage and Implementation

The customer can use the flash memory for various purposes such as storage for RTOS, user data storage, OpenBoot PROM information or to store *dropins.* Dropins simplify customizing a system for the user.

When OpenBoot PROM in system flash is corrupted, and if a backup copy of OpenBoot PROM is stored in user flash, you can switch the SMC switch to boot the OpenBoot PROM from the user flash and then use flash update to get a good OpenBoot PROM image back into the system flash.

A user flash switch SW2501 determines whether the user flash is detected during OpenBoot PROM boot and whether or not it is write-enabled. See "Switch Settings" on page 19 for more information.

The user flash includes flash PROM chips that can be programmed by users (see TABLE 2-1).

TABLE 2-1 User Flash Implementation

CompactPCI Board	Implementation	Total Memory Size
Netra CP2040	Two user flash modules	2 X 4MB
Netra CP2060	One user flash module	1 x 4 MB
Netra CP2080	One user flash module	1 x 4 MB
Netra CP2140	Two user flash modules	2 x 4MB
Netra CP2160	One user flash module	1 x 8MB

User Flash Address Range

The address range for 1 x 4MB user flash : 0x1ff.f040.0000 to 0x1ff.f07f.ffff.

The address range for 1 X 8MB flash: 0x1ff.f040.0000 to 0x1ff.f0bf.ffff

System Compatibility

TABLE 2-2 lists the compatible releases that support the user flash driver.

TABLE 2-2 Compatible Releases That Support the User Flash Driver

CompactPCI Board	Component	Compatible Release
Netra CP2060	Hardware	All board versions
	OpenBoot PROM	OpenBoot PROM Release 4.0.45 SMC Firmware Release 3.10.5 FPGA Version 1.2 PLD Version 4.2 All the above versions or other versions that support this feature
	Operating environment	Solaris 8 1/01 operating environment or other versions that support this feature
Netra CP2080	Hardware	All board versions
	OpenBoot PROM	OpenBoot PROM Release 4.0.45 SMC Firmware Release 3.10.5 FPGA Version 1.2 PLD Version 4.2 All the above versions or other versions that support this feature
	Operating environment	Solaris 8 1/01 operating environment or other versions that support this feature

TABLE 2-2 Compatible Releases That Support the User Flash Driver

CompactPCI Board	Component	Compatible Release
Netra CP2040	Hardware	All board versions
	OBP	OpenBoot PROM Release 4.0.27 SMC Firmware Release 3.4.4 FPGA Version 1.0 PLD Version 1.2 All the above versions or other versions that support this feature
	Operating environment	Solaris 8 1/01 operating environment or other versions that support this feature
Netra CP2140	Hardware	All board versions
	OBP	OpenBoot PROM Release 4.0.3 SMC Firmware Release 3.4.10 FPGA Version 1.0 PLD Version 1.3 All the above versions or other versions that support this feature
	Operating environment	Solaris 8 2/02 operating environment or other versions that support this feature
Netra CP2160	Hardware	All board versions
	OBP	OpenBoot PROM Release 4.0.11 SMC Firmware Release 4.0.6 FPGA Version 1.2 PLD Version 4.2 All the above versions or other versions that support this feature
		Solaris 8 2/02 operating environment or other versions that support this feature

User Flash Driver

The *uflash* is the device driver for flash PROM devices on the Netra CP2000/CP2100 series boards. Access to the driver is carried out through `open`, `read`, `write`, `pread`, `pwrite` and `ioctl` system interfaces.

Depending on the platform, one or more of these devices are supported. There is one logical device file for each physical device that can be accessed from applications. Users can use these devices for storing applications and data.

When multiple user flash devices are supported by the system, an instance of the driver is loaded per device. The driver blocks any reads to the device, while a write is in progress. Multiple, concurrent reads can go through to the same device at the same time. Writes to a device occur independently of the others. All read and write operations are supported at this time.

Access to the device normally happens a byte at a time. Devices support buffers to speed up writes. The driver automatically switches to the buffer mode, when the feature is available and the request is of sufficient size.

Devices also support erase and lock features. Applications can use them through the IOCTL interface. Devices are divided into logical blocks. Applications that issue these operations also supply a block number or a range of blocks that are a target of these operations. Locks are preserved across reboots. Locking a block prevents an erase or write operation on that block.

Switch Settings

The user flash modules on the Netra boards are write enabled by default. The user flash is detected during OpenBoot PROM boot by default.

See the following documents for more details on switch settings:

- *Netra CP2040 Technical Reference and Installation Manual*, (806-4994-*xx*)
- *Netra CP2140 Technical Reference and Installation Manual* (816-4908-*xx*)
- *Netra CP2060 and CP2080 Technical Reference and Installation Manual (*806-6658-*xx*)
- *Netra CP2160 CompactPCI Board Installation and Technical Reference Manual* (816-5772-*xx*)

OpenBoot PROM Device Tree and Properties

This section provides information on the user flash OpenBoot PROM device node and its properties.

User flash OpenBoot PROM device node:

```
/pci@1f,0/pci@1,1/ebus@1/flashprom@10,800000
/pci@1f,0/pci@1,1/ebus@1/flashprom@10,400000
```

See TABLE 2-3 for the user flash node properties.

TABLE 2-3 User Flash Node Properties

Property	Description/Value
compatible	`user flash`
user	
reg	`00000010 00400000 00400000`
block-size	`00010000`
dcode-offset	`00000002`
blocks-per-bank	`00000020`
model	`SUNW,yyy-yyyy`

User Flash Packages

The user flash packages are as follows:

- `SUNWufr.u`—32 bit driver
- `SUNWufrx.u`—64 bit driver
- `SUNWufu`—include files

These packages are available with the rest of the software on the *CP2000 Supplemental CD 4.0 for Solaris 8*.

User Flash Device Files

The user flash device files are as follows:

- `/dev/uflash0`—Netra CP2060, Netra CP2080, and Netra CP2160
- `/dev/uflash0, /dev/uflash1`—Netra CP2040
- `/dev/uflash0, /dev/uflash1`—Netra CP2140

Interface (Header) File

The user flash header file is located in the following path:

`/usr/include/sys/uflash_if.h`

Application Programming Interface

Access to the user flash device from the Solaris operating environment is through a C program. No command-line tool is available. User programs open these device files and then issue `read`, `write`, or `ioctl` commands to use the user flash device.

The systems calls are listed below in TABLE 2-4.

TABLE 2-4 System Calls

Call	Description
`read(), pread()`	reads devices
`pwrite()`	writes devices
`ioctl()`	erases device, queries device parameters

The `ioctl` commands are listed below.

```
#define UIOCIBLK (uflashIOC|0) /* identify */

#define UIOCQBLK (uflashIOC|1) /* query a block */

#define UIOCLBLK (uflashIOC|2) /* lock a block */

#define UIOCMLCK (uflashIOC|3) /* master lock */

#define UIOCCLCK (uflashIOC|4) /* clear all locks */
```

```
#define UIOCEBLK (uflashIOC|5) /* erase a block */

#define UIOCEALL (uflashIOC|6) /* erase all unlocked blocks */

#define UIOCEFUL (uflashIOC|7) /* erase full chip */
```

Structures to Use in IOCTL Arguments

PROM Information Structure

The PROM information structure holds device information returned by the driver in response to an identify command.

CODE EXAMPLE 2-1 PROM Information Structure

```
/*
 * PROM info structure.
 */
typedef struct {
        uint16_t        mfr_id;            /* manufacturer id */
        uint16_t        dev_id;            /* device id */
        /* allow future expansion */
        int8_t          blk_status[256];   /* blks status filled
by driver */
       int32_t         blk_num;           /* total # of blocks */
      int32_t        blk_size;         /* # of bytes per block */
} uflash_info_t;
```

User Flash User Interface Structure

The user flash user interface structure holds user parameters to commands such as `erase`.

CODE EXAMPLE 2-2 User Flash Interface Structure

```
/*
 * uflash user interface structure.
 */
typedef struct {
        int             blk_num;
```

CODE EXAMPLE 2-2 User Flash Interface Structure

```
         int             num_of_blks;
        uflash_info_t   info;             /* to be filled by the
driver */
} uflash_if_t;
```

Errors

EINVAL	Application passed one or more incorrect arguments to the system call.
EACCESS	Write or Erase operation was attempted on a locked block.
ECANCELLED	A hardware malfunction has been detected. Normally, retrying the command should fix this problem. If the problem persists, power cycling the system may be necessary.
ENXIO	This error indicates problems with the driver state. Power cycle of the system or reinstallation of driver may be necessary.
EFAULT	An error was encountered when copying arguments between the application and driver (kernel) space.
ENOMEM	System was low on memory when the driver attempted to acquire it.

Example Programs

Example programs are provided in this section for the following actions on user flash device:

- Read
- Write
- Erase
- Block Erase

Read Example Program

CODE EXAMPLE 2-3 contains the Read Action on the user flash device.

CODE EXAMPLE 2-3 Read Action on User Flash Device

```
/*
* uflash_read.c
 * An example that shows how to read user flash
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <uflash_if.h>
char *uflash0 = "/dev/uflash0";
char *uflash1 = "/dev/uflash1";
int ufd0, ufd1;
uflash_if_t ufif0, ufif1;
char *buf0;
char *buf1;
char *module;
static int
uflash_init() {
  char *buf0 = malloc(ufd0.info.blk_size);
  char *buf1 = malloc(ufd1.info.blk_size);
if (!buf0 || !buf1) {
    printf("%s: cannot allocate memory\n", module);
    return(-1);
  }
/* open device(s) */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
  }
if ((ufd1 = open(uflash1, O_RDWR)) == -1 ) {
    perror("uflash1: ");
  }
if ((ufd0 == -1) && (ufd1 == -1)) {
    printf("\n%s: cannot open uflash devices\n");
    exit(1);
  }
if (ufd0 == -1)
```

CODE EXAMPLE 2-3 Read Action on User Flash Device *(Continued)*

```
    ufd0 = 0;
  if (ufd1 == -1)
    ufd1 = 0;
/* get uflash sizes */
  if (ufd0 && ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
if (ufd1 && ioctl(ufd1, UIOCIBLK, &ufif1) == -1 ) {
    perror("ioctl(ufd1, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufd0.info.mfr_id);
    printf("device id = 0x%p\n", ufd0.info.dev_id);
    printf("number of blocks = 0x%p", ufd0.info.blk_num);
    printf("block size = 0x%p"  ufd0.info.blk_size);
  }
if (ufd1) {
    printf("%s: \n", uflash1);
    printf("manfacturer id = 0x%p\n", ufd1.info.mfr_id);
    printf("device id = 0x%p\n", ufd1.info.dev_id);
    printf("number of blocks = 0x%p", ufd1.info.blk_num);
    printf("block size = 0x%p"  ufd1.info.blk_size);
  }
}
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
  if (ufd1)
    close(ufd1);
cleanup:
  if (buf0)
    free(buf0);
  if (buf1)
    free(buf1);
}
static int
uflash_read() {
  /* read block 0 of user flash 0 */
```

CODE EXAMPLE 2-3 Read Action on User Flash Device *(Continued)*

```
  if (pread(ufd0, buf0, ufd0.info.blk_size, 0) !=
ufd0.info.blk_size)
        perror("uflash0:read");
/* read block 1 of user flash 1 */
  if (pread(ufd1, buf1, ufd1.info.blk_size, ufd0.info.blk_size)
       != ufd1.info.blk_size)
        perror("uflash1:read");
return(0);
}
main() {
  int ret;
module = argv[0];
ret = uflash_init();
if (!ret)
    uflash_read();
uflash_uninit();
}
```

Write Example Program

CODE EXAMPLE 2-4 contains the Write Action on the user flash device.

CODE EXAMPLE 2-4 Write Action on User Flash Device

```
/*
 * uflash_write.c
 * An example that shows how to write user flash
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <uflash_if.h>
char *uflash0 = "/dev/uflash0";
char *uflash1 = "/dev/uflash1";
int ufd0, ufd1;
uflash_if_t ufif0, ufif1;
char *buf0;
char *buf1;
char *module;
```

CODE EXAMPLE 2-4 Write Action on User Flash Device *(Continued)*

```
static int
uflash_init() {
  char *buf0 = malloc(ufd0.info.blk_size);
  char *buf1 = malloc(ufd1.info.blk_size);
if (!buf0 || !buf1) {
    printf("%s: cannot allocate memory\n", module);
    return(-1);
  }
/* open device(s) */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
  }
if ((ufd1 = open(uflash1, O_RDWR)) == -1 ) {
    perror("uflash1: ");
  }
if ((ufd0 == -1) && (ufd1 == -1)) {
    printf("\n%s: cannot open uflash devices\n");
    exit(1);
  }
if (ufd0 == -1)
    ufd0 = 0;
  if (ufd1 == -1)
    ufd1 = 0;
/* get uflash sizes */
  if (ufd0 && ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
if (ufd1 && ioctl(ufd1, UIOCIBLK, &ufif1) == -1 ) {
    perror("ioctl(ufd1, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufd0.info.mfr_id);
    printf("device id = 0x%p\n", ufd0.info.dev_id);
    printf("number of blocks = 0x%p", ufd0.info.blk_num);
    printf("block size = 0x%p"  ufd0.info.blk_size);
  }
if (ufd1) {
    printf("%s: \n", uflash1);
    printf("manfacturer id = 0x%p\n", ufd1.info.mfr_id);
    printf("device id = 0x%p\n", ufd1.info.dev_id);
```

CODE EXAMPLE 2-4 Write Action on User Flash Device *(Continued)*

```
    printf("number of blocks = 0x%p", ufd1.info.blk_num);
    printf("block size = 0x%p"  ufd1.info.blk_size);
  }
}
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
  if (ufd1)
    close(ufd1);
cleanup:
  if (buf0)
    free(buf0);
  if (buf1)
    free(buf1);
}
static int
uflash_write() {
  int i;
/* write some pattern to the buffers */
  for (i = 0; i < ufd0.info.blk_size; i += sizeof(int))
        *((int *) (buf0 + i)) = 0xDEADBEEF;
for (i = 0; i < ufd1.info.blk_size; i += sizeof(int))
        *((int *) (buf1 + i)) = 0xDEADBEEF;
/* write block 0 of user flash 0 */
  if (pwrite(ufd0, buf0, ufd0.info.blk_size, 0) !=
ufd0.info.blk_size)
        perror("uflash0:write");
/* write block 1 of user flash 1 */
  if (pwrite(ufd1, buf1, ufd1.info.blk_size, ufd0.info.blk_size)
       != ufd1.info.blk_size)
        perror("uflash1:write");
return(0);
}
main() {
  int ret;
module = argv[0];
ret = uflash_init();
if (!ret)
    uflash_write();
uflash_uninit();
}
```

Erase Example Program

CODE EXAMPLE 2-5 contains the Erase Action on the User Flash Device.

CODE EXAMPLE 2-5 Erase Action on User Flash Device

```
/*
 * uflash_erase.c
 * An example that shows how to erase user flash
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <uflash_if.h>
char *uflash0 = "/dev/uflash0";
char *uflash1 = "/dev/uflash1";
int ufd0, ufd1;
uflash_if_t ufif0, ufif1;
char *module;
static int
uflash_init() {
 /* open device(s) */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
  }
if ((ufd1 = open(uflash1, O_RDWR)) == -1 ) {
    perror("uflash1: ");
  }
if ((ufd0 == -1) && (ufd1 == -1)) {
    printf("\n%s: cannot open uflash devices\n");
    exit(1);
  }
if (ufd0 == -1)
    ufd0 = 0;
  if (ufd1 == -1)
    ufd1 = 0;
/* get uflash sizes */
  if (ufd0 && ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
```

CODE EXAMPLE 2-5 Erase Action on User Flash Device *(Continued)*

```
if (ufd1 && ioctl(ufd1, UIOCIBLK, &ufif1) == -1 ) {
    perror("ioctl(ufd1, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufd0.info.mfr_id);
    printf("device id = 0x%p\n", ufd0.info.dev_id);
    printf("number of blocks = 0x%p", ufd0.info.blk_num);
    printf("block size = 0x%p"  ufd0.info.blk_size);
  }
if (ufd1) {
    printf("%s: \n", uflash1);
    printf("manfacturer id = 0x%p\n", ufd1.info.mfr_id);
    printf("device id = 0x%p\n", ufd1.info.dev_id);
    printf("number of blocks = 0x%p", ufd1.info.blk_num);
    printf("block size = 0x%p"  ufd1.info.blk_size);
  }
}
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
  if (ufd1)
    close(ufd1);
}
static int
uflash_erase() {
  if (ufd0 && ioctl(ufd0, UIOCEFUL, &ufd0) == -1 ) {
    perror("ioctl(ufd0, UIOCEFUL): ");
    return(-1);
  }
  printf("\nerase successful on %s\n", uflash0);
if (ufd1 && ioctl(ufd1, UIOCEFUL, &ufd1) == -1 ) {
    perror("ioctl(ufd1, UIOCEFUL): ");
    return(-1);
  }
  dprintf("\nerase successful on %s\n", uflash1);
return(0);
}
main() {
  int ret;
module = argv[0];
```

CODE EXAMPLE 2-5 Erase Action on User Flash Device *(Continued)*

```
ret = uflash_init();
if (!ret)
    uflash_erase();
uflash_uninit();
}
```

Block Erase Example Program

CODE EXAMPLE 2-6 contains the Block Erase Action on the user flash device.

