Netra™ CP2300 cPSB Board Programming Guide

for Solaris Operating Environment

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-1331-10
May 2003, Revision A

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

Copyright 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.sun.com/patents and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and in other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, AnswerBook2, docs.sun.com, Netra, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and in other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and in other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

U.S. Government Rights—Commercial use. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie qui est décrit dans ce document. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à http://www.sun.com/patents et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats-Unis et dans les autres pays.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y ena.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, AnswerBook2, docs.sun.com, Netra, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits protant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une license non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciées de Sun qui mettent en place l'interface d 'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.

Adobe PostScript™

Contents

Preface xiii

1. Watchdog Timer 1

Watchdog Timers 1

Watchdog Timer Driver 2

Operations on the Watchdog Timers 3

Parameters Transfer Structure 3

Input/Output Controls 7

Errors 8

Example 8

Configuration 11

OpenBoot PROM Interface 11

Data Structure 12

Watchdog Operation 12

Commands at OpenBoot PROM Prompt 12

Corner Cases 13

Setting the Watchdog Timer at OpenBoot PROM 13

2. User Flash 15

User Flash Usage and Implementation 15

System Compatibility 16
User Flash Driver 16
Switch Settings 17
OpenBoot PROM Device Tree and Properties 17
User Flash Device Files 17
Interface (Header) File 17
Application Programming Interface 18
Structures to Use in IOCTL Arguments 19
Errors 20
Example Programs 20
Sample User Flash Application Program 27

3. Advanced System Management 35
ASM Component Compatibility 36
Typical ASM System Application 37
Typical Cycle From Power Up to Shutdown 38
ASM Protection at the OpenBoot PROM 38
ASM Protection at the Operating Environment Level 39
Post Shutdown Recovery 40
Hardware ASM Functions 41
Power On/Off Switching 45
Inlet, Exhaust, and CPU Temperature Monitoring 45
Adjusting the ASM Warning, Critical, and Shutdown Parameter Settings on the Board 46
OpenBoot PROM Environmental Parameters 47
OpenBoot PROM/ASM Monitoring 49
CPU Monitoring 49
show-sensors Command at OpenBoot PROM 51
IPMI Command Examples at OpenBoot PROM 51

ASM Application Programming 57
Specifying the ASM Polling Rate 58
Monitoring the Temperature 58
Solaris Driver Interface 58
Sample Application Program 59

4. Programming the User LED 63
Files and Packages Required to Support the Alarm/User LED 65
Applications 65
Application Programming Interface (API) 66
Compile 67
Link 67
Sample Application Program 68

Index 69

Figures

FIGURE 3-1 Typical ASM Application Block Diagram 37

FIGURE 3-2 Location of ASM Hardware on the Netra CP2300 cPSB Board (Top Side) 42

FIGURE 3-3 Location of ASM Hardware on the Netra CP2300 cPSB Board (Bottom Side) 43

FIGURE 3-4 Netra CP2300 cPSB Board ASM Functional Block Diagram 44

FIGURE 4-1 Illustration of a Typical Netra CP2300 cPSB Board Front Panel Showing the Alarm/User LED 64

Tables

TABLE 1-1 OpenBoot PROM Prompt Commands 13

TABLE 2-1 User Flash Node Properties 17

TABLE 2-2 System Calls 18

TABLE 3-1 Compatible ASM Components 36

TABLE 3-2 Typical Netra CP2300 cPSB Board Hardware ASM Functions 41

TABLE 3-3 I2C Components 41

TABLE 3-4 Reported Temperature Readings at an Ambient Room Temperature of 21˚C on a Typical Netra CP2300 cPSB Board 47

TABLE 3-5 Typical Netra CP2300 Board Temperature Thresholds and Firmware Action 48

TABLE 4-1 Supported LED and Command Combinations for the Netra CP2300 Board 67

Code Samples

CODE EXAMPLE 1-1 Include File wd_if.h 4
CODE EXAMPLE 1-2 Status of Watchdog Timers and Starting Timers 8
CODE EXAMPLE 2-1 PROM Information Structure 19
CODE EXAMPLE 2-2 User Flash Interface Structure 19
CODE EXAMPLE 2-3 Read Action on User Flash Device 20
CODE EXAMPLE 2-4 Write Action on User Flash Device 22
CODE EXAMPLE 2-5 Erase Action on User Flash Device 24
CODE EXAMPLE 2-6 Block Erase Action on User Flash Device 25
CODE EXAMPLE 2-7 Sample User Flash Application Program 27
CODE EXAMPLE 3-1 Input Output Control Data Structure 59
CODE EXAMPLE 3-2 Sample ASM Application Program 59
CODE EXAMPLE 4-1 Application Programming Interface for the Netra CP2300 Board 66
CODE EXAMPLE 4-2 Sample LED Application Program 68

Preface

The Netra™ CP2300 compactPCI Packet Switched Backplane (cPSB) board is 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 CP2300 cPSB Board Programming Guide* is written for program developers and users who want to program this board 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.

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 CP2300 cPSB board watchdog timer driver and its operation.

Chapter 2 describes the user flash driver for the Netra CP2300 cPSB board onboard flash PROMs and how to use it.

Chapter 3 describes the specific Advanced System Management (ASM) functions of the Netra CP2300 cPSB board.

Chapter 4 describes how to program the User LED on the Netra CP2300 cPSB board.

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

Title	Part Number
Netra CP2300 cPSB Board Product Note	816-7185
Netra CP2300 cPSB Board Installation and Technical Reference	816-7186
Netra CP2300 cPSB Board Programming Guide	817-1331
Netra CP2300 cPSB Board Transition Card Product Note	816-7187
Netra CP2300 cPSB Board Transition Card Installation and Technical Reference	816-7188
Netra CP2300 cPSB Board Release Notes	817-1741
Important Safety Information for Sun Hardware Systems	816-7190

Accessing Sun Documentation

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

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

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

`http://www.sun.com/service/contacting`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

`http://www.sun.com/hwdocs/feedback`

Please include the title and part number of your document with your feedback:

Netra CP2300 cPSB Board Programming Guide, part number 817-1331-10

CHAPTER **1**

Watchdog Timer

The System Management Controller (SMC) on the Netra CP2300 cPSB 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 CP2300 cPSB board. This chapter also describes the user-level application programming interface (API) and behavior of the Netra CP2300 cPSB 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" on page 1
- "Watchdog Timer Driver" on page 2
- "Operations on the Watchdog Timers" on page 3
- "Parameters Transfer Structure" on page 3
- "Input/Output Controls" on page 7
- "Data Structure" on page 12
- "Watchdog Operation" on page 12

Watchdog Timers

There are two watchdog timers:

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

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 SPARC CPU through the isa bus and the WD2 pre-timeout counter is set to a nonzero value when interrupt option is enabled. Otherwise the SMC resets the SPARC 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 SPARC CPU 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 CP2300 cPSB board 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 forces the SPARC™ processor to reset. The maximum range for WD2 is 255 seconds.

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 CP2300 cPSB board, the SMC-based watchdog timers are not independent. The WD2 timer is a continuation of the WD1 timer. There are some behavioral consequences to this implementation that result in the Netra CP2300 cPSB board 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))... */
```

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

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


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 behavioral consequences are associated with the WIOCSTART command that arise from the fact that WD1 and WD2 timers are not independent in the Netra CP2300 cPSB 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;
```

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

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

- SUNW,Netra-CP2300

The watchdog configuration file resides in /platform/***implementation***/kernel/drv. The watchdog driver binary resides in /platform/***implementation***/kernel/drv/sparcv9. The value of ***implementation*** for a given Netra CP2300 cPSB board system can be obtained by running the uname(1) command on that machine with the -i option:

```
# uname -i
SUNW, Netra-CP2300
```

The wdog.conf driver configuration 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 CP2300 cPSB 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="off";
```

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 open and start the watchdog timer.

If `watchdog-enable?` is set to `true`, the watchdog timer is enabled at boot time with its default actions, as follows. The WD1 timer is controlled by the value in the `watchdog-timeout` variable. The default value for `watchdog-timeout` is 65535 (in the unit of one-tenth of a second). When WD1 expires, it sends an asynchronous message to the SPARC CPU and starts the WD2 timer. The default value for WD2 is one second. If WD2 expires, it resets the system.

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 watchdog timer may reset the SPARC CPU when the watchdog expires.

Data Structure

For information on the data structure that is used with watchdog timer programs, refer to CODE EXAMPLE 1-1.

Watchdog Operation

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

Commands at OpenBoot PROM Prompt

Commands for `smc` are available in the SMC controller device mode (`/pci@1f,0/pci@1,1/isa@7/sysmgmt@0,8010 alias hsc`). You need to go to the `sysmgmt` node before executing the `smc` commands and execute the following once:

```
ok dev hsc
```

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 SPARC CPU is restarted, the SPARC CPU must restart the watchdog timer.

The SPARC CPU must perform a corner case. After the SMC resets the SPARC CPU, the output buffer full (OBF) bit and OEM1 bit in the isa bus status register remain set. Since this is a read-only bit, the SMC cannot reset the bit. The SPARC CPU must ignore the status bits and clear the OBF bit by reading one byte of data from the isa bus. This action must be performed after watchdog reset. Otherwise, the SPARC 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 SPARC CPU. The code is set up on the SPARC CPU 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 application program 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

The examples in this section are performed at the OpenBoot PROM level.

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

In this example, after level one expires, the CPU is 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. In this example, after level one expires, there are 80 seconds before the reset.

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

   Set the countdown value to 10 minutes, as in the previous procedure, and set the pre-timeout 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 PROM and how to use it. The Netra CP2300 cPSB board is equipped with user flash memory. This chapter includes the following sections:

- “User Flash Usage and Implementation” on page 15
- “System Compatibility” on page 16
- “User Flash Driver” on page 16
- “Application Programming Interface” on page 18
- “Example Programs” on page 20

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.

The user flash includes a flash PROM chip that can be programmed. The Netra CP2300 cPSB board has an 8MB flash that is logically divided into two parts: 1MB for the system/boot flash and 7MB for the user flash. The physical address for the flash is 1ff.f000.0000.

System Compatibility

The following releases support the user flash driver:

- Solaris 8 2/02 operating environment or other compatible versions that support this feature
- Netra CP2300 cPSB board OpenBoot PROM
  - Firmware version 1.0.1
  - Firmware CORE Release 1.0.3
  - Release 4.0 Version 16
  - SMCFW FLASH Code Version 4.0.12
  - SMCFW BOOT Code Version 4.15.1
  - PLD Revision 1.1
  - CORE 1.0.3
  - All the above versions or other compatible versions that support this feature

User Flash Driver

The *uflash* is the device driver for the flash PROM device on the Netra CP2300 cPSB board. Access to the driver is carried out through `open`, `read`, `write`, `pread`, `pwrite` and `ioctl` system interfaces.

On the Netra CP2300 cPSB board, one of these devices is 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.

An instance of the driver is loaded for the 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 one at a time. All read and write operations are supported at this time. Access to the device normally happens a byte at a time.

The device also supports erase and lock features. Applications can use them through the IOCTL interface. The device is 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 CP2300 cPSB board are write enabled by default. The user flash is detected during OpenBoot PROM boot by default.

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 is:

`/pci@1f,0/pci@1,1/isa@7/flashprom@1f,100000`

See TABLE 2-1 for the user flash node properties.

TABLE 2-1 User Flash Node Properties

Property	Description/Value
dcode-offset	`00000002`
blocks-per-bank	`00000038`
block-size	`00020000`
model	`SUNW,yyy-yyyy`
compatible	`nct-uflash`
reg	`0000001f 00100000 00700000`

User Flash Device Files

The user flash device files are as follows:

- `/dev/uflash0`

Interface (Header) File

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

`/usr/`***platform***`/SUNW,Netra-CP2300/include/sys/uflash_if.h`

Application Programming Interface

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

The system calls are listed below in TABLE 2-2.

TABLE 2-2 System Calls

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

The `ioctl` supported commands are listed below:

```
#define UIOCIBLK  (uflashIOC|0) /* identify */
#define UIOCQBLK  (uflashIOC|1) /* query a block */
#define UIOCLBLK  (uflashIOC|2) /* lock a block */
#define UIOCCLCK  (uflashIOC|4) /* clear all locks */
#define UIOCEBLK  (uflashIOC|5) /* erase a block */
```

Note that these `ioctl` commands are *not* supported:

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

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

```
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <uflash_if.h>
char *uflash0 = "/dev/uflash0";
int ufd0;
uflash_if_t ufif0;
char *buf0;
char *module;
static int
uflash_init() {
  char *buf0 = malloc(ufif0.info.blk_size);
if (!buf0) {
    printf("%s: cannot allocate memory\n", module);
    return(-1);
  }
/* open device */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
    exit(1);
  }
/* get uflash sizes */
  if (ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufif0.info.mfr_id);
    printf("device id = 0x%p\n", ufif0.info.dev_id);
    printf("number of blocks = 0x%p", ufif0.info.blk_num);
    printf("block size = 0x%p"  ufif0.info.blk_size);
  }
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
cleanup:
  if (buf0)
    free(buf0);
}
static int
uflash_read() {
```

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

```
  /* read block 0 of user flash */
  if (pread(ufd0, buf0, ufif0.info.blk_size, 0) !=
ufif0.info.blk_size)
        perror("uflash0: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";
int ufd0;
uflash_if_t ufif0;
char *buf0;
char *module;
static int
uflash_init() {
  char *buf0 = malloc(ufif0.info.blk_size);
if (!buf0) {
```

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