CODE EXAMPLE 2-6 Block Erase Action on User Flash Device

```
/*
 * uflash_blockerase.c
 * An example that shows how to erase block(s) of user flash
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <uflash_if.h>
char *uflash0 = "/dev/uflash0";
char *uflash1 = "/dev/uflash1";
int ufd0, ufd1;
uflash_if_t ufif0, ufif1;
char *module;
static int
uflash_init() {
 /* open device(s) */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
  }
if ((ufd1 = open(uflash1, O_RDWR)) == -1 ) {
    perror("uflash1: ");
  }
if ((ufd0 == -1) && (ufd1 == -1)) {
    printf("\n%s: cannot open uflash devices\n");
    exit(1);
  }
if (ufd0 == -1)
```

CODE EXAMPLE 2-6 Block Erase Action on User Flash Device *(Continued)*

```
    ufd0 = 0;
  if (ufd1 == -1)
    ufd1 = 0;
/* get uflash sizes */
  if (ufd0 && ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
if (ufd1 && ioctl(ufd1, UIOCIBLK, &ufif1) == -1 ) {
    perror("ioctl(ufd1, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufd0.info.mfr_id);
    printf("device id = 0x%p\n", ufd0.info.dev_id);
    printf("number of blocks = 0x%p", ufd0.info.blk_num);
    printf("block size = 0x%p"  ufd0.info.blk_size);
  }
if (ufd1) {
    printf("%s: \n", uflash1);
    printf("manfacturer id = 0x%p\n", ufd1.info.mfr_id);
    printf("device id = 0x%p\n", ufd1.info.dev_id);
    printf("number of blocks = 0x%p", ufd1.info.blk_num);
    printf("block size = 0x%p"  ufd1.info.blk_size);
  }
}
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
  if (ufd1)
    close(ufd1);
}
static int
uflash_blockerase() {
  /* erase 2 blocks starting from block 1 of user flash 0 */
  uf0.blk_num = 1;
  uf0.num_of_blks = 2;
  if (ufd0 && ioctl(ufd0, UIOCEBLK, &ufd0) == -1 ) {
    perror("ioctl(ufd0, UIOCEBLK): ");
    return(-1);
  }
```

CODE EXAMPLE 2-6 Block Erase Action on User Flash Device *(Continued)*

```
  printf("\nblockerase successful on %s\n", uflash0);
/* erase 4 blocks starting from block 5 of user flash 1 */
  uf1.blk_num = 5;
  uf1.num_of_blks = 4;
  if (ufd1 && ioctl(ufd1, UIOCEBLK, &ufd1) == -1 ) {
    perror("ioctl(ufd1, UIOCEBLK): ");
    return(-1);
  }
  printf("\nblockerase successful on %s\n", uflash1);
return(0);
}
main() {
  int ret;
module = argv[0];
ret = uflash_init();
if (!ret)
    uflash_blockerase();
uflash_uninit();
}
```

Sample User Flash Application Program

You can use the following program to test the user flash device and driver. This program also demonstrates how this device can be used.

CODE EXAMPLE 2-7 Sample User Flash Application Program

```
/*
 *
 *        This application program demonstrates the user program
 *        interface to the User Flash PROM driver.
 *
 *       One can read or write a number of bytes up to the size of
 *        the user PROM by means of pread() and pwrite() calls.
 *        All other functions of the PROM can be reached by the
means
 *        of ioctl() calls such as:
 *         -) identify the chip,
 *         -) query block,
 *         -) lock block/unlock block,
 *         -) master lock,
 *         -) erase block, erase all unlocked blocks, and
```

CODE EXAMPLE 2-7 Sample User Flash Application Program *(Continued)*

```
 *            erase whole PROM
 *        Please note that not all of the above ioctl calls are
 *        available for all flash PROMs. It is the user's
responsibility
 *       to find out the features of a given PROM. The type, block
size,
 *        and number of blocks of the PROM are returned by
 *"identify" ioctl().
 *
 *        The pwrite() erases the block[s] and then does the .
writing.
 *       The driver uses the buffered write. If the buffered write
 *        is not supported in a particular PROM, the non-buffered
 *        writes are used instead. The buffered write is 15 folds
 *        faster than the non-buffered write.
 *
 *       Use the following line to compile your custom application
 *        programs:
 *             make uflash_test
 */

#pragma ident   "@(#)uflash_test.c 1.3     99/08/03 SMI"

#include <stdio.h>
#include <sys/signal.h>
#include <stdio.h>
#include <sys/time.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/stream.h>
#include "uflash_if.h"
/*
      * PROM size: 4 or 8 MBytes
      * Uncomment the right block
      */
     #if 1
     #define PROM_SIZE 0x400000 /* 4 MBytes */
     #endif
     #if 0
     #define PROM_SIZE 0x800000 /* 8 MBytes */
     #endif
static char *help[14] = {
```

CODE EXAMPLE 2-7 Sample User Flash Application Program *(Continued)*

```
        "0 -- read     user flash PROM",
        "1 -- write    user flash PROM",
        "2 -- identify user flash PROM",
        "3 -- query    blocks",
        "4 -- lock     blocks",
        "5 -- master   lock",
        "6 -- clear    all locks",
        "7 -- erase    blocks",
        "8 -- erase    all unlocked blocks",
        "9 -- erase    whole PROM",
        "a -- switch   PROMs",
        "q -- quit",
        "?/h -- display this menu",
        ""
};

/*char            get_cmd(); */

static char
get_cmd()
{
           char    buf[10];
           gets(buf);
           return (buf[0]);
}

/*
 * Main
 */
main(int argc, char *argv[])
{
       int        n_byte;    /* returned from pread/pwrite */
       int        size, offset, pat;
       int        fd0, fd1, h, i;
       int        fd, prom_id;
       uflash_if_tuflash_if;
       caddr_t r_buf, w_buf;
       char    *devname0 = "/dev/uflash0";
       char    *devname1 = "/dev/uflash1";
       char    c;

       /*
        * Assume that the PROM size is 4 MB.
```

CODE EXAMPLE 2-7 Sample User Flash Application Program *(Continued)*

```
         */
        r_buf = (caddr_t)malloc(PROM_SIZE);
        w_buf = (caddr_t)malloc(PROM_SIZE);

        /*
         * Open the user flash PROM #0.
         */
        if ((fd0 = open(devname0, O_RDWR)) < 0) {
            fprintf(stderr, "couldn't open device: %s\n",
devname0);
            exit(1);
        }
        /*
         * Open the user flash PROM #1.
         */
        if ((fd1 = open(devname1, O_RDWR)) < 0) {
            fprintf(stderr, "couldn't open device: %s\n",
devname1);
            exit(1);
        }

        /* set the default PROM */
        prom_id = 0;
        fd = fd0;

        /* let them know about the help menu */
        fprintf(stderr, "Enter <h> or <?> for help on commands\n");

        while (1) {
            fprintf(stderr, "[%d]command> ", prom_id);

            switch(get_cmd()) {
            case 'q':
                goto getout;

            case 'h':
            case '?':
                h = 0;
                while (*help[h]){
                    fprintf(stderr, "%s\n", help[h]);
                    h++;
                }
                break;
```

CODE EXAMPLE 2-7 Sample User Flash Application Program *(Continued)*

```
case 'a':/* switch PROM */
    fd = (fd == fd0)? fd1: fd0;
    prom_id = (prom_id == 1)? 0: 1;
    break;

case '9':       /* erase the whole flash PROM */
    fprintf(stderr,
                        "Are you sure?[y/n]");
                scanf ("%c", &c);

    if (c != 'y')
        continue;
    if (ioctl(fd, UIOCEFUL, &uflash_if) == -1)
                        goto getout;
    break;

case '8':      /* erase all unlocked flash PROM blocks */
    /*
     * This ioctl is valid only for those
     * chips that have query command.
     */
    if (ioctl(fd, UIOCEALL, &uflash_if) == -1)
                        goto getout;
    break;

case '7':       /* erase flash PROM block */
    fprintf(stderr,
                    "Enter PROM block number[0, 31]> ");
                scanf ("%d", &uflash_if.blk_num);

    fprintf(stderr,
        "Enter number of block> ");
    scanf ("%d", &uflash_if.num_of_blks);

    if (ioctl(fd, UIOCEBLK, &uflash_if) == -1)
                        goto getout;
    break;

case '6':       /* clear all locks */
    /* on certain PROMs */
    if (ioctl(fd, UIOCCLCK, &uflash_if) == -1)
                        goto getout;
```

CODE EXAMPLE 2-7 Sample User Flash Application Program *(Continued)*

```
            break;

        case '5':       /* master lock */
            /* on certain PROMs */
            if (ioctl(fd, UIOCMLCK, &uflash_if) == -1)
                            goto getout;
            break;

        case '4':       /* lock flash PROM block */
            /* on certain PROMs */
            fprintf(stderr,
                            "Enter PROM block number[0, 31]> ");
                        scanf ("%d", &uflash_if.blk_num);

            fprintf(stderr,
                "Enter number of block> ");
            scanf ("%d", &uflash_if.num_of_blks);

            if (ioctl(fd, UIOCLBLK, &uflash_if) == -1)
                            goto getout;
            break;

        case '3':       /* query flash PROM */
            /* on certain PROMs */
            fprintf(stderr,
                            "Enter PROM block number[0, 31]> ");
                        scanf ("%d", &uflash_if.blk_num);

            fprintf(stderr,
                "Enter number of block> ");
            scanf ("%d", &uflash_if.num_of_blks);

            if (ioctl(fd, UIOCQBLK, &uflash_if) == -1)
                            goto getout;
            for (i = uflash_if.blk_num;
                i < (uflash_if.blk_num+uflash_if.num_of_blks);
                i++)
            {
                fprintf(stderr, "block[%d] status = %x\n",
                    i, uflash_if.info.blk_status[i] & 0xF);
            }
            break;
```

CODE EXAMPLE 2-7 Sample User Flash Application Program *(Continued)*

```
            case '2':       /* identify flash PROM */
                if (ioctl(fd, UIOCIBLK, &uflash_if) == -1)
                                goto getout;
                fprintf(stderr, "manufacturer id = 0x%x, device id
=\
                            0x%x\n# of blks = %d, blk size = 0x%x\n",
                        uflash_if.info.mfr_id & 0xFF,
                        uflash_if.info.dev_id & 0xFF,
                        uflash_if.info.blk_num,
                        uflash_if.info.blk_size);
                break;

            case '1':       /* write to user flash PROM */
                fprintf(stderr,
                              "Enter PROM offset[0, 0xXX,XXXX]> ");
                          scanf ("%x", &offset);

                fprintf(stderr,
                    "Enter number of bytes[hex]> ");
                scanf ("%x", &size);

                fprintf(stderr,
                                  "Enter data pattern[0, 0xFF]> ");

                          scanf ("%x", &pat);

                /*
                 * init write buffer.
                 */
                for (i = 0; i < size; i++) {
                    w_buf[i] = pat;
                }

                n_byte = pwrite (fd, w_buf, size, offset);
                if (n_byte != size) {
                    /* the write failed */
                    printf ("Write process was failed at byte 0x%x \
n",
                        n_byte);
                }
                break;

            case '0':/* read from user flash PROM */
```

CODE EXAMPLE 2-7 Sample User Flash Application Program *(Continued)*

```
                fprintf(stderr,
                                "Enter PROM offset[0, 0xXX,XXXX]> ");
                          scanf ("%x", &offset);

                fprintf(stderr,
                    "Enter number of bytes[hex]> ");
                scanf ("%x", &size);

                getchar();/* clean up the char buf */

                n_byte = pread (fd, r_buf, size, offset);
                if (n_byte != size) {
                                    /* the read failed */
                                printf ("Read process was failed at \
                                byte 0x%x \n",
                                           n_byte);
                    continue;
                           }

                printf ("\nuser data buffer:\n");
                for (i = 0; i < size; i++) {
                    printf("%2x ", r_buf[i] & 0xff);
                }
            printf("\n");

            default:
                continue;
            }
        }

        /* exit */
getout:
        close(fd0);
        close(fd1);
        return;

} /* end of main() */
```

CHAPTER 3

Advanced System Management

Advanced System Monitoring (ASM) is an intelligent fault detection system that increases uptime and manageability of the board. The System Management Controller (SMC) module on the Netra CP2000/CP2100 series supports the temperature monitoring functions of ASM. This chapter describes the specific ASM functions of the Netra CP2000/CP2100 series. This chapter includes the following sections:

ASM Component Compatibility

TABLE 3-1 lists the compatible ASM hardware, OpenBoot PROM, and Solaris operating environment for the Netra CP2000/CP2100 series.

TABLE 3-1 Compatible Netra CP2000/CP2100 Series ASM Components

Component	ASM Compatibility
Hardware	All board versions support ASM
OpenBoot PROM	ASM is supported by OpenBoot PROM.
Operating environment	Solaris 8 2/02 operating environment or subsequent compatible versions, with one of the following CD supplements: • *CP2000 Supplemental CD 4.0 for Solaris 8* • *CP2000 Supplemental CD 3.1 for Solaris 8*

Typical ASM System Application

FIGURE 3-1 illustrates the Netra CP2000/CP2100 series ASM application block diagram.

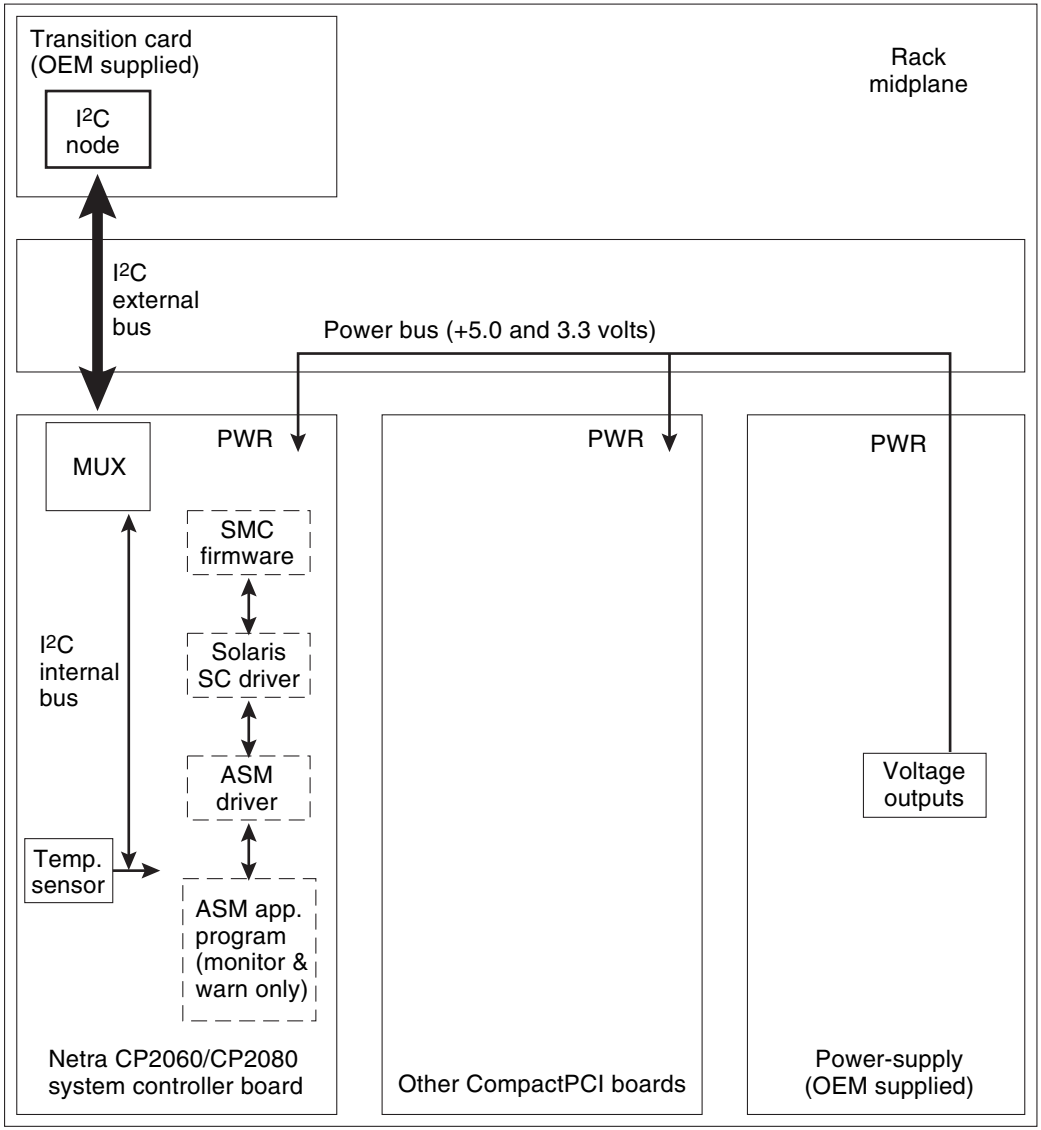


FIGURE 3-1 Typical Netra CP2000/CP2100 Series ASM Application Block Diagram

FIGURE 3-1 is a typical Netra CP2000/CP2100 series system application block diagram. For locations of the temperature sensors, see FIGURE 3-2, FIGURE 3-3 and FIGURE 3-4.