```
    printf("%s: cannot allocate memory\n", module);
    return(-1);
  }
/* open device */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
    exit(1);
  }
/* get uflash sizes */
  if (ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufif0.info.mfr_id);
    printf("device id = 0x%p\n", ufif0.info.dev_id);
    printf("number of blocks = 0x%p", ufif0.info.blk_num);
    printf("block size = 0x%p"  ufif0.info.blk_size);
  }
}
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
cleanup:
  if (buf0)
    free(buf0);
}
static int
uflash_write() {
  int i;
/* write some pattern to the buffers */
  for (i = 0; i < ufif0.info.blk_size; i += sizeof(int))
        *((int *) (buf0 + i)) = 0xDEADBEEF;
/* write block 0 of user flash */
  if (pwrite(ufd0, buf0, ufif0.info.blk_size, 0) !=
ufif0.info.blk_size)
        perror("uflash0:write");
return(0);
}
main() {
  int ret;
```

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

```
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";
int ufd0;
uflash_if_t ufif0;
char *module;
static int
uflash_init() {
 /* open device */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
    exit(1);
  }
/* get uflash sizes */
  if (ufd0 && ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
```

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

```
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufif0.info.mfr_id);
    printf("device id = 0x%p\n", ufif0.info.dev_id);
    printf("number of blocks = 0x%p", ufif0.info.blk_num);
    printf("block size = 0x%p"  ufif0.info.blk_size);
  }
}
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
}
static int
uflash_erase() {
  if (ufd0 && ioctl(ufd0, UIOCEFUL, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCEFUL): ");
    return(-1);
  }
  printf("\nerase successful on %s\n", uflash0);
return(0);
}
main() {
  int ret;
module = argv[0];
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>
```

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

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <uflash_if.h>
char *uflash0 = "/dev/uflash0";
int ufd0;
uflash_if_t ufif0;
char *module;
static int
uflash_init() {
 /* open device */
  if ((ufd0 = open(uflash0, O_RDWR)) == -1 ) {
    perror("uflash0: ");
    exit(1);
  }
/* get uflash sizes */
  if (ioctl(ufd0, UIOCIBLK, &ufif0) == -1 ) {
    perror("ioctl(ufd0, UIOCIBLK): ");
    exit(1);
  }
if (ufd0) {
    printf("%s: \n", uflash0);
    printf("manfacturer id = 0x%p\n", ufif0.info.mfr_id);
    printf("device id = 0x%p\n", ufif0.info.dev_id);
    printf("number of blocks = 0x%p", ufif0.info.blk_num);
    printf("block size = 0x%p"  ufif0.info.blk_size);
  }
}
static int
uflash_uninit() {
  if (ufd0)
    close(ufd0);
}
static int
uflash_blockerase() {
  /* erase 2 blocks starting from block 1 of user flash */
  uf0.blk_num = 1;
  uf0.num_of_blks = 2;
  if (ufd0 && ioctl(ufd0, UIOCEBLK, &ufif0) == -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);
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 accessed by
 *        means of ioctl() calls such as:
 *         -) identify the chip,
 *         -) query block,
 *         -) lock block/unlock block,
 *         -) master lock,
 *         -) erase block, erase all unlocked blocks, and
 *            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
```

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

```
 *        writing.
 *
 *       Use the following line to compile your custom application
 *        programs:
 *             make uflash_test
 */

#pragma ident   "@(#)uflash_test.c 1.0     03/04/30 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"
/*
      */
     #if 1
     #define PROM_SIZE 0x700000 /* 7 MBytes */
     #endif
static char *help[14] = {
       "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
```

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

```
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, h, i;
       int        fd, prom_id;
       uflash_if_t uflash_if;
       caddr_t r_buf, w_buf;
       char    *devname0 = "/dev/uflash0";
       char    c;

       r_buf = (caddr_t)malloc(PROM_SIZE);
       w_buf = (caddr_t)malloc(PROM_SIZE);

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

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

```
            switch(get_cmd()) {
            case ’q’:
                goto getout;

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

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

```
                                goto getout;
                break;

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

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

```
                i++)
            {
                fprintf(stderr, "block[%d] status = %x\n",
                    i, uflash_if.info.blk_status[i] & 0xF);
            }
            break;

        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 */
```

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

```
                printf ("Write process was failed at byte 0x%x \
n",
                        n_byte);
            }
            break;

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

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

```
} /* 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 CP2300 cPSB board supports the temperature and voltage monitoring functions of ASM. This chapter describes the specific ASM functions of the Netra CP2300 cPSB board. This chapter includes the following sections:

- "ASM Component Compatibility" on page 36
- "Typical ASM System Application" on page 37
- "Typical Cycle From Power Up to Shutdown" on page 38
- "Hardware ASM Functions" on page 41
- "Adjusting the ASM Warning, Critical, and Shutdown Parameter Settings on the Board" on page 46
- "OpenBoot PROM Environmental Parameters" on page 47
- "OpenBoot PROM/ASM Monitoring" on page 49
- "ASM Application Programming" on page 57

ASM Component Compatibility

TABLE 3-1 lists the compatible ASM hardware, OpenBoot PROM, and Solaris operating environment for the Netra CP2300 cPSB board.

TABLE 3-1 Compatible ASM Components

Component	ASM Compatibility
Hardware	Board supports ASM
OpenBoot PROM	ASM is supported by OpenBoot PROM.
Operating environment	Solaris 8 2/02 operating environment or subsequent compatible versions

Typical ASM System Application

FIGURE 3-1 illustrates the Netra CP2300 cPSB board ASM application block diagram.

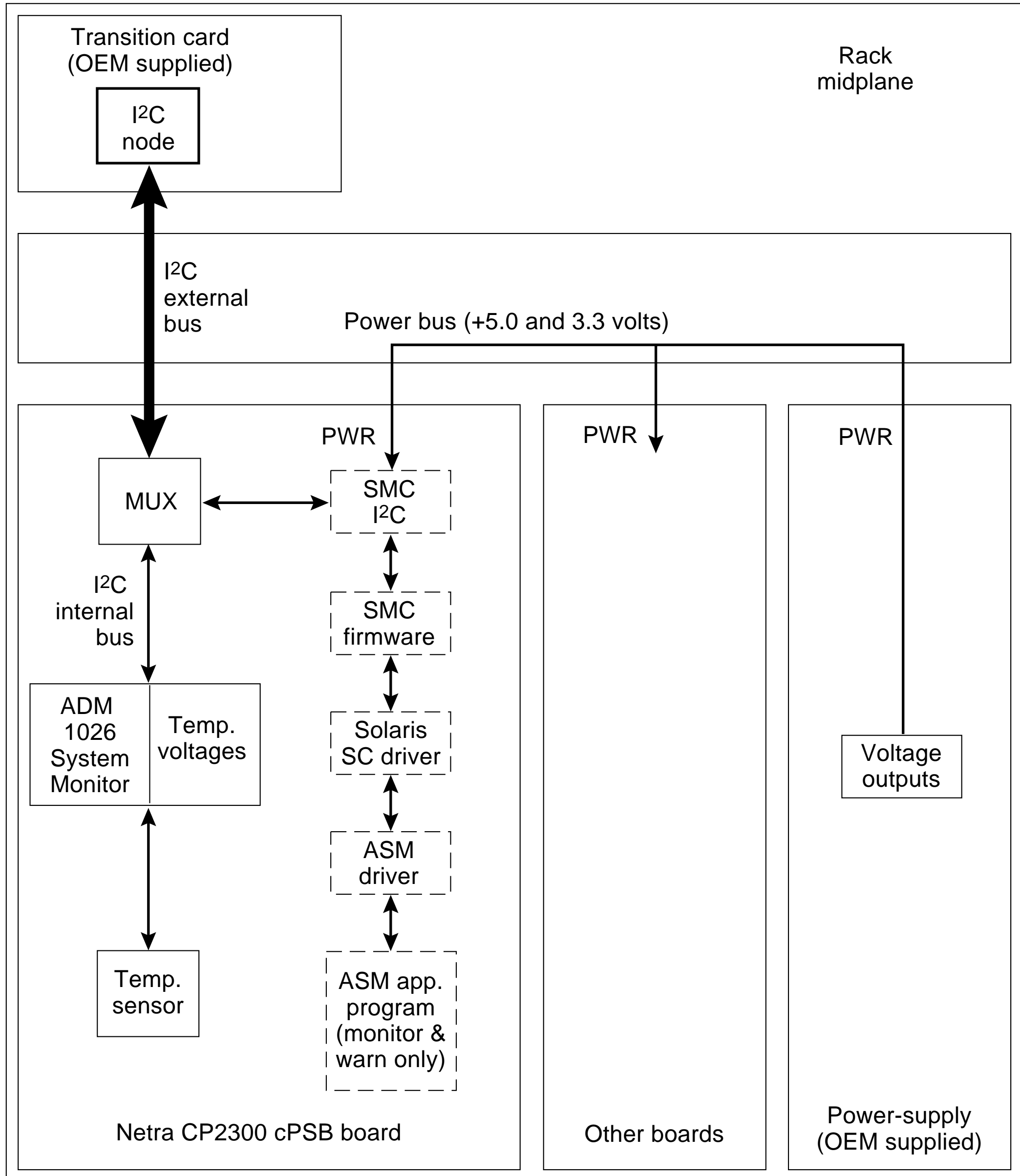


FIGURE 3-1 Typical ASM Application Block Diagram

FIGURE 3-1 is a typical Netra CP2300 cPSB board system application block diagram. For locations of the temperature sensors, see FIGURE 3-2 and FIGURE 3-3.

The Netra CP2300 cPSB board functions as a node board in a cPSB system rack. The Netra CP2300 cPSB board monitors its CPU diode 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 board. At the OBP level, the CPU diode temperature is monitored if 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 the CPU diode temperature at the fixed polling rate of 10 seconds and 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), the critical temperature (the `critical-temperature` parameter), or the shutdown temperature (the `shutdown-temperature` parameter). See "OpenBoot PROM Environmental Parameters" on page 47 for information on changing these pre-programmed parameters.

OpenBoot PROM-level protection takes place only when the `env-monitor` parameter is enabled (it is not the default setting). If the NVRAM variable `env-monitor` is set to `enabled-with-shutdown` (`env-monitor=enabled-with-shutdown`), and if the board temperature exceeds the shutdown temperature, the OpenBoot PROM will shut down power to the Netra CP2300 cPSB board CPU. If the NVRAM variable `env-monitor` is set to `enabled` (`env-monitor=enabled`), the OpenBoot PROM will send a warning, critical, or shutdown temperature message to the user that the Netra CP2300 cPSB board is overheating.

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 node board and pushes the ASM streams module, the ASM module is loaded.

To access the CPU diode 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 CPU, inlet, and exhaust temperature sensors once every polling cycle. The program then compares the measured CPU diode 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 diode 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 CP2300 cPSB board Ethernet port.

Refer to "Sample Application Program" on page 59 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 power sequencing/monitor.

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, if the NVRAM variable `env-monitor` is set to `enabled-with-shutdown` (`env-monitor=enabled-with-shutdown`), and if the board temperature exceeds the shutdown temperature, the OpenBoot PROM will shut down power to the Netra CP2300 cPSB board CPU.

There is no mechanism for the Solaris operating environment to either recover or restore power to the Netra CP2300 cPSB board when an unusual condition occurs (for example, if the CPU diode temperature exceeds its maximum recommended level). In either case, the end user must intervene and manually recover the Netra CP2300 cPSB board as well as the cPSB system through hardware control. Once a shutdown has occurred, you can recover the board using a cold-reset IPMI command to SMC or by extracting and reinserting the board.

Hardware ASM Functions

This section summarizes the hardware ASM features on the Netra CP2300 cPSB board. TABLE 3-2 lists the ASM functions on a Netra CP2300 cPSB board.

TABLE 3-2 Typical Netra CP2300 cPSB Board Hardware ASM Functions

Function	Capability
Board Exhaust Air Temperature	Senses the air temperature at the trailing edge of the board. (Assumes air direction from the processor/heatsink toward the PMC slots.)
CPU Diode Temperature	Senses a diode temperature in the processor junction.
Board Inlet Air Temperature	Senses the air temperature at the leading edge of the board under the solder-side cover. (Assumes air direction from the processor/heatsink toward the PMC slots.)

TABLE 3-3 shows the I^2C components.

TABLE 3-3 I^2C Components

Component	Function
DS80CH11	SMC I^2C controller - IPMB
PCF8584	I^2C controller
PCF9545	4 channel I^2C multiplexor
AT24C64	I^2C EEPROM - motherboard FRUID
AT24C01	I^2C EEPROM - RTM FRUID + external I^2C header
ADM1026	System monitor/general purpose I/O
AT24C01	I^2C EEPROM - onboard memory SPD
DS1307	I^2C TOD
AT24C64	I^2C EEPROM - NVRAM/Ethernet MAC ID
LTC4300	I^2C hotswap isolator
AT24C*xx*	I^2C EEPROM - SO DIMM 1 SPD (add-on dependent)
AT24C*xx*	I^2C EEPROM - SO DIMM 0 SPD (add-on dependent)