The Netra CP2000/CP2100 series functions as a system controller board or as a satellite board in a CompactPCI system rack. The Netra CP2000/CP2100 series board monitors its CPU-vicinity temperature and issues warnings at both the OpenBoot PROM and Solaris operating environment levels when these environmental readings are out of limits. At the Solaris operating environment level, the application program monitors and issues warnings for the system controller and the satellite board. In the host and satellite modes of operation, at the OBP level, the CPU vicinity temperature is monitored if the the NVRAM variable *env-monitor* is enabled.

Typical Cycle From Power Up to Shutdown

This section describes a typical ASM cycle from power up to shutdown.

ASM Protection at the OpenBoot PROM

The OpenBoot PROM monitors CPU-vicinity temperature at the fixed polling rate (from the `env-mon-interval` parameter) of 10 seconds and the OpenBoot PROM displays warning messages on the default output device whenever the measured temperature exceeds the pre-programmed NVRAM module configurable variable warning temperature (the `warning-temperature` parameter) or the pre-programmed NVRAM module configurable variable shutdown temperature (the `shutdown-temperature` parameter). See "OpenBoot PROM Environmental Parameters" on page 57 for information on changing these pre-programmed parameters.

The OpenBoot PROM cannot shut down power to the Netra CP2000/CP2100 series board. The shutdown temperature message is only a warning message to the user that the Netra CP2000/CP2100 series board is overheating and needs to be shut down immediately by external means.

OpenBoot PROM-level protection takes place only when the `env-monitor` parameter is enabled (it is not the default setting). Disabling `env-monitor` completely disables ASM protection at the OpenBoot PROM level but does not affect ASM protection at the Solaris operating environment level.

Note – To protect the system at OpenBoot PROM level, the `env-monitor` should be enabled at all times.

ASM Protection at the Operating Environment Level

Monitoring changes in the ASM temperatures can be a useful tool for determining problems with the room where the system is installed, functional problems with the system, or problems on the board. Establishing baseline temperatures early in deployment and operation could be used to trigger alarms if the temperatures from the sensors increase or decrease dramatically. If all the sensors go to room ambient, power has probably been lost to the host system. If one or more sensors rise in temperature substantially, there may be a system fan malfunction, the system cooling may have been compromised, or room air conditioning may have failed.

When the application program opens the system controller device and pushes the ASM streams module, the ASM module is loaded.

To access the CPU-vicinity temperature measurements at the Solaris operating environment level, use the `ioctl` system call in an application program. To specify the ASM polling rate, use the `sleep` system call.

Protection at the operating environment level takes place only when the ASM application program is running, which is initiated by the end user. Failure to run the ASM application program completely disables ASM protection at the Solaris level but does not affect ASM protection at the OpenBoot PROM level. Keep the ASM application program running at all times.

In a typical ASM application program, the software reads the following temperature sensors once every polling cycle:

- *Netra CP2040/CP2060/CP2080/CP2140 boards*: CPU, heat sink, board memory, power module, SDRAM memory module 1
- *Netra CP2080 boards only*: SDRAM memory module 2
- *Netra CP2160 boards*: CPU, inlet 1, exhaust 1, exhaust 2, power module, and SDRAM module 1

The program then compares the measured CPU-vicinity temperature with the warning temperature and displays a warning message on the default output device whenever the warning temperature is exceeded.

The program can also issue a shutdown message on the default output device whenever the measured CPU-vicinity temperature exceeds the shutdown temperature. In addition, the ASM application program can be programmed to sync and shut down the Solaris operating environment when conditions warrant.

The use of system calls to access the ASM device driver at the Solaris level enables OEMs to implement their own monitoring, warning, and shutdown policies through a high-level programming language such as the C programming language. An OEM can log and analyze the environmental data for trends (such as drift rate or sudden

changes in average readings). Or, an OEM can communicate the occurrence of an unusual condition to a specialized management network using the Netra CP2000/CP2100 series board Ethernet port.

Refer to "Sample Application Program" on page 71 for an example of how a simple ASM monitoring program can be implemented.

The power module is controlled by the SMC subsystem (except for automatic controls such as overcurrent shutdown or voltage regulation). The functions controlled are core voltage output level and module on/off state.

Post Shutdown Recovery

The onboard voltage controller is a hardware function that is not controlled by either firmware or software. At the OpenBoot PROM level, there is no mechanism for the OpenBoot PROM to either remove or restore power to the Netra CP2000/CP2100 series board when the CPU-vicinity temperature exceeds its maximum recommended level.

There is no mechanism for the Solaris operating environment to either recover or restore power to the Netra CP2000/CP2100 series board when an unusual condition occurs (for example, if the CPU-vicinity temperature exceeds its maximum recommended level). In either case, the end user must intervene and manually recover the Netra CP2000/CP2100 series board as well as the CompactPCI system through hardware control.

Hardware ASM Functions

This section summarizes the hardware ASM features on the Netra CP2000/CP2100 series board. TABLE 3-2 lists the ASM functions and shows the location of the ASM hardware on a *typical* Netra CP2060 board. TABLE 3-3 shows the same information for the Netra CP2160 board.

Note that in TABLE 3-2 and TABLE 3-3 the readings for the SDRAM modules show the sensor readings as currently unavailable because the tables list information of a typical Netra board that does not support memory modules.

TABLE 3-2 Typical Netra CP2060 Hardware ASM Functions

Function	Capability
PMC Temperature	Senses the PMC temperature
CPU heat sink	Senses the temperature of the heat sink
Netra CP2060 Memory	Senses the temperature of Netra CP2060 memory module
SDRAM module#1 Temperature (for Netra boards with memory modules)	Sensor reading is currently unavailable*
SDRAM module#2 Temperature (for Netra boards with memory modules)	Sensor reading is currently unavailable
Power Module Temperature	Senses the temperature of the power module

* This reading would be available on a typical Netra board that supports memory modules.

TABLE 3-3 Typical Netra CP2160 Hardware ASM Functions

Function	Capability
Board exhaust air temperature #1	Senses the board exhaust air temperature
Board exhaust air temperature #2	Senses the board exhaust air temperature
CPU sensor temperature	Senses the CPU sensor temperature
Board inlet air temperature	Senses the board inlet air temperature
SDRAM module #1 temperature (for Netra boards with memory modules)	Sensor reading is currently unavailable*
Power module temperature	Senses the temperature of the power module

* This reading would be available on a typical Netra board that supports memory modules.

TABLE 3-4 Local I^2C Bus

Function	Device
I^2C Multiplexer	PCA9540
CPU-vicinity temperature	MAX1617
Inlet 1	MAX1617
Exhaust 1	MAX1617
Exhaust 2	MAX1617
General I/O*	PCF8574
FRU ID	AT24C64 EEPROM
Ethernet ID	AT24C64 EEPROM
SDRAM module 1 temperature	MAX1617
SDRAM module 1 ID	AT24C64 EEPROM
SDRAM module 2 temperature	MAX1617
SDRAM module 2 ID	AT24C64 EEPROM
Power module temperature	DS1721
Power module†	PCF8574
Power module ID	AT24C64 EEPROM

* General Purpose I/O bit assignments:
P7 = Input; CPU EPD
P6 = Input; PLD_FLASH0_SEL
P5 = Input; PLD_FLASH1_SEL
P4 = Input; VID<0>
P3 = Input; VID<1>
P2 = Input; VID<2>
P1 = Input; VID<3>
P0 = Not used, not connected.

† Power module interface gives control of 4-bit V_{ID} setting.

FIGURE 3-2, FIGURE 3-3, FIGURE 3-4 and FIGURE 3-5 show the location of the ASM hardware on the Netra CP2000/CP2100 series boards.

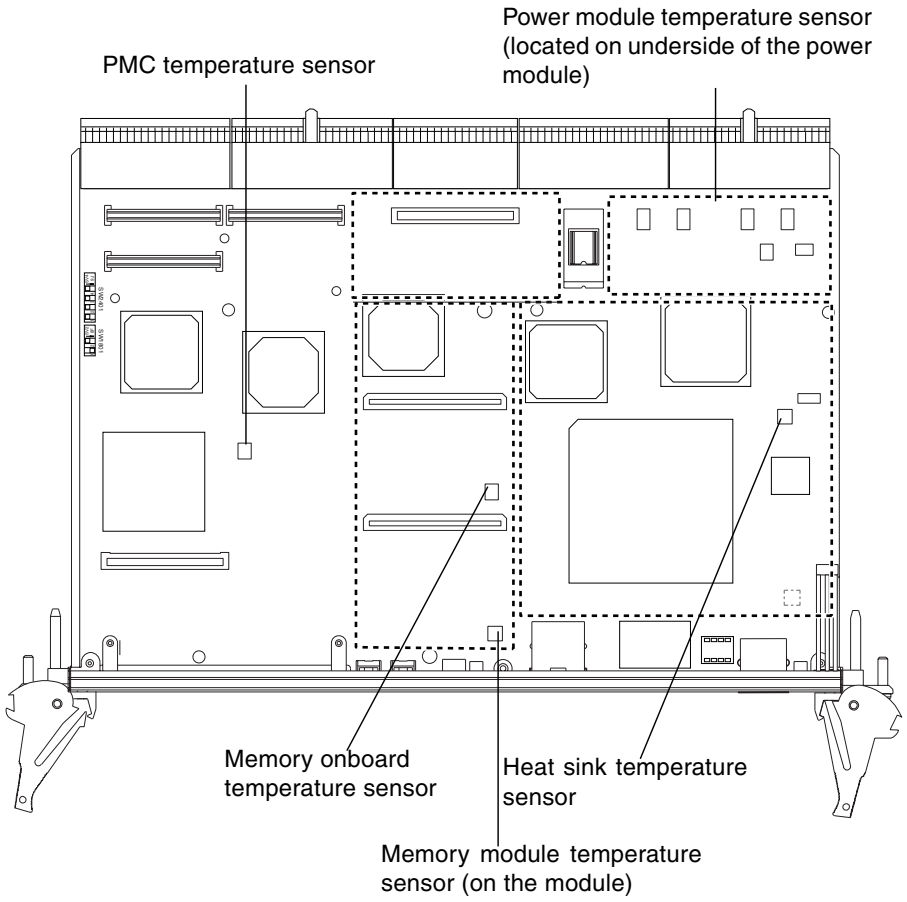


FIGURE 3-2 Location of ASM Hardware on the Netra CP2040/CP2140 Board

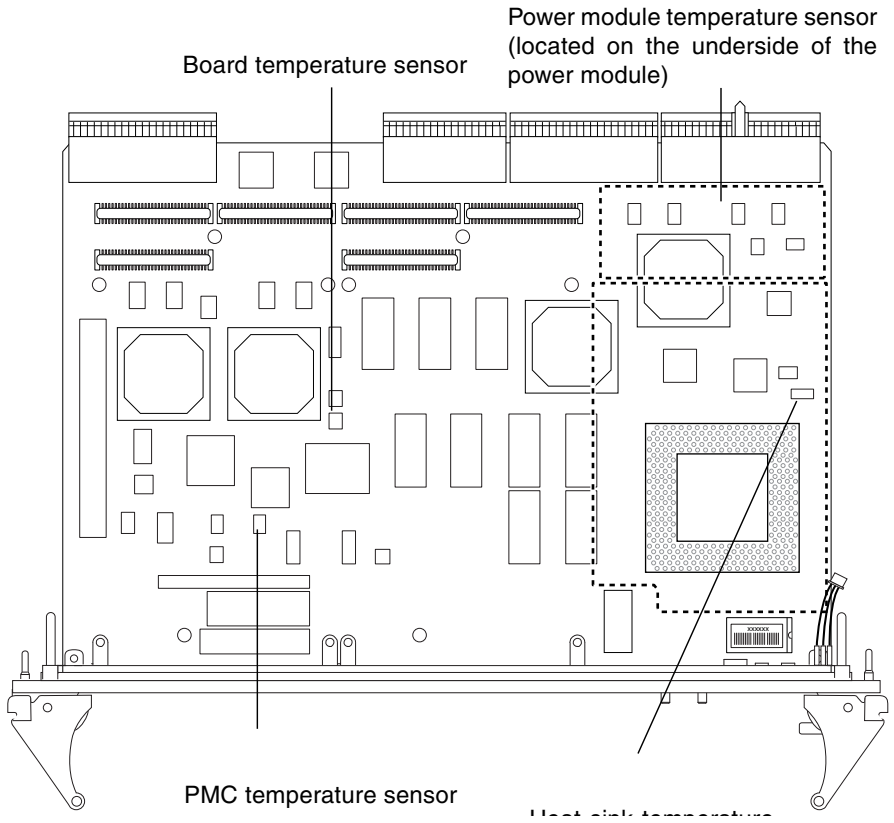


FIGURE 3-3 Location of ASM Hardware on the Netra CP2060 Board

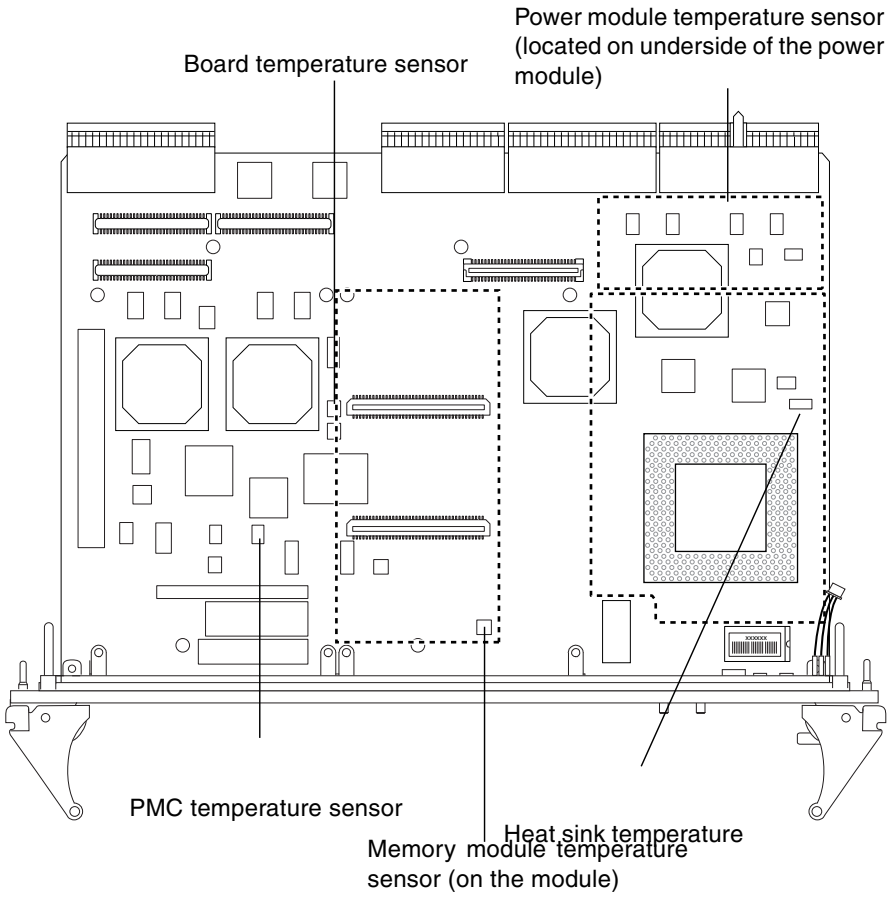


FIGURE 3-4 Location of ASM Hardware on the Netra CP2080 Board

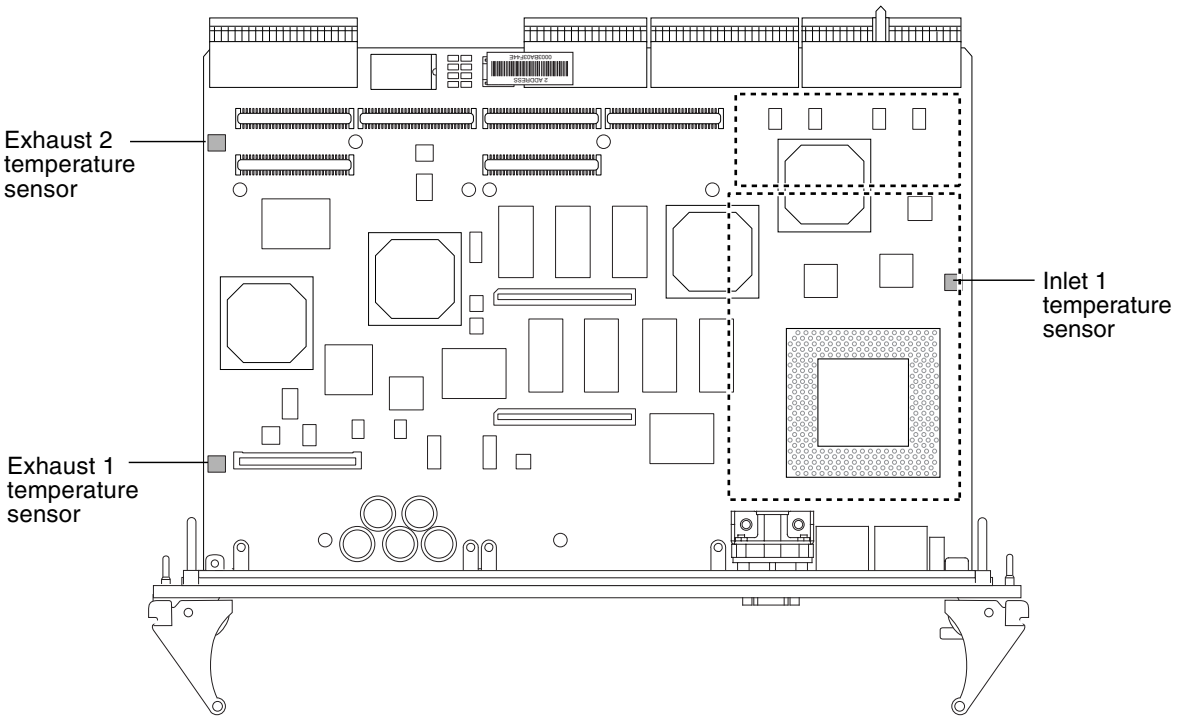


FIGURE 3-5 Location of ASM Hardware on the Netra CP2160 Board

FIGURE 3-6 is a block diagram of the ASM functions.