AT24C*xx*	PMC/PTMC B (add-on card dependent)

TABLE 3-3 I^2C Components *(Continued)*

Component	Function
AT24C*xx*	PMC/PTMC A (add-on card dependent)
87LPC764	“IMAX” configurable 4 channel I^2C multiplexor
ALi1535D+	Southbridge - SMBUS/I^2C controller

FIGURE 3-2 and FIGURE 3-3 show the location of the ASM hardware on the Netra CP2300 cPSB board.

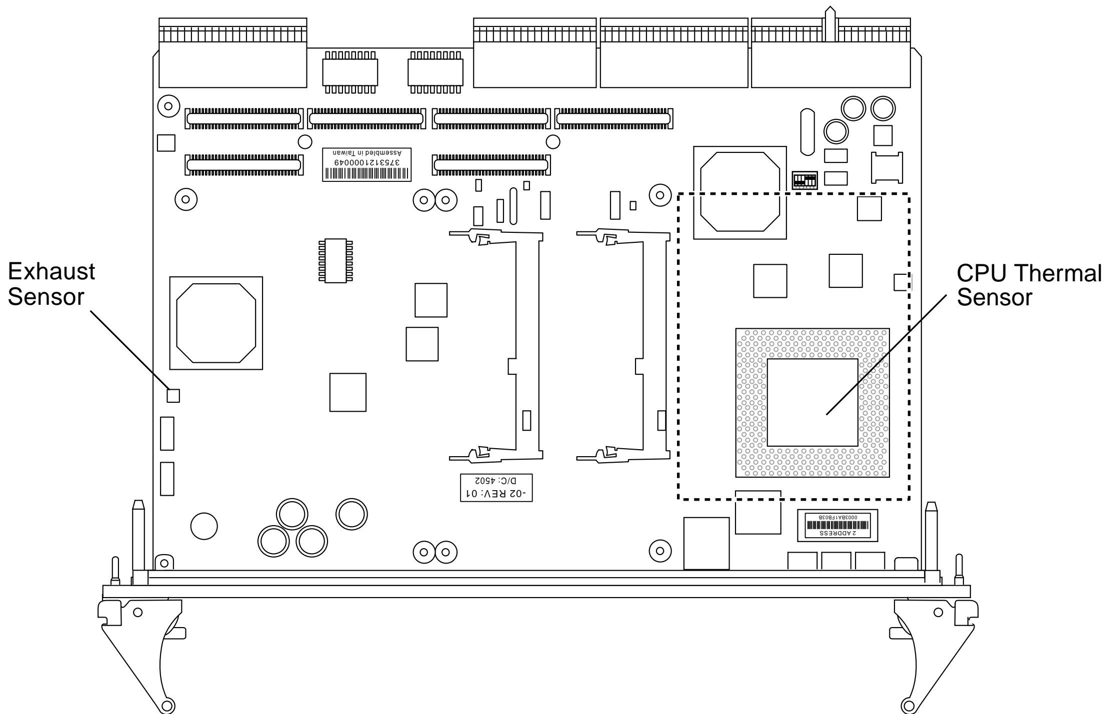


FIGURE 3-2 Location of ASM Hardware on the Netra CP2300 cPSB Board (Top Side)

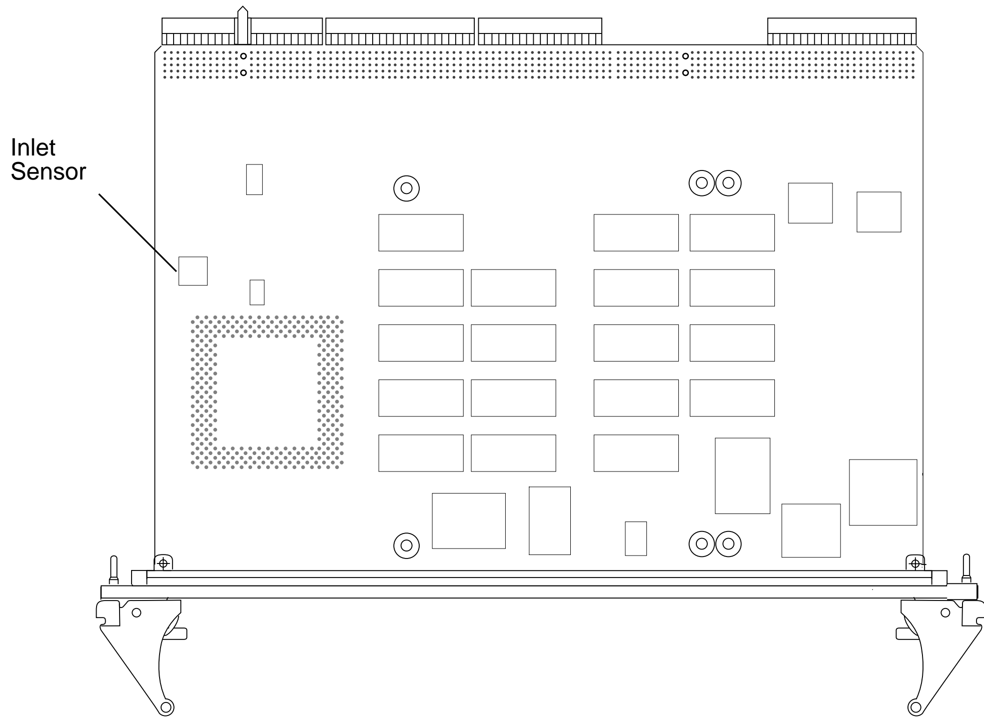


FIGURE 3-3 Location of ASM Hardware on the Netra CP2300 cPSB Board (Bottom Side)

FIGURE 3-4 is a block diagram of the ASM functions.

CPU
Die Temp
Sensor

CPU

ASM
Device
Driver

Solaris
Operating
Env

ASM
Application
Program

OEMs can write
their own device drivers

OEMs can implement
their own monitoring
and control logic

Southbridge

OBP

OEMS can modify
Factory Defaults

SMC Micro
Controller

I2C Controller

PLD

I2C Mux

cPCI J5
External I2C

Power
Control
and Monitor

Exhaust
Temp Sensor

ADM 1026
System
Monitor

OEMs are original equipment
manufacturers

Inlet Temp
Sensor

FIGURE 3-4 Netra CP2300 cPSB Board ASM Functional Block Diagram

Power On/Off Switching

The onboard voltage controller allows power to the CPU of the Netra CP2300 cPSB 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 CPU power of the board.

Inlet, Exhaust, and CPU Temperature Monitoring

The CPU diode sensor reading may vary from slot to slot and from board to board in a system, and is dependent primarily on system cooling. As an example, a system may have sensor readings for the CPU diode from 35˚C to 49˚C with an ambient inlet of 21˚C across many boards, with a variety of configurations and positions within a chassis. Care must be taken when setting the alarm and shutdown temperatures based on the CPU diode sensor value. This sensor typically is linear across the operating range of the board.

The exhaust sensor measures the local air temperature at the trailing edge of the board for systems with bottom to top airflow. This value depends on the character and volume of the airflow across the board. Typical values in a chassis may range from a delta over inlet ambient of 0˚C to 12˚C, depending on the power dissipation of the board configuration and the position in the chassis. The exhaust sensor is nonlinear with respect to ambient inlet temperature.

The inlet sensor measures the local air temperature at the leading edge of the board on the solder-side under the solder-side cover. This value typically can range from a reading of 0˚C to 13˚C above inlet system ambient in a chassis; care must be taken to understand the application and installation of the board to use this temperature sensor.

A sudden drop of all temperature sensors close to or near room ambient temperature can mean loss of power to one or more Netra CP2300 cPSB boards.

A gradual increase in the delta temperature from inlet to outlet can be due to dust clogging system filters. This feature can be used to set service levels for filter cleaning or changing.

The CPU diode 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, Critical, and Shutdown Parameter Settings on the Board

The Netra CP2300 cPSB 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, critical, 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, critical, and shutdown temperature parameter settings.

The Netra CP2300 cPSB 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 diode 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 51.

The temperature read by the CPU sensor will trigger OpenBoot PROM warning, critical, and shutdown messages. When the CPU 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 3-4 shows the sensor readings of a Netra CP2300 cPSB 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 3-4 Reported Temperature Readings at an Ambient Room Temperature of 21˚C on a Typical Netra CP2300 cPSB Board

Board Sensor Location	Reported Temperatures (in Degrees Celsius)	Difference Between Reported and Ambient Room Temperature (in Degrees Celsius)
CPU	41	20
Inlet 1	31	10
Exhaust 1	29	8

Since the temperature reported by the CPU diode sensor might be different than the actual CPU temperature, you may want to adjust the settings for the `warning-temperature`, `critical-temperature`, and `shutdown-temperature` OpenBoot PROM parameters. The default values of these parameters have been conservatively set at 60˚C for the warning temperature, 65˚C for the critical temperature, and 70˚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 57 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. On a Netra CP2300 cPSB board, 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 are shown in TABLE 3-5.

TABLE 3-5 Typical Netra CP2300 Board Temperature Thresholds and Firmware Action

Thresholds with Default	Firmware Action
warning-temperature = 60˚ C	OBP displays warning message
critical-temperature = 65˚ C	OBP displays warning message
shutdown-temperature = 70˚ C	OBP shuts down the CPU processor and the Netra CP2300 board if `env-monitor=enabled-with-shutdown`

Note that there is a lower limit of 50˚ C on shutdown-temperature value. If you try to set the temperature to a value lower than 50˚ C, OpenBoot PROM will not accept it. This safeguards a user from setting the shutdown-temperature lower than the room temperature and thereby causing the CPU processor and the Netra CP2300 cPSB 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 61
```