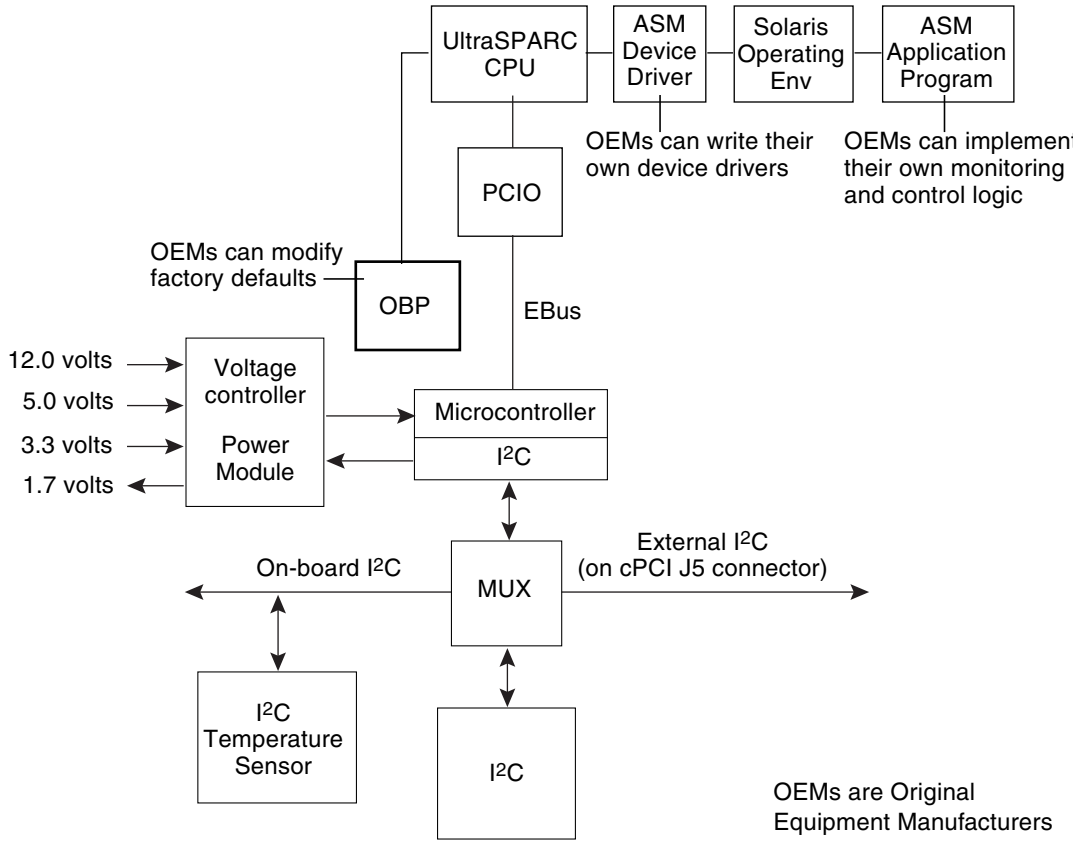


FIGURE 3-6 Netra CP2000/CP2100 Series ASM Functional Block Diagram

CPU-Vicinity Temperature Monitoring

The Netra CP2040/CP2060/CP2080/CP2140 boards use a MAX1617 temperature sensor located near the CPU underneath its heat sink. The Netra CP2160 board does not have this temperature sensor.

Power On/Off Switching

The onboard voltage controller allows power to the rest of the Netra CP2000/CP2100 series board only when the following conditions are met:

- The VDD core-1.7-volt supply voltage is greater than 1.53 volts (within 10% of nominal).
- The 12-volt supply voltage is greater than 10.8 volts (within 10% of nominal).
- The 5-volt supply voltage is greater than 4.5 volts (within 10% of nominal)
- The 3.3-volt supply voltage is greater than 3.0 volts (within 10% of nominal).

The controller requires these conditions to be true for at least 100 milliseconds to help ensure the supply voltages are stable. If any of these conditions become untrue, the voltage monitoring circuit shuts down the power of the board.

Inlet/Exhaust Temperature Monitoring

The inlet board temperature sensor can be used to ensure that the maximum allowable short-term system-level air inlet temperature is not exceeded. The sensor can also be used to monitor potential issues with the system or installation, since inlet temperature for the Netra CP2160 board should be kept low for the installation reliability requirements.

The two exhaust temperature sensors can be used to ensure that the proper airflow across the board is being maintained. The difference in the temperature between the inlet air temperature and exhaust temperatures can be monitored to determine if system filters need servicing, if air movers have failed, or if an electrical problem has occured due to components drawing too much power on the board.

During normal operation of the Netra CP2160 board, any sudden, sustained, or substantial changes in the delta temperature across the board can be used to alert service personnel to a potential system or board service issue.

CPU Sensor Temperature Monitoring

The CPU sensor temperature can be used to prevent damage to the board by shutting the board down if this sensor exceeds predetermined limits.

Adjusting the ASM Warning and Shutdown Parameter Settings on the Board

The Netra CP2000/CP2100 board uses the Advanced System Monitoring (ASM) detection system to monitor the temperature of the board. The ASM system will display messages if the board temperature exceeds the set warning and shutdown settings. Because the on-board sensors may report different temperature readings for different system configurations and airflows, you may want to adjust the warning and shutdown temperature parameter settings.

The CP2000/CP2100 board determines the board temperature by retrieving temperature data from sensors located on the board. A board sensor reads the temperature of the immediate area around the sensor. Although the software may appear to report the temperature of a specific hardware component, the software is actually reporting the temperature of the area near the sensor. For example, the CPU heat sink sensor reads the temperature at the location of the sensor and not on the actual CPU heat sink. The board's OpenBoot PROM collects the temperature readings from each board sensor at regular intervals. You can display these temperature readings using the `show-sensors` OpenBoot PROM command. See "show-sensors Command at OpenBoot PROM" on page 61

The temperature read by the CPU heat sink sensor will trigger OpenBoot PROM warning and shutdown messages. When the CPU heat sink sensor reads a temperature greater than the warning parameter setting, the OpenBoot PROM will display a warning message. Likewise, when the sensor reads a temperature greater than the shutdown setting, the OpenBoot PROM will display a shutdown message.

Many factors affect the temperature readings of the sensors, including the airflow through the system, the ambient temperature of the room, and the system configuration. These factors may contribute to the sensors reporting different temperature readings than expected.

TABLE 3-5 shows the sensor readings of a *typical* Netra CP2040 board operating in a Sun server in a room with an ambient temperature of 21˚C. The temperature readings were reported using the `show-sensors` OpenBoot PROM command. Note that the reported temperatures are higher than the ambient room temperature.

TABLE 3-5 Reported Temperature Readings at an Ambient Room Temperature of 21˚C on a Typical Netra CP2040 Board*

Board Sensor Location	Reported Temperatures (in Degrees Celsius)	Difference Between Reported and Ambient Room Temperature (in Degrees Celsius)
CPU heat sink	28˚C	7˚C
PMC	33˚C	12˚C
Board heat sink	29˚C	8˚C
Board memory	37˚C	16˚C
SDRAM module 1	42˚C	21˚C
SDRAM module 2	36˚C	15˚C
Power module	34˚C	13˚C

* Other boards will have differnt but similar readings.

TABLE 3-6 shows the sensor readings of a typical Netra CP2160 board, which has different sensor locations than those on the other Netra CP2000/CP2100 series boards.

Note that the inlet temperature sensor typically does not capture true board inlet temperature due to the heat of nearby components. For typical Netra CP2000/CP2100 series systems, subtract 4˚C from the temperature sensor value. Note that the temperature sensor has an accuracy of up to plus or minus 2˚C. Users should conduct their own temperature sensor tests to obtain accurate readings.

TABLE 3-6 Reported Temperature Readings at an Ambient Room Temperature of 21˚C on a Typical Netra CP2160 Board

Board Sensor Location	Reported Temperatures (in Degrees Celsius)	Difference Between Reported and Ambient Room Temperature (in Degrees Celsius)
CPU sensor temperature	37˚C	16˚C
Board inlet air temperature	34˚C	13˚C
Board exhaust air temperature #1	35˚C	14˚C

TABLE 3-6 Reported Temperature Readings at an Ambient Room Temperature of 21˚C on a Typical Netra CP2160 Board

Board Sensor Location	Reported Temperatures (in Degrees Celsius)	Difference Between Reported and Ambient Room Temperature (in Degrees Celsius)
Board exhaust air temperature #2	35˚C	14˚C
SDRAM module #1 temperature	33˚C	12˚C
Power module temperature	25˚C	4˚C

Since the temperature reported by the CPU sensor might be different than the actual CPU die temperature, you may want to adjust the settings for both the `warning-temperature` and `shutdown-temperature` OpenBoot PROM parameters. The default values of these parameters have been conservatively set at 70˚C for the warning temperature and 80˚C for the shutdown temperature.

Note – If you have developed an application that uses the ASM software to monitor the temperature sensors, you may want to adjust your application's settings accordingly.

OpenBoot PROM Environmental Parameters

This section describes how to change the OpenBoot PROM environmental monitoring parameters. These global OpenBoot PROM parameters do not apply at the Solaris level. Instead, the ASM application program provides equivalent parameters that do not necessarily have to be set to the same values as their OpenBoot PROM counterparts. Refer to "ASM Application Programming" on page 68 for information about using ASM at the Solaris level. The OpenBoot PROM polling rate is at fixed intervals of 10 seconds.

OpenBoot PROM Warning Temperature Parameter

OBP programs SMC for temperature monitoring using the sensor commands. TABLE 3-7 lists the default threshold temperature settings for the CP2000/CP2100 series boards.

TABLE 3-7 Default Threshold Temperature Settings

Netra cPCI Board	Default Threshold Temperature Settings for Netra Boards (In Degrees Celsius)		
	Warning Temperature	Critical Temperature	Shutdown Temperature
Netra CP2060/CP2080 Board	60	not applicable	65
Netra CP2040 Board	60	not applicable	65
Netra CP2140 Board	60	65	70
Netra CP2160 Board	70	75	80

For example, on a Netra CP2160 there are three NVRAM variables that provide different temperature levels. The critical-temperature limit lies between warning and shutdown thresholds. The default values of these temperature thresholds and corresponding action is shown in TABLE 3-8:

TABLE 3-8 Typical Netra CP2160 Board Temperature Thresholds and Firmware Action

Thresholds With Default	Firmware Action
warning-temperature = 70˚ C	OBP displays warning message
critical-temperature=75˚ C	OBP displays warning message
shutdown-temperature=80˚ C	SMC shuts down the CPU processor and the Netra CP2160 board

Note that there is a lower limit of 50˚ C on shutdown-temperature value. If the temperature is set to a value lower than 50˚ C, OpenBoot PROM resets it back to 50˚ C in SMC. However, OpenBoot PROM does not reset the NVRAM variable *shutdown-temperature* to 50˚ C. Therefore, everytime the user resets the system, the OpenBoot PROM displays a warning message similar to the message below:

```
WARNING!!! shutdown-temperature is set too low at 40˚ C. Setting
the threshold at a safer value of 50˚ C.
```

This safeguards against a user setting the shutdown-temperature lower than the room temperature and thereby causing the CPU processor and the Netra CP2160 board to be powered off by SMC on the next reset.

The `warning-temp` global OpenBoot PROM parameter determines the temperature at which a warning is displayed. The `shutdown-temperature` global OpenBoot PROM parameter determines the temperature at which the system is shut down. The temperature monitoring environment variables can be modified at the OpenBoot PROM command level as shown in examples below:

```
ok setenv warning-temperature 71
```

OR,:

```
ok setenv shutdown-temperature 82
```

The critical-temperature is a second-level warning temperature with a default value of 75° C. This variable can be modified using the OpenBoot PROM level *setenv* command as shown in example below::

```
ok setenv critical-temperature 76
```

OpenBoot PROM/ASM Monitoring

This section describes the ASM monitoring in the OpenBoot PROM. Please note that the figures in the examples below are for a typical Netra CP2160 board.

CPU Sensor Monitoring

The following NVRAM module variables are in OpenBoot PROM for ASM *for a typical Netra CP2160 board*:

- NVRAM module variable name: `env-monitor`
  - Function: enables or disables environment monitoring at OpenBoot PROM
  - Data type: string
  - Valid values: disabled or enabled
  - Default value: disabled
  - OpenBoot PROM usage:

```
ok setenv env-monitor disabled or enabled
```

- NVRAM module variable name: `warning-temperature`
  - Function: sets the CPU warning temperature threshold
  - Data type: byte
  - Unit: decimal
  - Default value: 70
  - OpenBoot PROM usage:

```
ok setenv warning-temperature temperature-value
```

- NVRAM module variable name : `critical-temperature`
  - Function: sets the CPU critical temperature threshold
  - Data type: byte
  - Unit: decimal
  - Default value: 75
  - OpenBoot PROM usage:

```
ok setenv critical-temperature temperature-value
```

- NVRAM module variable name: `shutdown-temperature`
  - Function: sets the CPU shutdown temperature threshold
  - Data type: byte
  - Unit: decimal
  - Default value: 80
  - OpenBoot PROM usage:

```
ok setenv shutdown-temperature temperature-value
```

Caution – Exercise caution while setting the above two parameters. Setting these values too high will leave the system unprotected against system over-heat.

Warning Temperature Response at OpenBoot PROM

When the CPU-vicinity temperature reaches “warning-temperature,” a similar message is displayed at the `ok` prompt at a regular interval:

```
Temperature sensor #2 has threshold event of

<<< WARNING!!! Upper Non-critical - going high >>>

The current threshold setting is : 70

The current temperature is : 71
```

Critical Temperature Response at OpenBoot PROM

When the CPU-vicinity temperature reaches “warning-temperature”, a similar message is displayed at the `ok` prompt at a regular interval:

```
Temperature sensor #2 has threshold event of

<<< !!! ALERT!!! Upper Critical - going high >>>

The current threshold setting is : 75

The current temperature is : 76
```

`show-sensors` Command at OpenBoot PROM

The `show-sensors` command at OpenBoot PROM displays the readings of all the temperature sensors on the board TABLE 3-9 shows typical sensor readings for a Netra CP2060 board (which would be similar to the Netra CP2040/CP2080/CP2140 boards) and TABLE 3-10 shows typical sensor readings for a Netra CP2160 board.

TABLE 3-9 OpenBoot PROM Sensor Reading Typical for a Typical Netra CP2060 Board

Sensor	Name	Current Reading
2	CPU-vicinity temperature (senses the local temperature of the CPU area)	28°C
3	PMC temperature	29°C
4	Motherboard Heat Sink temperature	33°C
5	Motherboard memory temperature for Netra C2060	32°C
a	SDRAM module#1 temperature for Netra CP2080	This sensor reading is not available*
c	SDRAM module#2 temperature for Netra CP2080	This sensor reading is not available
e	Power module temperature	25°C

* The readings are from a typical Netra CP2060 board which does not support memory modules.

TABLE 3-10 OpenBoot PROM Sensor Reading Typical for a Typical Netra CP2160 Board

Sensor	Name	Current Reading
2	CPU	37°C
3	Inlet 1	34°C
4	Exhaust 1	35°C
5	Exhaust 2	35°C
a	SDRAM module 1	33°C
e	Power module	25°C

IPMI Command Examples at OpenBoot PROM

The Intelligent Platform Management Interface (IPMI) commands can be used to enable the sensors monitoring and subsequent event generation from satellite boards in the Netra CP2000/CP2100 series CompactPCI system.

The IPMI command examples provided in this section are based on the *IPMI Specification Version 1.0*. Please use the IPMI Specification for additional information on how to implement these IPMI commands.

Note – To execute an IPMI command, at the OpenBoot PROM `ok` prompt, type the packets *in reverse order* followed by the relevant information as shown in examples in "Examples of IPMI Command Packets" on page 64. Change the bytes in the example packet to accommodate different IPMI addresses, different threshold values or different sensor numbers. See also the *IPMI Specification Version 1.0.*

▼ Set or Change the Thresholds for a Sensor

1. **Set the thresholds for the sensors.**

   See "Set Sensor Threshold" on page 64. If no threshold is set, the default threshold operates:

```
ok packet bytes number-of-bytes-in-packet 34 execute-smc-cmd
```

2. **Follow instructions in "Check Whether the IPMI Commands Are Executed Properly" on page 63 to check proper execution of the command.**

▼ Enable Events From a Sensor

1. **To execute a command to enable events from the sensor, type:**

```
ok packet bytes number-of-bytes-in-packet 34 execute-smc-cmd
```

   See "Set Sensor Event Enable Command" on page 66 and "Get Sensor Event Enable" on page 67.

   There are supporting commands for any sensor and the corresponding packets at these commands: `get sensor threshold`, `get sensor reading`, and `get sensor event enable`.

2. **Follow instructions in "Check Whether the IPMI Commands Are Executed Properly" on page 63 to check proper execution of the command.**

▼ Check Whether the IPMI Commands Are Executed Properly

1. **Check whether the stack on the `ok` prompt displays `0` when the command is issued.**

   A 0 indicates that the command packet sent to the board was successful.

2. **Type** `execute-smc-cmd (cmd 33)` **command at the** `ok` **prompt as follows:**

```
ok 0 33 execute-smc-cmd
```

This command verifies that the target satellite board received and executed the command and sent a response.

3. **Check the completion code which is the seventh byte from left.**

If the completion code is 0, then the target board successfully executed the command. Otherwise the command was not successfully executed by the board.

4. **Check that rsSA and rqSA are swapped in the response packet.**

The rsSA is the responder slave address and the rqSA is the requestor slave address.

5. **(Optional) If command not correctly executed, resend the IPMI command.**

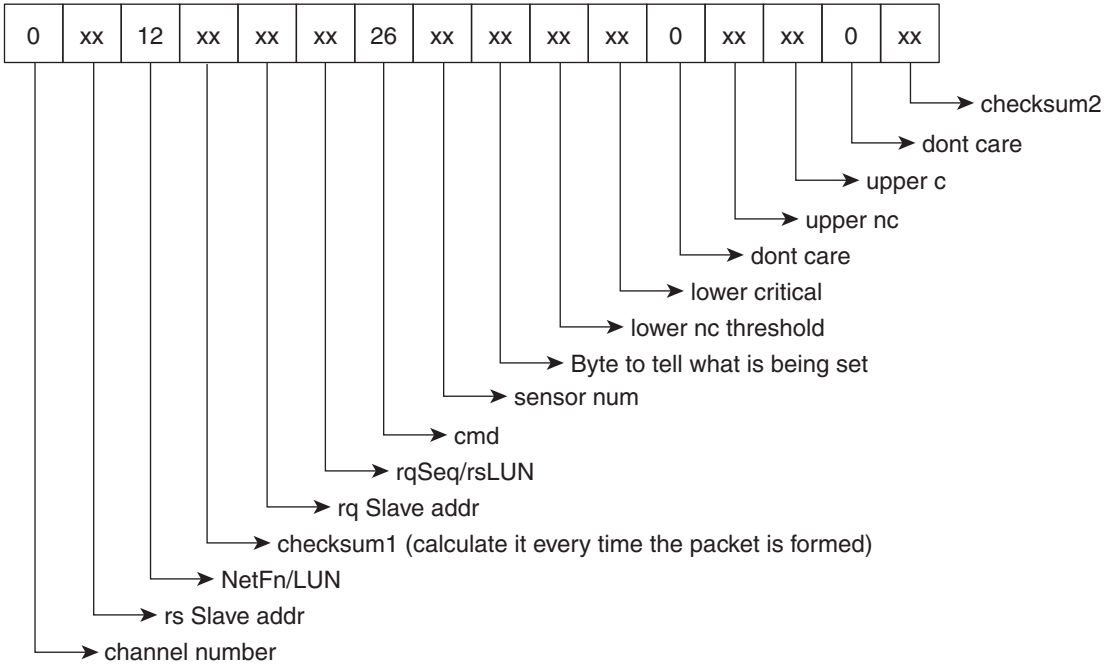
Examples of IPMI Command Packets

The following packets are IPMI command packets that can be sent from the OpenBoot PROM `ok` prompt:

Set Sensor Threshold

A typical example of the sensor command is as follows:

```
37 0 41 10 0 0 3 1b 2 26 12 20 34 12 ba 0 10 34 execute-smc-cmd
```

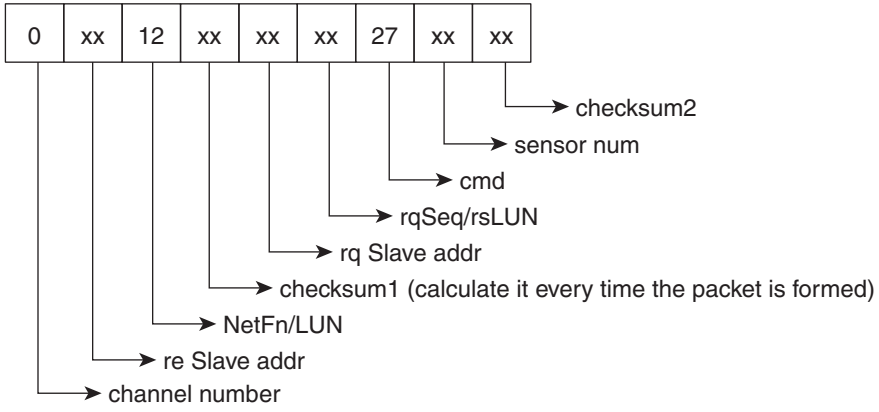


Note – In byte number 9, if the bit for a corresponding threshold is set to 1, then that threshold is set. If the bit is 0, the System Management Controller ignores that threshold. But if an attempt is made to set a threshold that is not supported, an error is returned in the command response.

Get Sensor Threshold

A typical example of the sensor command is as follows

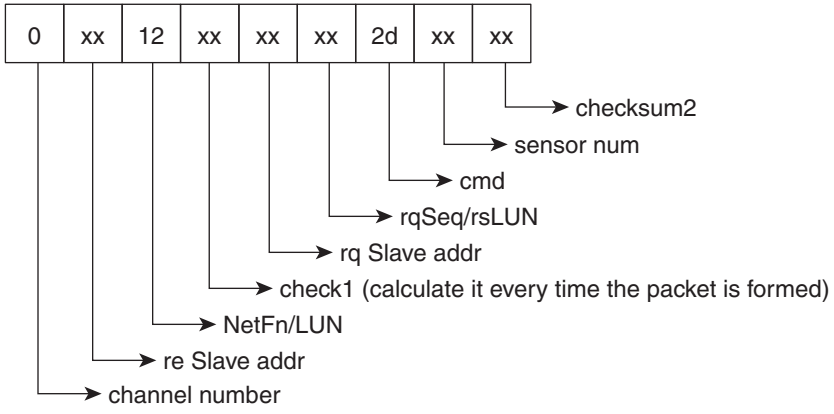
```
a5 2 27 12 20 34 12 ba 0 9 34 execute-smc-cmd
```



Get Sensor Reading

A typical example of the sensor command is as follows:

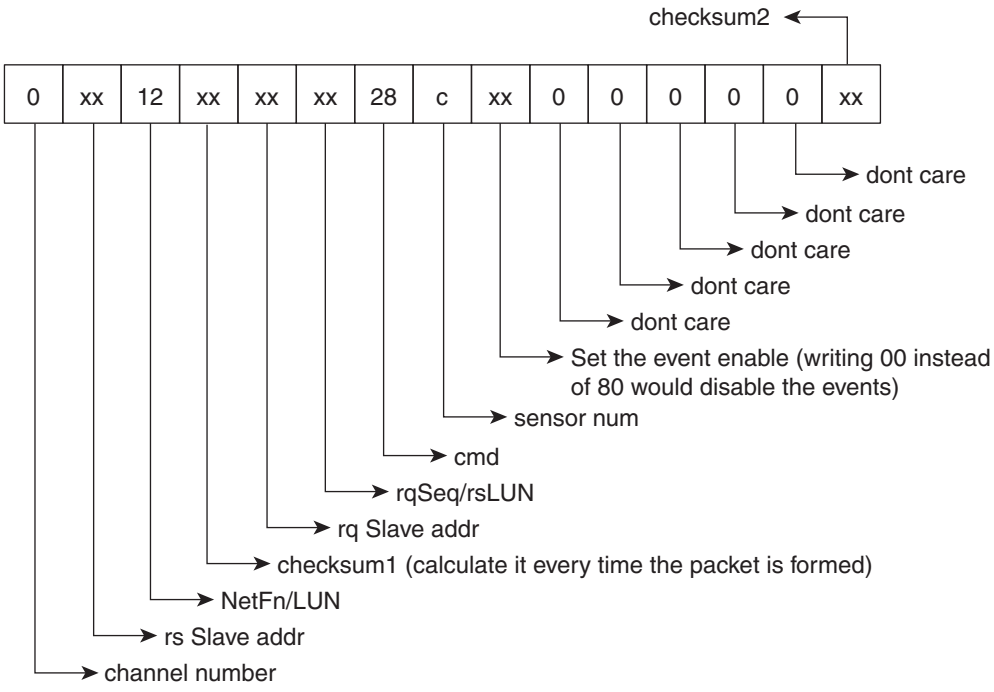
```
93 e 2d 12 20 34 12 ba 0 9 34 execute-smc-cmd
```



Set Sensor Event Enable Command

A typical example of the sensor command is as follows:

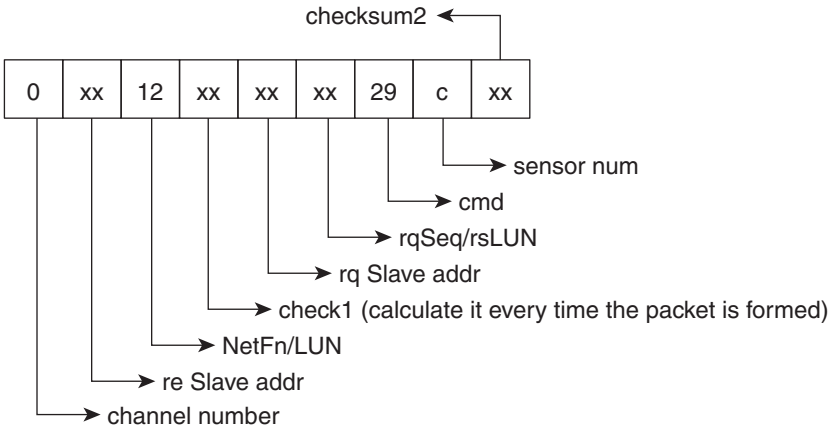
```
24 0 0 0 0 80 2 28 12 20 34 12 ba 0 e 34 execute-smc-cmd
```



Get Sensor Event Enable

A typical example of the sensor command is as follows:

```
a3 2 29 12 20 34 12 ba 0 9 34 execute-smc-cmd
```



Note – The NetFN/LUN for all sensor IPMI commands is 12, which implies that the netFn is 0x04 lun= 0x2.

ASM Application Programming

The following sections describe how to use the ASM functions in an application program.

For the ASM application program to monitor the hardware environment, the following conditions must be met:

- The system controller device driver must be installed.
- The ASM device driver must be present.
- The ASM application program must be installed and running.

The ASM parameter values in the application program apply when the system is running at the Solaris level and do not necessarily have to be the same as the corresponding to the parameter settings in the OpenBoot PROM.

To change the ASM parameter setting at the OpenBoot PROM level, see "OpenBoot PROM Environmental Parameters" on page 57 for the procedure. The OpenBoot PROM ASM parameter values only apply when the system is running at the OpenBoot PROM level.

Specifying the ASM Polling Rate

For most applications, an ASM polling rate of once every 60 seconds is adequate.

To specify a polling rate of every 60 seconds in an ASM application program, type the following at the command line for the Solaris operating environment:

```
do {

... /* read and process I2C bus devices data */

sleep (60); /* sets the ASM polling rate to every 60 seconds */

} while (1);
```

Monitoring the Temperature

The ASM application program monitors the CPU-vicinity temperature as follows (see "Sample Application Program" on page 71 for C code):

1. **Get the CPU-vicinity temperature measurements and other sensor measurements using the `ioctl` system call.**
2. **Examine the measurement readings and take the appropriate action.**

Note – The warning and shutdown temperatures are set for the CPU processor.

---

3. **Repeat the process for every ASM polling cycle.**

Solaris Driver Interface

The ASM driver is a STREAMS module that sits on top of the Solaris system controller driver. The Netra CP2000/CP2100 series ASM driver accepts STREAMS IOCTL input to the ASM driver, passes it onto the system controller driver as a command, and sends the sensor temperature as the output to the user. Currently, this driver handles only the local I^2C bus. On the Netra CP2000 series and the Netra CP2140 board, this driver enables the user to monitor the CPU-vicinity temperature, PMC temperature, memory module heat sink temperature, memory module temperature, SDRAM module1 temperature, SDRAM module2 temperature, and the

power module temperature. On the Netra CP2160 board, th driver enables the user to monitor the CPU temperature, the Inlet 1, Exhaust 1, Exhaust 2, SDRAM module 1 and the power module temperatures.

Note – The local I^2C bus is supported by the Solaris driver interface.

Interface Summary

Input Output Control with I_STR should be used to get sensor information. The data structure used to pass it as an argument for streams IOCTL is as follows.

CODE EXAMPLE 3-1 Input Output Control Data Structure

```
typedef struct stdasm_data_t {
       uchar_t busId;/* reserved */
       uchar_t sensorValue;/* return sensor Temperature */
       uchar_t scportNum; /* scport number for SC driver */
       uchar_t res1; /* Reserved */
       uchar_t res2; /* Reserved */
       uchar_t sensorNum; /* sensor Number */
       uchar_t res3; /* Reserved */
       uchar_t res4; /* Reserved */
} stdasm_data;

#define STDASM_CPU2/* CPU Temperature Sensor */
#define STDASM_INLET1/* Inlet1 Temperature Sensor */
#define STDASM_EXHAUST1/* Exhaust1 Temperature Sensor */
#define STDASM_EXHAUST2/* Exhaust 2 Temperature Sensor */
#define STDASM_SDRAM10xa/* SDRAM module 1 Temperature Sensor */
#define STDASM_SDRAM20xc/* SDRAM module 2 Temperature Sensor */
#define STDASM_POWER0xe/* Power Module Temperature Sensor */
```

When the monitoring is successful, it returns a `0`. For any error, it returns `-1` and the `errno` is set correspondingly. Trying to read any sensor which is not physically present sets `errno` as `ENXIO`. For any hardware or firmware failures, the `errno` is `EINVAL`. For any memory allocation problems, the `errno` is `EAGAIN`.

Sample Application Program

This section presents a sample ASM application that monitors the CPU-vicinity temperature. Please refer to `/usr/platform/sun4u/include/sys/stdasm.h` if you want to add support for the other six sensors in the application.