or:

```
ok setenv shutdown-temperature 72
```

The critical-temperature is a second-level warning temperature with a default value of 65˚ C. This variable can be modified using the OpenBoot PROM level `setenv` command as shown in example below:

```
ok setenv critical-temperature 66
```

OpenBoot PROM/ASM Monitoring

This section describes the ASM monitoring in the OpenBoot PROM.

CPU Monitoring

The following NVRAM module variables are in OpenBoot PROM for ASM.

- 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: 60
  - 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: 65
  - 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: 70
- 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. Setting these values too low will power down the system in an unpredictable manner.

Warning Temperature Response at OpenBoot PROM

When the CPU diode 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 : 60

The current temperature is : 61
```

Critical Temperature Response at OpenBoot PROM

When the CPU diode temperature reaches "critical-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 : 65

The current temperature is : 66
```

`show-sensors` Command at OpenBoot PROM

The `show-sensors` command at OpenBoot PROM displays the readings of all the temperature sensors on the board. A sample output for typical sensor readings for a Netra CP2300 cPSB board is as follows:

```
ok show-sensors
Sensor#    Sensor Name                            Sensor Reading
=======    ====================================   ===================
   1       EP 5v                      Sensor      (d7)  5.112 volts
   2       EP 3.3v                    Sensor      (8e)  3.408 volts
   3       BP +12v                    Sensor      (d3)  12.048 volts
   4       BP -12v                    Sensor      (62)  -12.020 volts
   5       IPMB Power                 Sensor      (d7)  5.088 volts
   6       SMC Power                  Sensor      (d7)  5.088 volts
   7       VDD 3.3v                   Sensor      (ac)  3.3368 volts
   8       VCCP                       Sensor      (90)  1.6992 volts
   9       +12v                       Sensor      (c2)  12.1250 volts
   a       -12v                       Sensor      (37)  -11.968 volts
   b       +5v                        Sensor      (c4)  5.096 volts
   c       Standby 3.3v               Sensor      (bf)  3.2852 volts
   d       Main 3.3v                  Sensor      (bf)  3.2852 volts
   e       External I  temp (CPU)     Sensor      (29)  41 degree C
   f       External II temp (Outlet)  Sensor      (1b)  31 degree C
  10       Internal    temp (Inlet)   Sensor      (1b)  29 degree C

Verifying Access to EEPROMs :

IPMI FRU EEPROM (EEPROM id 00) : Passed
SUN FRU EEPROM  (EEPROM id 20) : Passed
FRU EEPROM      (EEPROM id 21) : Passed
ADM chip EEPROM (EEPROM id 22) : Passed
ok
```

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 other boards in the 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 53. 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

The command `execute-smc-cmd` is available in SMC controller device mode (`/pci@1f,0/pci@1,1/isa@7/sysmgmt@0,8010 alias hsc`). You need to go to the `sysmgmt` node before executing the command `execute-smc-cmd` using the following:

```
ok dev hsc
```

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

   See "Set Sensor Threshold" on page 53. 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 53 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 56 and "Get Sensor Event Enable" on page 56.

   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 53 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.**