CODE EXAMPLE 3-2 Sample ASM Application Program

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
#include <sys/uadmin.h>
#include <stdasm.h>/* lives in /usr/platform/sun4u/include/sys directory */

/* Right now, this application monitors the CPU temperature only, if you want
       to add support for the other 6 sensors, you have to duplicate 12 lines
       in the ProcessAllTemps routine. Also refer the stdasm.h for sensorNum */

#define MaxTemperature 65

static void ProcessTemp(int CurrentTemp)
{
       FILE *WarnFile;
       printf(" %d C\n", CurrentTemp);
       if (CurrentTemp > MaxTemperature) {
           printf("WARNING!! Current Temperature <%d> exceeds MaxTemp <%d> \n",
CurrentTemp, MaxTemperature);
           WarnFile = fopen("WarnFile", "w");
           if (WarnFile) {
               fprintf(WarnFile, "WARNING!! Current Temperature <%d> exceeds
MaxTemp <%d> \n", CurrentTemp, MaxTemperature);
               system("wall -a *WarnFile");
               fclose(WarnFile);
               uadmin(A_SHUTDOWN, AD_HALT, 0);
           } else {
               printf("Creation of WarnFile failed\n");
               uadmin(A_SHUTDOWN, AD_HALT, 0);
               exit(4);
           }
       }
}
static void ProcessAllTemps(int AsmFd, int ScPort)
{
       int Result;
```

CODE EXAMPLE 3-2 Sample ASM Application Program *(Continued)*

```
        stdasm_data SAData;
        struct strioctl sioc;

        SAData.sensorNum = STDASM_CPU; /* Can be STDASM_PMC or any other */
        SAData.scportNum = ScPort;
        sioc.ic_cmd = STDASM_GETSENSOR; /* Ioctl flag for asm driver */
        sioc.ic_len = sizeof(stdasm_data);
        sioc.ic_dp = (char *)&(SAData);
        sioc.ic_timout = 200;
        do {
            system("date");
            printf("                                \n");
            printf("*****************************\n");
            printf("                                \n");

            /* Read the CPU Temperature */
            Result = ioctl(AsmFd, I_STR, &sioc);
            if (Result == -1) printf("ioctl RetValue %d\n", errno);  /* error cond
*/
            else printf("Temperature %d\n", SAData.sensorValue); /* Sensor Temp
*/
            ProcessTemp(SAData.sensorValue);

            /* Duplicate the above 12 lines for other 6 sensors STDASM_PMC,
                STDASM_MBHS, STDASM_MBMem, STDASM_SDRAM1, STDASM_SDRAM2,
                STDASM_POWER too */

            sleep(60);/* Recommended polling rate */
        } while(1);
}
int main(int argc, char *argv[])
{
        int AsmFd;
        int Result;
        struct strioctl sioc;
        int ScPort = 0;

        if ((AsmFd = open("/dev/sc", O_RDWR)) < 0) {  /* open the SC device */
            printf("Unable to open device /dev/sc; errno=%d\n", errno);
            exit(1);
        }
        /* Reserve the SC port for SC driver */
        sioc.ic_cmd = SCIOC_RESERVE_PORT;
        sioc.ic_len = sizeof(ScPort);
        sioc.ic_dp = (char *)&(ScPort);
        sioc.ic_timout = 200;
        Result = ioctl(AsmFd, I_STR, &sioc);
```

CODE EXAMPLE 3-2 Sample ASM Application Program *(Continued)*

```
        if (Result == -1) {
            printf("I_STR RetValue %d\n", errno);
            exit(2);
        } else printf("SC PORT is <%d>\n", ScPort);

        /* Push the 'ASM' driver module */
        Result = ioctl(AsmFd, I_PUSH, "stdasm");
        if (Result == -1) {
            printf("I_PUSH stdasm failed RetValue %d\n", errno);
            exit(3);
        }
        ProcessAllTemps(AsmFd, ScPort);
}
```

Note – The `stdasm.h` header file is located in the following directory: `/usr/platform/sun4u/include/sys`

Temperature Table Data

This section describes the test configuration used to generate the data used for the OpenBoot PROM temperature table in the ASM table temperature monitoring function. It should be used as a guideline by OEMs who need to revise the OpenBoot PROM temperature table because of changes to the enclosure, system, or fan configuration.

System Configuration and Test Equipment

The system configuration and test equipment used to obtain the ASM temperature data is as follows:

- Netra CP2000 or CP2100 series board with memory module
- Chassis: 5-slot CompactPCI chassis, 8-slot HA CompactPCI chassis, power supply, hard disk drive, floppy disk drive, and fan
- Environmental chamber
- Air Flow Measurement Tool
- Data Logger
- Two thermocouples

Thermocouple Locations

The two thermocouples are positioned as follows:

- The first thermocouple is attached at the base of a fin on the CPU heat sink in the center area of the heat sink so that it is directly above the CPU.
- The second thermocouple is placed near the bottom edge of the board to measure inlet temperature to the board. It is not positioned in direct air flow in order to read the true ambient temperature for the board.

▼ To Attach and Test Thermocouples

1. **Attach the thermocouples on the board.**

   See the section on "Thermocouple Locations" on page 74 above for further details.

2. **Install the board in the far left slot (slot #1) of the CompactPCI chassis**

   For location of thermocouple see FIGURE 3-2, FIGURE 3-3 and FIGURE 3-4 and FIGURE 3-5.

3. **Install a dummy 6U CompactPCI board in the next slot to control the air flow.**

   The front panels of the chassis should be filled.

4. **Set up the fan speed to maintain air flow of 320 linear feet per minute (LFM) or greater.**

   Air flow is measured by securing the air flow sensor approximately 5 mm from the side of CPU heat sink.

5. **Place the chassis inside the environmental chamber.**

6. **Set up the chamber temperature to cycle from 0ºC to 60ºC in 5ºC steps.**

7. **Run the SunVTS™ software during the test.**

8. **Read the thermocouple temperatures after at least one hour.**

   Wait at each temperature step.

CHAPTER 4

Programming the User LED

This chapter describes how to use the Alarm/User LED. The Alarm/User LED is located on the front panel of the Netra CP2100 series boards. The bi-colored LED is red and green in color (see FIGURE 4-1 for the location of the Alarm/User LED on the board front panel).

Note – Programming the User LED is supported on the Netra CP2140 and the Netra CP2160 boards when they are used with the *CP2000 Supplemental CD 4.0 for Solaris 8* only.

In order to use the LED function, support with a `sparc v9 64 bit C` library and the `led.h` file are required. The Application Programming Interface (API) for the user is documented in the `led.h` file. The library and the file are available on the *CP2000 Supplemental CD 4.0 for Solaris 8*.

FIGURE 4-1 Illustration of a Typical Netra CP2140 Board Front Panel Showing the Alarm/User LED

Files and Packages Required to Support the Alarm/User LED

To use the Alarm/User LED feature, the user should update the firmware with the appropriate firmware version that supports this feature on the Netra board.

Note – To check the current firmware version and for instructions on how to update the firmware, refer to the technical reference manual of the Netra board that you are using.

The list of packages that are required are as follows:

- `SUNWledl`: SPARC V9 64-bit C library `libcp2000.so` available at:

  `/usr/platform/${PLATFORM}/lib`

- `SUNWledu`: LED include file available at:

  `/usr/include/sys/`

Ensure that the following drivers are also there, as needed:

- `SUNWcph`: 32-bit sc driver and
- `SUNWcphx`: 64-bit sc driver available at:

  `/platform/${PLATFORM}/kernel/drv/sparcv9/sc`

- `SUNWled.u`: 32-bit LED driver and
- `SUNWledx.u`: 64-bit LED driver available at:

  `/platform/${PLATFORM}/kernel/strmod/sparcv9/s_led`

A typical example of ${PLATFORM} is `UltraSPARCengine_CP-60` for the Netra CP2160 board. An example for the library directory is:

```
/usr/platform/UltraSPARCengine_CP-60/lib
```

Applications

This section provides the application programming interface (API) to control the command combination of the Alarm/User LED, and instructions on how to compile and link the information.

Note – Since the LED interface installs and then removes the led_s streams module, an error can occur when multiple applications attempt to use this interface at the same time. If the user desires more than one application to use this interface, application software should incorporate a synchronization method such that only one access to the interface exists at any time.

CODE EXAMPLE 4-1 Application Programming Interface for the Netra CP2140 Board

```
extern        int led(int led, int cmd);

/*       leds */
#define       USER_LED_RED0x2
#define       USER_LED_GREEN0x4

/*       commands*/
#define       LED_OFF0x0
#define       LED_ON0x1
#define       LED_SQUAREWAVE0x2
#define       LED_HEARTBEAT0x3
```

CODE EXAMPLE 4-2 Application Programming Interface for the Netra CP2160 Board

```
extern        int led(int led, int cmd);

/*       leds */
#define       USER_LED_RED0x2
#define       USER_LED_GREEN0x4

/*       commands*/
#define       LED_OFF0x0
#define       LED_ON0x1
```

The supported LED and command combinations are shown in TABLE 4-1 and TABLE 4-2.

TABLE 4-1 Supported LED and Command Combinations for the Netra CP2140 Board

Color of LED*	LED_OFF	LED_ON	LED_SQUAREWAVE	LED_HEARTBEAT
USER_LED_RED	Yes	Yes	No	No
USER_LED_GREEN	Yes	Yes	Yes	Yes

* When the user turns on the red and green LED at the same time, the light shows as amber. There is no support for a red LED blinking light.

TABLE 4-2 Supported LED and Command Combinations for the Netra CP2160 Board

Color of LED*	LED_OFF	LED_ON	LED_SQUAREWAVE	LED_HEARTBEAT
USER_LED_RED†	Yes	Yes	No	No
USER_LED_GREEN	Yes	Yes	No	No

* When the user turns on the red LED, the green LED goes out and when the user turns out the green LED, the red LED goes out. When the user turns off the red LED, only the red LED turns off, and when the user turns off the green LED, only the green LED turns off..

† The Netra CP2160 board has a green and amber light, rather than a green and red light. In the software code, however, the amber light is represented by USER_LED_RED.

Compile

As you compile your application, you need to use the compiler command (cc) flag **-I**, to include the `sys/led.h` file named in "Files and Packages Required to Support the Alarm/User LED" on page 77. Specify 64-bit binaries by setting the -xarch=v9 and -D__sparcv9 compiler flags.

For example:

```
-xCC -xarch=v9 -D__sparcv9 -I/usr/platform/
sun4u/include/
```

Note – Type the above command all on one line.

Link

To create a link to the library named (`libcp2000.so`) listed in "Files and Packages Required to Support the Alarm/User LED" on page 77, use the linker flag **-L** command.

For example:

```
-L /usr/platform/UltraSPARCengine_CP-60/lib
```

CHAPTER 5

Programming Netra CP2100 Series Board Controlled Devices

This chapter describes, for developers, how to create applications that can identify and control hardware devices connected to Netra CP2100 series board-controlled systems.

Note – These applications are supported on the Netra CP2140 and Netra CP2160 boards when they are used with the *CP2000 Supplemental CD 4.0 for Solaris 8* only.

This document contains the following sections:

Overview of Hot-Swap Device States

The Netra CP2100 series hot-swap software can display the various hot-swap states for a CompactPCI device connected to the system. A device that has been installed and connected to a system's slot can have one of the following states:

- Configured – The device has been powered on in a slot and its hardware resources are available to the operating system.
- Unconfigured – The device's resources are not available to the operating system. The device can safely be removed from the system.

- Unknown – The device has been powered on in a slot and connected to the system, but the system has not attempted to configure the device.
- Failed – The device has failed an attempt to be unconfigured from a slot. The resources from the device remain available to the operating system and the Solaris software `cfgadm`(1M) command reports that the device is still in the configured state.

Use the `cfgadm` hot-swap command to verify the state of a device. Note that a configured device remains in the configured state until it has been successfully unconfigured.

Retrieving Device Type Information

Using a pseudo device, an `ioctl()`, and `libdevinfo` library interfaces, you can retrieve the device type information (for example, the vendor IDs and the driver names) for every configured CompactPCI card in a system. With this information, you can deduce the type of CompactPCI card configured in each system slot.

Using `cphsc` to Collect Information

The CompactPCI hot-swap controller pseudo device driver, `cphsc`, maintains the new device state information for all slots in a system. You can access this device state information by using an `ioctl()` on an instance of the `cphsc` pseudo device. The `cphsc` device returns a table containing an entry for each slot within the system's chassis. Each entry contains the new device state, the slot state, the `cpci` device number, and the logical slot number. Access permission for the `/dev/cphsc` device is read-write (`rw`) for superuser only.

The cphsc device driver exports an hsc_slotinfo_t element that has the following structure:

```
typedef struct hsc_slotinfo {
       hsc_slot_state_t hsc_slot_state;
       hsc_dev_state_t hsc_dev_state;
       uint16_t hsc_devnum;
       uint16_t hsc_slotnum;
} hsc_slotinfo_t;

typedef enum {HSC_SLOT_EMPTY, HSC_SLOT_DISCONNECTED,
       HSC_SLOT_CONNECTED, HSC_SLOT_UNKNOWN} hsc_slot_state_t;

typedef enum {HSC_DEV_CONFIG, HSC_DEV_UNCONFIG,
       HSC_DEV_UNCONFIG_FAILED, HSC_DEV_UNKNOWN} hsc_dev_state_t;
```

Where the state of a chassis's CompactPCI slot (hsc_slot_state_t) can be:

- HSC_SLOT_EMPTY – The slot is empty.
- HSC_SLOT_DISCONNECTED – A card occupies the slot, but the slot is not connected to the system. The hot-swap software must connect the slot to the system before the card's software resources can be configured to the system.
- HSC_SLOT_CONNECTED – A card occupies the slot and the slot's resources are available to the system.
- HSC_SLOT_UNKNOWN –The slot may or may not be occupied. The system cannot derive the state of the slot.

Note – See "Overview of Hot-Swap Device States" on page 81 for a description of each hot-swap device state (hsc_dev_state_t).

HSIOC_GET_INFO ioctl()

A single HSIOC_GET_INFO ioctl() returns the entire table of hsc_slotinfo_t structures. The structure is defined as 64-bit aligned. Constraining the structure to align to the larger of the two data models enables the structure to have the same format in either a 32-bit or a 64-bit application.

Creating a Header File for the HSIOC_GET_INFO ioctl()

To make full use of the HSIOC_GET_INFO ioctl(), create a header file containing the required preprocessing directives and macros (see CODE EXAMPLE 5-1). After creating the header file, include the file in any application that uses the ioctl().

CODE EXAMPLE 5-1 HSIOC_GET_INFO ioctl() Header File

```
/*
 * HSIOC_GET_INFO ioctl() Header File
 */

/*
 * Argument to HSIOC_GET_INFO ioctl()
 * Define struct to be 64-bit aligned
 */
typedef struct hsc_gi_arg {
        union hsc_gi_tbl {
                hsc_slotinfo_t *tbl;
                uint64_t tbl64;
        } hsc_gi_tbl_u;
        union hsc_gi_tblsize {
                int *tblsize;
                uint64_t tblsize64;
        } hsc_gi_tblsize_u;
} hsc_gi_arg_t;

/*
 * Binary definition of the HSIOC_GET_INFO ioctl()
 */
#define HSIOC_GET_INFO (('h' << 8) | 1)

#define hs_tbl hsc_gi_tbl_u.tbl
#define hs_tbl64 hsc_gi_tbl_u.tbl64
#define hs_tblsize hsc_gi_tblsize_u.tblsize
#define hs_tblsize64 hsc_gi_tblsize_u.tblsize64
```

Note – The hsc_gi_tbl_u.tbl and hsc_gi_tblsize_u.tblsize entries can only be used in 64-bit applications. If you are developing a 32-bit application, use the hsc_gi_tbl_u.tbl64 and the hsc_gi_tblsize_u.tblsize64 entries, which work for either 32-bit or 64-bit applications.

Using the HSIOC_GET_INFO ioctl()

This ioctl() has one argument, arg, which is a pointer to an hsc_gi_arg_t structure. The first field of hsc_gi_arg_t is a pointer to user-allocated storage; the second field is a pointer to the size of this storage in bytes.

```
hsc_gi_arg_t arg;

ioctl(fd, HSIOC_GET_INFO, &arg)
```

To find out how much user storage to allocate for the slot table, call the ioctl() with the first field of hsc_gi_arg_t set to NULL and the second argument defined as a pointer to an integer. Since the tbl field is NULL, the ioctl() returns only the number of elements in the table in the tblsize field.

When both fields have non-NULL values, the slot table is copied into the user-allocated storage if tblsize is large enough to contain the table. The returned table is an array of hsc_slotinfo_t structures where the size of the table is:

```
hsc_gi_arg_t arg;
int n;
int tblsize;
hsc_slotinfo_t *tbl;

arg.hs_tbl64 = (uint64_t)tbl;
arg.hs_tblsize64 = (uint64_t)&n;

tblsize = n * sizeof (hsc_slotinfo_t);
```

If tblsize is too small, the ioctl() returns an EINVAL error. If either the tbl field or the tblsize field contains invalid pointers, the ioctl() returns an EFAULT error.

Note – In a failed device state—for instance, where a device has failed to be unconfigured—the cfgadm command shows that the device remains configured to the system and the HSIOC_GET_INFO ioctl() reports that the device is in the HSC_DEV_UNCONFIG_FAILED state. In this device state, the card that an operator attempted to extract remains plumbed and connected to the system's resources. When the condition preventing the device from being unconfigured is removed, the operator can unconfigure the device using the cfgadm command. A Reconfiguration Coordination Manager (RCM) script can also be written to automatically shut down applications using the device so that unconfiguration is successful. See Chapter 6 for infomation on RCM.

In CODE EXAMPLE 5-2, a pseudo function finds the device type information for a given slot number (hsc_slotnum) and device number (hsc_devnum). The paragraphs that follow CODE EXAMPLE 5-2 describe how you might implement this function in your application.

CODE EXAMPLE 5-2 Using cphsc to Find Device Type Information

```
int n;
int tblsize = 0;
hsc_slotinfo_t *tbl;
hsc_gi_arg_t arg;

if ((fd = open("/dev/cphsc")) < 0)
       return (FAILURE);

arg.hs_tbl64 = NULL;
arg.hs_tblsize64 = (uint64_t)&n;

/* get the number of entries in the slotinfo table */
if (ioctl(fd, HSIOC_GET_INFO, &arg) < 0)
       return (FAILURE);

tblsize = n * sizeof (hsc_slotinfo_t);
tbl = (hsc_slotinfo_t *)malloc(tblsize);
arg.tbl64 = (uint64_t)tbl;

/* get the slotinfo table */
if (ioctl(fd, HSIOC_GET_INFO, &arg) < 0)
       return (FAILURE);

tp = tbl;

for (i = 0; i < n; i++, tp++) {

       if (tp->hsc_dev_state == HSC_DEV_UNCONFIG_FAILED)
              if ((dip = find_device_info(tp->hsc_slotnum,
                  tp->hsc_devnum)) != NULL) {
                         unplumb_device(dip);
                         unconfig_device_again(dip);
                  }
}
```

The cphsc pseudo driver registers the attachment points (slots) with the hot-swap framework when the user-level system controller daemon, sctrld, activates it. The sctrld daemon plumbs the cphsc driver to the system management controller driver.

The attachment points (slots) are represented as minor nodes of the CompactPCI bus node. The minor number of these nodes corresponds to the PCI device number. This is not a true device tree node because it represents a receptacle rather than the occupant itself. When a device is configured into the slot, a device node is created that represents the configured device. This device node is a child of the system board controller's (SBC) CompactPCI root nexus, and its PCI bus address corresponds to the PCI device number of the corresponding slot.

To search the SBC's device tree for slots with *configured* devices, use the interfaces provided by the `libdevinfo` library.

When an attachment point represents a multifunction device, device nodes are created for each of these functions. In the SBC's device tree, these device nodes appear as children of the attachment point device node. Use interfaces provided by `libdevinfo` to get the following information for each function (for example, for each child node):

- Vendor ID
- Subsystem vendor ID (if present)
- Device ID
- Subsystem ID (if present)
- Driver name and instance number

Using the above information, you can find out what type of device was configured in a given slot.

Note – For satellite CPU boards, the above information must be obtained from the attachment point device node itself, because the device tree ends at that node.

Using Library Interfaces to Collect Information

Use the `libdevinfo` library interfaces to derive the device type information (vendor ID, device ID, subsystem ID, and so on) based on a CompactPCI card's device number. The following example procedure shows how you can implement the `find_device_info()` routine shown in CODE EXAMPLE 5-2 using the `libdevinfo` application programmer's interface.

Note – You can use the `libdevinfo` library interfaces to find information only on devices that are in a configured state. If a device is in an unconfigured state, you cannot retrieve this information.You can use the `libdevinfo` library interfaces to find information only on devices that are in a configured state. If a device is in an unconfigured state, you cannot retrieve this information.

1. **Call the `di_init()` function to retrieve a snapshot of the kernel device tree starting from the top (root node denoted by "`/`").**

   For example:

```
root = di_init("/", DINFOSUBTREE | DINFOPROP | DINFOMINOR)
```

2. **Call the `di_walk_minor()` function, starting from the root node, to find the attachment point node with a minor node whose minor number is equal to the PCI device number under question.**

   For example:

```
di_walk_minor(root, DDI_NT_PCI_ATTACHMENT_POINT, 0,
              (void *)&pci_device_num, ....)
```

   The attachment point node returned in this step is the CompactPCI root nexus node.

3. **Look for a child node (immediate child) using `libdevinfo` function calls such as `di_child_node()` and `di_sibling_node()` to find the desired attachment point node.**

   This attachment point node has a bus address equal to the PCI device number under question. Use the `libdevinfo` function `di_bus_addr()` to get the bus address.

4. **To find the properties of the child nodes of the attachment point node, use the following `libdevinfo` function calls:**

   - `di_driver_name()` – To find the driver name
   - `di_instance()` – To find the instance number

   Since the properties of interest (*device-id*, *subsystem-id*, *vendor-id*, *subsystem-vendor-id*) are all of type `DDI_PROP_TYPE_INT` (32-bit integer), you can use the `di_prop_lookup_ints()` and `di_prom_prop_lookup_ints()` functions to find the property values.

   To get the PROM properties using `di_prom_lookup_ints()`, call the `di_prom_init()` and `di_prom_fini()` functions before accessing the PROM property list, as the list is not included in the snapshot returned by the `di_init()` function.

5. **Call the `di_fini()` function to destroy the snapshot of the kernel device tree and free up memory.**

High Availability Signal Support

Using high availability (HA) signal support, you can write your application to control which CompactPCI slots in an HA chassis are powered on. Before your application can control these signals, you must first set two Netra CP2140 series board OpenBoot™ PROM configuration variables.

Note – The following section applies only to CP2140 boards used as system controllers.

Setting OpenBoot PROM Configuration Variables

You must set these two Netra CP2140 series board OpenBoot PROM configuration variables before an application can gain control of the HA signals:

- `ha-signal-handler`
- `poweron-vector`

The `ha-signal-handler` variable indicates whether the system management controller (SMC) module controls the HA signals or if an external application controls these signals. By default, the variable's value is `0`, which means that the SMC module controls the HA signals. If you set the variable to `1`, an external application can control the HA signals.

If you are developing an application that must control the HA signals, set the `ha-signal-handler` variable to `1` using the `setsmcenv` OpenBoot PROM command:

```
ok setsmcenv ha-signal-handler 1
```

To confirm that the variable has been set correctly, use the `printsmcenv` command at the `ok` prompt:

```
ok printsmcenv ha-signal-handler
1
```

Note – To display the setting for every configuration variable, type the `printsmcenv` command at the `ok` prompt without a variable.

The poweron-vector configuration variable is an 8-bit bit-vector that indicates which CompactPCI slots will be powered on during the chassis power-on. By setting this variable, you set which slots in the chassis will be powered on.

Note – You can use the poweron-vector variable only when the ha-signal-handler variable is set to 1.

TABLE 5-1 defines the poweron-vector variable's 8-bit vector.

TABLE 5-1 poweron-vector Variable Bit Definition and Power Setting

Bit	Slot	Power Setting
7	Not in use	
6	8	0 is on, 1 is off
5	7	0 is on, 1 is off
4	6	0 is on, 1 is off
3	5	0 is on, 1 is off
2	4	0 is on, 1 is off
1	3	0 is on, 1 is off
0	2	Not supported since the SBC can occupy this slot

If you are developing an application that controls the power of every slot in the chassis, you must shut off the power-on bit-vector for every slot. At the ok prompt, use the setsmcenv command to set the poweron-vector variable to 0xff:

```
ok setsmcenv poweron-vector ff
```

Note – It is not possible to control the powering of the Netra CP2140 board system controller slots. They will always be powered on by the SMC module irrespective of the value of ha-signal-handler.

Controlling and Monitoring High Availability Signals

Each slot has BDSEL#, HEALTHY#, and RESET# signals. These HA signals can be controlled and monitored by an application. An extra flag, called slot_flg, is also available that indicates whether a slot is empty or full.

An 8-bit bit-vector represents all the slots within an 8-slot HA CompactPCI chassis. The SBC and the standby SBC (SSBC) are reserved slots in an HA chassis and are not counted in the 8-bit bit-vector. See TABLE 5-1 for the bit number to slot number sequencing.

Use the `bdsel, healthy` and `reset` bit-vectors to control a slot's BDSEL#, HEALTHY#, and RESET# signals. When using these bit-vectors, any set bit means that the signal is de-asserted on that slot. Conversely, any clear bit means that the signal is asserted. For example, a slot that is not healthy will have its bit set, which means that the HEALTHY# signal is de-asserted. When a slot has its RESET# signal asserted, the appropriate bit for this slot is cleared, and the board will not be visible on the CompactPCI bus until this RESET# signal is de-asserted or has its bit set.

The `bdsel` bit-vector has two meanings depending on whether the bit-vector is being written to or read. When the `bdsel` bit-vector is written, the value of the bit-vector should set the bits for slots that should have BDSEL# disabled, and cleared for slots that have BDSEL# asserted. The value of the `bdsel` bit-vector that is read has set bits for slots that are requesting BDSEL# to be asserted. For more information about this process, see "Bringing a Slot Online" on page 92.

The `slot_flg` bit-vector has two values: `0` (for an empty slot) or `1` (for an occupied slot).

The following data structure encapsulates all of the above bit-vectors:

```
typedef struct hsc_ha_sigstate {
       uchar_t bdsel;                /* bdsel bit-vector */
       uchar_t healthy;              /* healthy bit-vector */
       uchar_t reset;                /* reset bit-vector */
       uchar_t slot_flg;             /* empty/full bit-vector */
} hsc_ha_sigstate_t;
```

The `cphsc` pseudo driver exports an `ioctl()` that retrieves an `hsc_ha_sigstate_t` data structure and an `ioctl()` that sets the BDSEL# and RESET# HA signals.

To get the HA signal state, use the `HSIOC_GETHASIG ioctl()` as follows:

```
hsc_ha_sigstate_t arg;

ioctl(cphscfd, HSIOC_GETHASIG, &arg);
```

To set the HA signal state, use the HSIOC_SETHASIG ioctl() as follows:

```
hsc_ha_sigstate_t arg;

ioctl(cphscfd, HSIOC_SETHASIG, &arg);
```

Note – Only the bdsel and reset fields in the hsc_ha_sigstate_t structure are writable. The other fields are ignored.

Bringing a Slot Online

Note – The following section applies only to CP2140 boards used as system controllers.

Only slots with slot_flg and bdsel bits set are in a state to be powered on. This state means that the slot is FULL and is requesting a BDSEL# signal.

For example, if the HSIOC_GETHASIG ioctl() returns an hsc_ha_signal_t struct, arg, with the value shown below, slot 5's slot_flg and bdsel bits are set.

```
arg.bdsel    ==  0x8
arg.healthy  ==  0xff
arg.reset    ==  0
arg.slot_flg ==  0x8
```

To power on slot 5, your application should assert the BDSEL# signal on this slot. You can calculate the new value for the bdsel bit-vector by clearing the slot 5 bit from the healthy bit-vector and storing the result in arg.bdsel as follows:

```
arg.bdsel = arg.healthy & ~(1 << (slot - 2));
```

Note – The value (slot - 2) is the bit position in the bdsel bit-vector for a given slot. For example, if the slot number is 5, the corresponding bit is 3 or (5 - 2).

The value in the reset bit-vector should remain unchanged and be set to the value returned by the HSIOC_GETHASIG ioctl().

The healthy bit-vector and slot_flg flag fields are read-only so the HSIOC_SETHASIG ioctl() ignores them.

Slot 5's BDSEL# signal is asserted after the HSIOC_SETHASIG ioctl() is executed, as follows:

```
arg.bdsel   = 0xf7
arg.healthy = 0xff
arg.reset   = 0

ioctl(cphscfd, HSIOC_SETHASIG, &arg)
```

Use the HSIOC_GETHASIG ioctl() to poll the HA signals until slot 5 asserts a HEALTHY# signal. The condition to wait for is:

```
(arg.healthy & (1 << (slot -2))) == 0
```

When the healthy bit is clear, the board in this slot is powered on and is ready to be taken out of reset.

The HSIOC_SETHASIG ioctl() sets both the bdsel and the reset bit-vectors. For slots that are to remain powered off, their bdsel bit should be set to 1. In this example, slot 5 is the only slot powered on, so the bit-vector should be:

```
arg.bdsel = 0xf7
```

To de-assert reset for slot 5, set the reset bit-vector to:

```
arg.reset = 0x8
```

Executing the HSIOC_SETHASIG ioctl() with the following arg takes the board out of reset and causes the board to generate an ENUM# signal if it has hot-swap friendly silicon:

```
ioctl(cphscfd, HSIOC_SETHASIG, &arg)
```

The Solaris hot-swap framework recognizes this ENUM# signal, which in this case indicates that the board has been freshly inserted, and then configures the board. Use the cfgadm command at a Solaris command prompt to verify that this slot has been successfully configured.

After a board is successfully removed from the chassis, the slot containing this board should have its BDSEL# and RESET# signals set to 1 so that future hot-swap insertions will be detected within this slot.

TABLE 5-2 Hot-Swap HA Signal States for a Single CompactPCI Slot

BDSEL	HEALTHY	RESET	slot_flg	Description	System Board Controller (SBC) Actions
0	1	0	0	Slot empty	Poll for the bdsel and slot_flg bits being set to 1.
1	1	0	1	Slot full	Use the HSIOC_SETHASIG ioctl() to assert bdsel. Note that the bdsel bit-vector should contain a 1 wherever a slot should remain powered off.
0	1	0	1	Slot powered	Poll for the board's HEALTHY# signal to be asserted.
0	0	0	1	Board healthy	Take board out of reset. Use the HSIOC_SETHASIG ioctl().
0	0	1	1	Board powered on CompactPCI bus	The board has successfully powered up and is visible. An ENUM# signal is generated.
0	1	1	1	Board unhealthy	Board is reporting problems, and should be taken off the bus.

Note – All other HA signal combinations indicate errors.

Using the HSIOC_SETHASIG ioctl()

Note – The following section applies only to CP2140 boards used as system controllers.

The HSIOC_SETHASIG ioctl() simultaneously sets both the bdsel and reset bit-vectors. The bdsel bit-vector should have the bits set for the slots that are not requesting the BDSEL# signal. This bit setting is the complement of the bit pattern returned from the HSIOC_GETHASIG ioctl(). The value returned from the HSIOC_GETHASIG ioctl() indicates which slots have requested the BDSEL# signal.

For example, if the bdsel bit-vector is 0xa8, the chassis slots 5 and 7 are requesting BDSEL# signals (bit 7 is ignored so this is equivalent to a setting of 0x28). Also, the slot configuration flag should be set to 0x28, which indicates that slots 5 and 7 have

boards installed in them. If both slots 5 and 7 assert the BDSEL# signal, the value for the `bdsel` bit-vector (used with the HSIOC_SETHASIG `ioctl()`) should be the complement of 0xa8 (or 0x57).

You can also use the `healthy` bit-vector as a guide to setting the `bdsel` bit-vector. Any bit set in the `healthy` bit-vector indicates that the HEALTHY# signal is de-asserted, and that the board should not be powered on unless the board was already powered on and then became unhealthy for some reason. To assert the BDSEL# signal on any slot, the bit corresponding to the slot requesting the BDSEL# signal can be cleared in the `healthy` bit-vector as follows:

```
hsc_ha_signal_t arg;

ioctl(cphscfd, HSIOC_GETHASIG, &arg); /* get current values */

arg.bdsel = arg.healthy & ~(0x28); /* clear bits for slot 5 and 7 */

ioctl(cphscfd, HSIOC_GETHASIG, &arg);
```

To enable the board configuration, the RESET# signal should be de-asserted when a board is asserting the HEALTHY# signal.

Add the following variables to your system's `/etc/system` file to generate an ENUM# signal when a board de-asserts a RESET# signal:

```
cphsc:hsc_do_enum=1
cphsc:hsc_enum_reactivate=1
```

Note – The `pkgadd` utility should have already added these lines to the `/etc/system` file when you installed the hot-swap software packages.

You must reboot your system before these settings affect the operating system.

Since the `HSIOC_SETHASIG ioctl()` sets both the `bdsel` and `reset` bit-vectors, the correct value for the `bdsel` bit-vector has to be constructed using the `healthy` bit-vector. The `healthy` bit-vector indicates which slots have the BDSEL# signal asserted.

For example, to de-assert the RESET# signal for slots 5 and 7, you would do the following:

```
hsc_ha_signal_t arg;

ioctl(cphscfd, HSIOC_GETHASIG, &arg); /* get current values */

arg.bdsel = arg.healthy; /* BDSEL set to HEALTHY */
arg.reset |= 0x28; /* de-assert reset for slot 5 and 7 */

ioctl(cphscfd, HSIOC_SETHASIG, &arg);
```

Please note that the preceding code section is only an example of how the `HSIOC_SETHASIG ioctl()` can be used to bring some slots out of reset and allow the occupants to get configured.

The preceding example assumes that your application controls the powering on of all of the slots. However, if you do not want to control the powering on and off of some of the slots and would prefer to delegate this control to the system (by appropriately setting the OpenBoot PROM variable, poweron-vector), you should only derive the `bdsel` mask from the healthy status of the slots under your application's control. Taking this action prevents accidentally powering off slots not under your application's control.

Note – Power off a slot only after the board has been unconfigured and the RESET# has been asserted. Do not attempt to control a slot's power while there is a board in the configured state and there is a driver attached as it could lead to a system panic or hang.

Creating a Header File for the CP2100 Series Software

CODE EXAMPLE 5-3 displays a header file containing all of the directives, macros, and definitions listed throughout this document. You must use a header file like the one below in order to develop applications using the software described in this document.

Note – The following `ioctl()` commands: `HSIOC_GETHASIG` and `HSIOC_SETHASIG` only apply to Netra CP2140 boards as system controllers. These commands cannot be used for Netra CP2160 boards.

CODE EXAMPLE 5-3 Netra CP2100 Series Software Header File