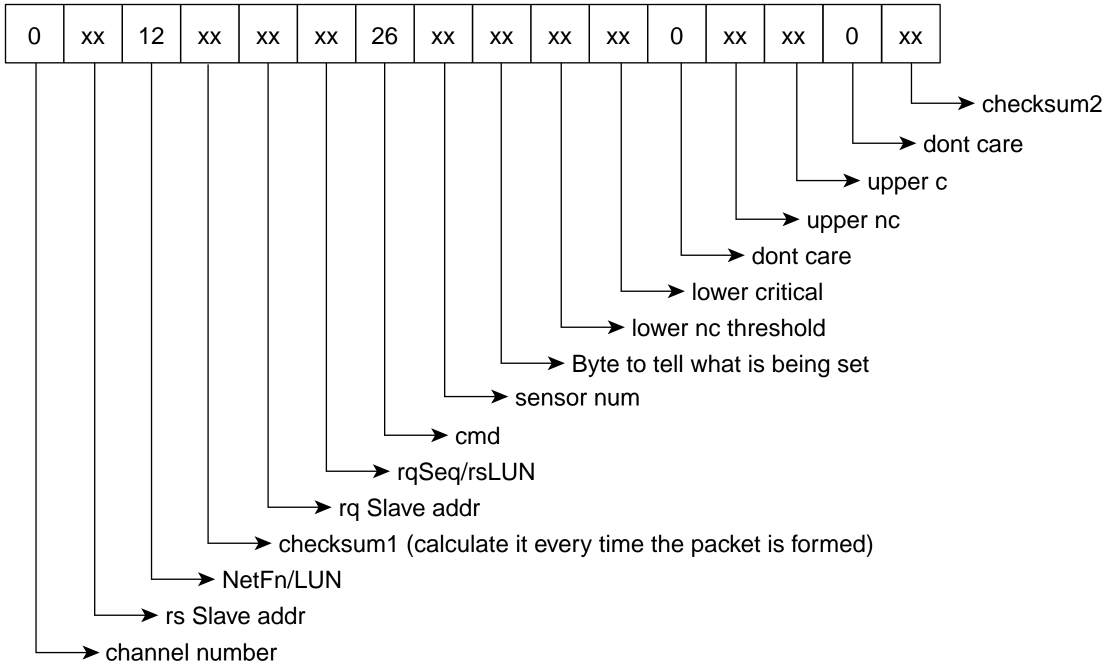
Examples of IPMI Command Packets

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 0 26 12 20 34 12 ba 0 10 34 execute-smc-cmd
```

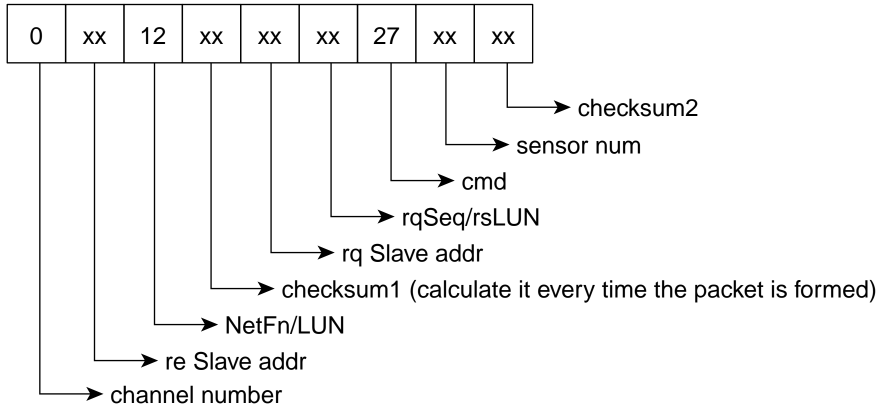


Note – In byte number 9, if the bit for a corresponding threshold is set to 1, then that threshold is set. If the bit is 0, the System Management Controller ignores that threshold. But if an attempt is made to set a threshold that is not supported, an error is returned in the command response.

Get Sensor Threshold

A typical example of the sensor command is as follows

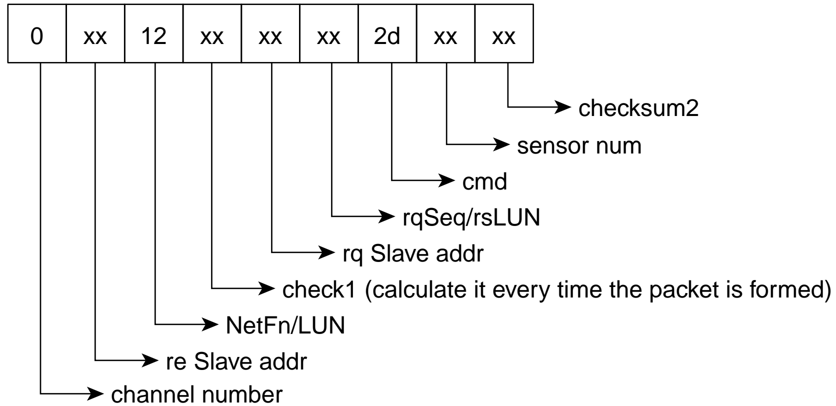
```
a5 0 27 12 20 34 12 ba 0 9 34 execute-smc-cmd
```



Get Sensor Reading

A typical example of the sensor command is as follows:

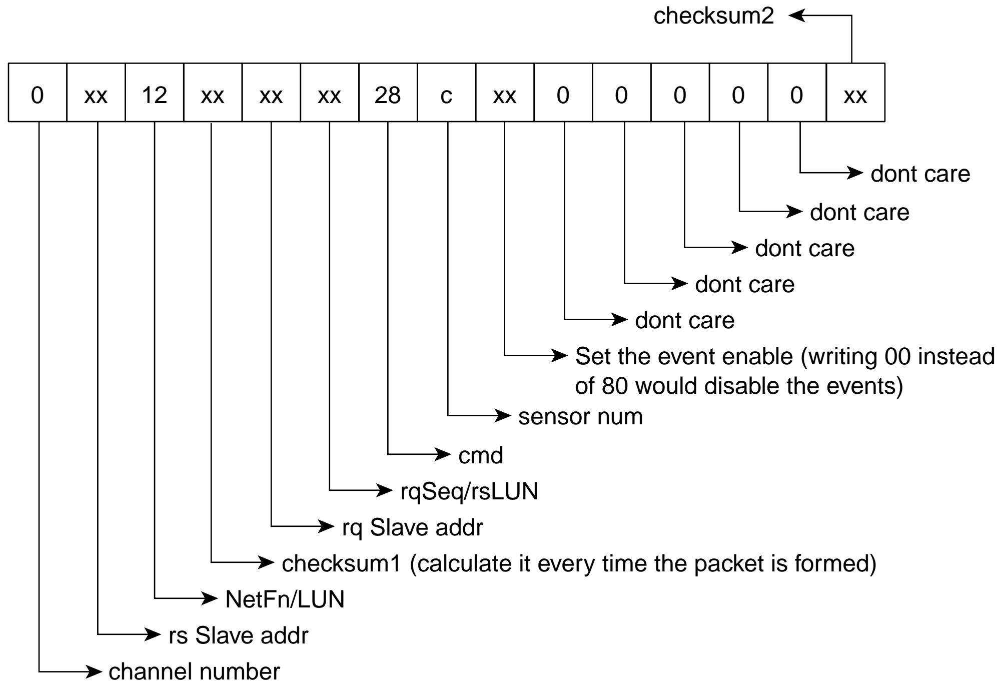
```
93 e 2d 12 20 34 12 ba 0 9 34 execute-smc-cmd
```



Set Sensor Event Enable Command

A typical example of the sensor command is as follows:

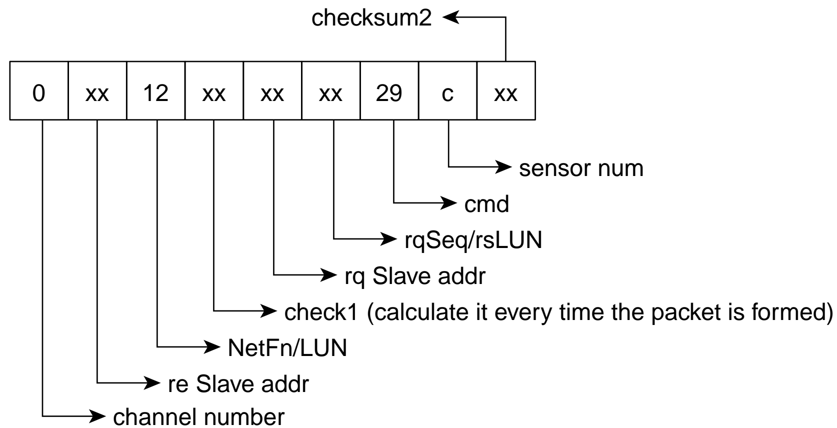
```
24 0 0 0 0 80 2 28 12 20 34 12 ba 0 e 34 execute-smc-cmd
```



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 47 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 diode temperature as follows (see “Sample Application Program” on page 59 for C code):

1. **Get the CPU diode 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 CP2300 cPSB board 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.

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;/* only local i2c supported - now not in use */
       uchar_t sensorValue;/* return sensor Temperature */
       uchar_t scportNum; /* scport number for SC driver */
       uchar_t sensorNum; /* sensor Number */
} stdasm_data;

#define STDASM_INLET1     3  /* Inlet1, CPU Temperature Sensor */
#define STDASM_EXHAUST1   4  /* Exhaust1, Power, sdram1 Temperature Sensor */
#define STDASM_EXHAUST2   5  /* Exhaust2, sdram2 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 diode temperature. Please refer to `/usr/platform/SUNw,Netra-CP2300/include/sys/ctasm.h` if you want to add support for other 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 <ctasm.h>/* lives in /usr/platform/SUNW,Netra-CP2300/include/sys
       directory */

/* Right now, this application monitors the CPU temperature only, if you want
       to add support for the other sensors, you have to duplicate 12 lines
```