```
/*
 * Binary definition of the HSIOC_* ioctl()s
 */

#define _HSIOC                  ('h' << 8)

#define HSIOC_GET_INFO          (_HSIOC | 1)    /* get hsc_slot_t table */
#define HSIOC_GETHASIG          (_HSIOC | 8)    /* get ha signal state */
#define HSIOC_SETHASIG          (_HSIOC | 9)    /* set ha signal state */

typedef enum { HSC_DEV_CONFIG, HSC_DEV_UNCONFIG, HSC_DEV_UNCONFIG_FAILED,
        HSC_DEV_CONFIG_FAILED, HSC_DEV_UNKNOWN } hsc_dev_state_t;

typedef enum { HSC_SLOT_EMPTY, HSC_SLOT_DISCONNECTED, HSC_SLOT_CONNECTED,
        HSC_SLOT_UNKNOWN } hsc_slot_state_t;

typedef struct hsc_slotinfo {
        hsc_slot_state_t hsc_slot_state;
        hsc_dev_state_t hsc_dev_state;
        uint16_t hsc_devnum;
        uint16_t hsc_slotnum;
} hsc_slotinfo_t;
/*
```

CHAPTER 6

Reconfiguration Coordination Manager

This document describes how you can use Reconfiguration Coordination Manager scripts to automate certain dynamic reconfiguration processes when a Netra CP2000/CP2100 series board is used as a system controller.

Note – The Reconfiguration Coordination Manager scripts are supported on the Netra CP2140 and Netra CP2160 boards when they are used with the *CP2000 Supplemental CD 4.0 for Solaris 8* only.

This chapter contains the following sections:

Note that the RCM framework can only be used in systems with the Netra CP2100 series CompactPCI boards as system controllers with full hot-swap support.

Reconfiguration Coordination Manager (RCM) Overview

Beginning with the Solaris 8 2/02 release, the Solaris operating environment provides a Reconfiguration Coordination Manager (RCM) scripting interface that enables you to create scripts that can shut down applications and release system resources from hardware devices during dynamic reconfiguration (DR) operations. For example, you can create an RCM script to release network interfaces of a network interface card prior to unconfiguring the card from the system.

Prior to the RCM software, operators had to release system resources manually before the operating system could dynamically remove hardware devices (for example, network interface cards and hard drives) from the system. Manually releasing these devices often left system applications and devices in unknown states, which would force the operators to shut down the system before they could remove the devices.

You can now write and install RCM scripts that can better control this dynamic reconfiguration process. When responding to reconfiguration requests, the Solaris RCM daemon will launch the RCM scripts at the appropriate time to allow for the orderly removal of system resources. With the system resources released, a hardware device can be successfully unconfigured during dynamic reconfiguration operations.

For instructions on how to write and install RCM scripts, and for a description of all of the RCM commands, refer to the "Reconfiguration Coordination Manager (RCM) Scripts" section of the *Solaris 8 System Administration Supplement* (806-7502-*xx*), which is part of the *Solaris 8 2/02 Update Collection*. You can view this document online on the `http://docs.sun.com` website. Refer to the `rcmscript`(4) man page for additional information about creating and installing RCM scripts.

Using RCM with the Netra CP2100 Series CompactPCI Board

The RCM framework is fully integrated into the *CP2000 Supplemental CD 4.0 for Solaris 8* software and can be used in systems with the Netra CP2100 series CompactPCI boards as system controllers (with full hot-swap support).

You can write RCM scripts to shut down applications running on peripheral CompactPCI cards installed into the system backplane. The RCM daemon will execute these scripts when an operator attempts to unconfigure a card using the `cfgadm` command or by opening the CompactPCI board's ejection levers. For more information about the `cfgadm` command, refer to the `cfgadm`(1M) man page and the Solaris documentation.

The Solaris RCM software includes RCM modules that interact with dynamic reconfigurable devices. These RCM modules are shared object files, shipped with the Solaris software, that use the RCM application programming interface (API) to interact with the RCM framework. Both RCM modules and user-created RCM scripts can act as RCM clients during dynamic reconfiguration processes.

RCM scripts contain RCM commands that the Solaris DR framework will use to perform its operations. These RCM commands include `register`, which is used to specify the devices the script will manage, and `preremove`, which is used to remove resources from devices. For a full list of RCM commands, refer to the Solaris RCM documentation.

When an operator starts to unconfigure a card, the Solaris DR framework will call the RCM script's `preremove` function to do the necessary quiescing prior to proceeding with the actual unconfiguration. However, you can write an RCM script that can deny extraction requests, so when an operator tries to unconfigure a card, the Solaris DR framework will not attempt to unconfigure the device. In this case, the device will remain in a configured state and the card's blue extraction LED will stay unlit.

In the event that a device does not have associated RCM clients registered to receive device extraction notifications, the Solaris DR framework will automatically proceed with the unconfigure operation. The unconfigure operation will fail if the device is still in use. If the device is not in use, the unconfigure operation will succeed and the blue extraction LED will turn on.

If you are using a Netra CP2100 series system controller in a system that is set to basic hot-swap mode, use the `cfgadm` command with the `-f` option to prevent any possible interference from other RCM clients that have been registered for removal notifications of the same device. See "Avoiding Error Messages When Extracting Devices in Basic Hot-Swap Mode" on page 107 for more information.

Note – The RCM functionality that enables you to write scripts to shut down applications operating on *dumb* I/O cards (like network interface cards) is not available on satellite CPU boards. You will need to shut down applications, including the Solaris operating environment, running on satellite CPU boards manually before dynamically removing these boards.

Using RCM to Work With the Intel 21554 Bridge Chip

CompactPCI cards containing the Intel 21554 PCI-PCI bridge chip falsely turn on the blue extraction LED when they are dynamically removed from a chassis using the Netra CP2100 series board as a system controller. This blue extraction LED, located on a CompactPCI card's front panel, incorrectly turns on when the ejection levers are opened for extraction while the card's devices (for example, the card's network interfaces) are still in use. The Intel 21554 bridge chip logic that clears the extraction bit during a dynamic removal operation will also turn on the card's blue extraction LED in error.

The Solaris DR framework must clear the extraction bit at the end of a dynamic removal operation, whether or not it is successful, to indicate that the ENUM interrupt has been handled. The Solaris DR framework will issue the command to turn on the blue extraction LED only when the card's unconfiguration operation is successful. Because the Intel 21554 bridge chip turns on the blue LED even when the unconfiguration fails, an operator may falsely assume that the card can be safely removed even though the card is actually still in use by the system.

With the addition of RCM script support, however, you can create applications that use the RCM framework to either approve or deny dynamic removal operations before the Solaris dynamic reconfiguration software proceeds with the card's unconfiguration. Using an RCM script, your application can deny the removal of the card if it currently cannot be freed, possibly because it is engaged in a critical task at the time. Since the RCM script denies the operation prior to starting the unconfiguration process, the blue extraction LED will not turn on.

If, on the other hand, your application approves the removal request and uses the RCM script to quiesce the card, the card's unconfiguration will succeed and the Solaris DR framework will turn on blue extraction LED correctly.

Note – If your application approves the dynamic removal operation, the RCM script should also shut down all applications using the card's devices. Otherwise, the blue extraction LED may come on in error if the unconfiguration fails because the device is still in use.

By providing a mechanism for either denying a dynamic reconfiguration request or quiescing the devices, the RCM framework provides a workaround to the incorrect lighting of the blue extraction LED.

RCM Script Example

CODE EXAMPLE 6-1 shows an example RCM Perl script designed to shut down applications running on a network interface of a Sun Dual FastEthernet/SCSI 6U CompactPCI Adapter with PMC. This CompactPCI card contains two network (`qfe`) interfaces and two SCSI interfaces.

The example script uses the following RCM commands:

- `scriptinfo` – Get script information
- `register` – Registers the resources the script handles
- `resourceinfo` – Get resource information
- `queryremove` – Queries whether the resource can be released
- `preremove` – Releases the resource

For more information about these RCM commands, refer to `rcmscript`(4) man page and the Solaris RCM documentation.

See "Testing the RCM Script Example" on page 105 for a test run of this script in a system controlled by a typical Netra CP2100 series CompactPCI board.

CODE EXAMPLE 6-1 RCM Script Example (`SUNW,cp2000_io.pl`)

```
#! /usr/bin/perl -w
#
# Copyright 2002 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
#ident  "@(#)SUNW,cp2000_io.pl 1.1     02/03/28 SMI"
#
# A Sample site customization RCM script.
#
# When RCM_ENV_FORCE is FALSE, the script indicates to RCM that it can not
# release the network interface when the interface is plumbed and up
# When RCM_ENV_FORCE is TRUE, this script allows DR to remove the QFE device
# by bringing down the network interface that has been plumbed and up.
#
# For more information on RCM scripts see the man page rcmscript(4).
#

use strict;

my ($cmd, $rsrc, %dispatch);
```

CODE EXAMPLE 6-1 RCM Script Example (`SUNW,cp2000_io.pl`) *(Continued)*

```
# dispatch table for RCM commands
%dispatch = (
        "scriptinfo"    =>      \&do_scriptinfo,
        "register"      =>      \&do_register,
        "resourceinfo"  =>      \&do_resourceinfo,
        "queryremove"   =>      \&do_preremove,
        "preremove"     =>      \&do_preremove
);

sub do_scriptinfo
{
        print "rcm_script_version=1\n";
        print "rcm_script_func_info=ifconfig coordinator for DR\n";

        #
        # optionally specify command timeout value in seconds to override
        # the default timeout value.
        # Eg:
        #   print "rcm_cmd_timeout=10\n";
        #

        exit (0);
}

sub do_register
{
        #
        # register all resource names of interest using
        # print "rcm_resource_name=resourcename\n";
        # Eg: to register /dev/rmt/0 and /dev/dsk/c1t1d0s0
        #   print "rcm_resource_name=/dev/rmt/0\n";
        #   print "rcm_resource_name=/dev/dsk/c1t1d0s0\n";
        #

        # register all resource names of interest

        my ($devname);
        $devname='/devices/pci@1f,0/pci@1/pci@1/pci@12/SUNW,qfe@0,1';
        print "rcm_resource_name=$devname\n";
        exit(0);
}

sub do_resourceinfo
{
        #
        # specify the resource usage information of the given resource $rsrc
        #
```

CODE EXAMPLE 6-1 RCM Script Example (`SUNW,cp2000_io.pl`) *(Continued)*

```
        print "rcm_resource_usage_info=ifconfig managed QFE device\n";
        exit (0);
}

sub do_preremove
{
        if ($ENV{RCM_ENV_FORCE} eq 'TRUE') {
                if ($cmd eq 'preremove') {
                                        `/usr/sbin/ifconfig qfe0 down`;
                                        if ($? == 0) {
                                                `/usr/sbin/ifconfig qfe0 unplumb`;
                                        }
                        }
                exit(0);
              } else {
                print "rcm_failure_reason=device in use by ifconfig\n";
                exit (3);
        }
}

$ENV{'RCM_ENV_FORCE'} = "TRUE";
$cmd = $ARGV[0];
if (defined($ARGV[1])) {
        # resource name
        $rsrc = $ARGV[1];
}

if (defined($dispatch{$cmd})) {
        &{$dispatch{$cmd}};
} else {
        # unsupported command
        exit (2);
}
```

Testing the RCM Script Example

This section demonstrates how the `SUNW,cp2000_io.pl` RCM script (CODE EXAMPLE 6-1) unconfigures a network interface of a Sun Dual FastEthernet/SCSI 6U CompactPCI adapter.

In this example, the test system is operating in full hot-swap mode and has the following configuration:

- 8-slot High-Availability (HA) CompactPCI chassis
- Netra CP2100 series board used as the system board controller (SBC)
- Sun Dual FastEthernet/SCSI 6U CompactPCI adapter in slot 4
- Netra CP2100 series board used as a satellite CPU board in slot 5

The `cfgadm` command output below shows that the Sun Dual FastEthernet/SCSI 6U CompactPCI adapter is inserted and configured in slot 4 (`pci1:cpci0_slot4`) of the chassis.

```
# cfgadm
Ap_Id                          Type         Receptacle   Occupant     Condition
c0                              scsi-bus     connected    configured   unknown
c1                              scsi-bus     connected    unconfigured unknown
pci1:cpci0_slot2                unknown      connected    unconfigured unknown
pci1:cpci0_slot3                unknown      connected    unconfigured unknown
pci1:cpci0_slot4                stpcipci/fhs connected    configured   ok
pci1:cpci0_slot5                mcd/fhs      connected    configured   ok
pci1:cpci0_slot6                unknown      connected    unconfigured unknown
pci1:cpci0_slot7                unknown      connected    unconfigured unknown
pci1:cpci0_slot8                unknown      connected    unconfigured unknown
```

Using the `ifconfig` command, the first FastEthernet device (`qfe0`) is plumbed (the streams needed for TCP/IP are set up) and brought up. Finally, the `ifconfig -a` command output shows that the adapter's `qfe0` device has been connected to a Ethernet network, where `test_ip` is the hostname that corresponds to the 192.168.210.225 IP address. Refer to the `ifconfig`(1M) man page for more information about using `ifconfig` to configure network devices.

```
# ifconfig qfe0 plumb
# ifconfig qfe0 test_ip up
# ifconfig -a
lo0: flags=1000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
eri0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 192.168.210.213 netmask ffffff00 broadcast 192.168.210.255
        ether 0:3:ba:3:f4:58
qfe0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 3
        inet 192.168.210.225 netmask ffffff00 broadcast 192.168.210.255
        ether 0:3:ba:3:f4:58
```

At this point, the operator opens the ejection levers of the Sun Dual FastEthernet/SCSI 6U CompactPCI adapter to begin the extraction process. When the levers are opened, the Solaris DR framework (in conjunction with the Solaris RCM framework) calls the `SUNW,cp2000_io.pl` RCM script, which unplumbs and brings down the FastEthernet (`qfe0`) interface. The Solaris DR framework then unconfigures the adapter and turns on the adapter's blue extraction LED, signalling that it can be safely removed.

The `cfgadm` and `ifconfig` output below show that the card has been successfully unconfigured from the system.

```
# cfgadm
Ap_Id                          Type         Receptacle   Occupant     Condition
c0                              scsi-bus     connected    configured   unknown
c1                              scsi-bus     connected    unconfigured unknown
pci1:cpci0_slot2                unknown      connected    unconfigured unknown
pci1:cpci0_slot3                unknown      connected    unconfigured unknown
pci1:cpci0_slot4                unknown      connected    unconfigured unknown
pci1:cpci0_slot5                mcd/fhs      connected    configured   ok
pci1:cpci0_slot6                unknown      connected    unconfigured unknown
pci1:cpci0_slot7                unknown      connected    unconfigured unknown
pci1:cpci0_slot8                unknown      connected    unconfigured unknown

# ifconfig -a
lo0: flags=1000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4> mtu 8232 index 1
        inet 127.0.0.1 netmask ff000000
eri0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 192.168.210.213 netmask ffffff00 broadcast 192.168.210.255
        ether 0:3:ba:3:f4:58
```

Avoiding Error Messages When Extracting Devices in Basic Hot-Swap Mode

If you attempt to extract a hardware device using the `cfgadm` command when the system is operating in basic hot-swap mode, and you have installed an RCM script for the device, you may see error messages displayed in the system console. To avoid seeing these error messages, use the `-f` option with the `cfgadm` command when unconfiguring hardware devices. Using the `-f` option will help avoid any possible interference with other registered Solaris RCM modules or user-created RCM scripts.

For example, if you attempt to extract the Sun Dual FastEthernet/SCSI 6U CompactPCI adapter from a system operating in basic hot-swap mode, and you have installed the RCM script shown in CODE EXAMPLE 6-1, you may see error messages produced by other Solaris RCM modules in the system's console. For instance, if the adapter has a plumbed IP address, the Solaris `SUNW,ip_rcm.so` RCM module will fail the dynamic removal operation and display error messages unless you use the `cfgadm -f` command. By using the `cfgadm -f` command, you will avoid interfering with any other registered Solaris RCM modules or scripts.

The purpose of the `SUNW,ip_rcm.so` module is to protect anonymous consumers from inadvertent denial of service. Therefore, if an IP address is plumbed on a network interface card, the `SUNW,ip_rcm.so` module will fail the removal operation unless you unconfigure the card with the `-f` option.

However, if your Netra CP2100 series board controlled system is operating in full hot-swap mode, the Solaris hot-swap framework will automatically apply the `-f` option. Consequently, network cards with plumbed IP addresses will be unconfigured successfully even if there is no user-level RCM script installed. The `-f` option will force the `SUNW,ip_rcm.so` module to release the board and properly shut down all applications using the network card.

For more information about the `cfgadm` command, refer to the `cfgadm`(1M) man page and the Solaris system administration documentation. You can view this documentation on the `http://docs.sun.com/` website.

Index

R

S

T

U

W