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

```
        in the ProcessAllTemps routine. Also refer the ctasm.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;
        stdasm_data SAData;
        struct strioctl sioc;

        SAData.sensorNum = STDASM_INLET1; /* 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);
```

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

```
            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 sensors STDASM_EXHAUST1,
                STDASM_EXHAUST2 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/scclone", O_RDWR)) < 0) {  /* open the SC device
*/
            printf("Unable to open device /dev/sc; errno=%d\n", errno);
            exit(1);
        }
        /* Push the 'ASM' driver module */
        Result = ioctl(AsmFd, I_PUSH, "ctasm");
        if (Result == -1) {
            printf("I_PUSH ctasm failed RetValue %d\n", errno);
            exit(3);
        }
        ProcessAllTemps(AsmFd, ScPort);
}
```

Note – The `ctasm.h` header file is located in the `/usr/platform/SUNW,Netra-CP2300/include/sys` directory.

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 CP2300 cPSB board. 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).

In order to use the LED function, a SPARC V9 64-bit C library and the `led.h` file are required. The library and the file are available in the `SUNWcp23u` package. The Application Programming Interface (API) for the user is documented in the `led.h` file. See "Files and Packages Required to Support the Alarm/User LED" on page 65 for more information.

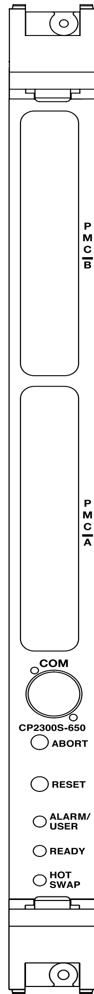


FIGURE 4-1 Illustration of a Typical Netra CP2300 cPSB Board Front Panel Showing the Alarm/User LED

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:

- `SUNWcp23u:` SPARC V9 64-bit C library `libcp2300.so.1` available at:

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

- `SUNWcp23u:` LED `include` file available at:

  `/usr/platform/${PLATFORM}/include/sys/`

Ensure that the following driver is also there, as needed:

- `SUNWcp23x.u`: 64-bit sc_nct driver available at:

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

A typical example of ${PLATFORM} is `SUNW,Netra-CP2300` for the Netra CP2300 board. An example for the library directory is:

```
/usr/platform/SUNW,Netra-CP2300 /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 sc_nct 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.

Application Programming Interface (API)

CODE EXAMPLE 4-1 Application Programming Interface for the Netra CP2300 Board

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

/* LEDS */
#define BLUE_LED          0x0
#define HOTSWAP_LED       BLUE_LED
#define PLD_GREEN_LED     0x04
#define GREEN_LED         PLD_GREEN_LED
#define AMBER_LED         0x08

/* LED COMMANDS */
#define LED_OFF        0x00
#define LED_ON         0x01
#define LED_BLINK_SLOW 0x02
#define LED_BLINK_FAST 0x03

/* ERROR CODES */
#define ESEQUENCE      200  /* portnum mismatch */
#define ECMDCOMP       201  /* non-zero command completion */
#define ECMDCODE       202  /* smc command mismatch */
```

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

TABLE 4-1 Supported LED and Command Combinations for the Netra CP2300 Board

Color of LED	LED_OFF	LED_ON	LED_BLINK_SLOW	LED_BLINK_FAST
BLUE_LED	Yes	Yes	No	No
GREEN_LED	Yes	Yes	Yes	Yes
AMBER_LED	Yes	Yes	No	No

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 65. Specify 64-bit binaries by setting the `-xarch=v9` and `-D__sparcv9` compiler flags.

For example:

```
-xCC -xarch=v9 -D__sparcv9 -I/usr/platform/SUNW,Netra-CP2300/include/
```

Note – Type the above command all on one line.

Link

To create a link to the library named (`libcp2300.so.1`) listed in “Files and Packages Required to Support the Alarm/User LED” on page 65, use the linker flag **`-L`** command.

For example:

```
-L /usr/platform/SUNW,Netra-CP2300/lib
```

Sample Application Program

This section presents a sample test.c application to turn the LED on, off, and blink.

CODE EXAMPLE 4-2 Sample LED Application Program

```
#include       <stdio.h>
#include       <sys/led.h>

main()
{
      /* blue on, rest off */
      printf("\n\nTesting Blue led ON, rest off\n");
      fflush(stdout);
      printf("BLUE_LED on returned %d\n", led(BLUE_LED, LED_ON));
      fflush(stdout);
      sleep(4);
      printf("GREEN_LED off returned %d\n", led(GREEN_LED, LED_OFF));
      fflush(stdout);
      sleep(4);
      printf("AMBER_LED off returned %d\n", led(AMBER_LED, LED_OFF));
      fflush(stdout);
      sleep(4);

      /* all lights on, and green blinking fast */
      printf("\n\nTesting all led's on and green blinking fast\n");
      fflush(std out);
      printf("BLUE_LED on returned %d\n", led(BLUE_LED, LED_ON));
      fflush(stdout);
      sleep(4);
      printf("AMBER_LED on returned %d\n", led(AMBER_LED, LED_ON));
      fflush(stdout);
      sleep(4);
      printf("GREEN_LED blink returned %d\n", led(GREEN_LED, LED_BLINK_FAST));
      fflush(stdout);
      sleep(4);
}

cc -xCC -xarch=v9 -D__sparcv9 \
        -I /usr/platform/SUNW,Netra-CP2300/include \
        -L /usr/platform/SUNW,Netra-CP2300/lib
        -l cp2300 \
        -o test \
        test.c
```

Index

A

address range, 15
ASM, 35
  application block diagram, 37
  application program, 57
  functional block diagram, 44
  polling rate, 58
  temperature monitoring, 46 to 47

D

device node, 17
diag-switch?, 13
documentation, xvi
drift rate, 39
dropins, 15

E

EACCESS, 20
EBUSY, 8
ECANCELLED, 20
EFAULT, 8, 20
EINVAL, 8, 20
ENOMEM, 20
env-monitor parameter, 38
ENXIO, 8, 20
execute-smc-cmd command, 52

I

Intelligent Platform Management Interface (IPMI), 51
IOCTL
  and ASM, 39
  and user flash, 18 to 27
  and watchdog timer, 7 to 11

K

keyboard controller style (KSC), 13

L

LED, alarm/user, 63

N

nonvolatile memory, 13

O

OpenBoot PROM
  and ASM, 38, 47
  and user flash, 17
  and watchdog timer, 11, 12
output buffer full (OBF), 13

P

PROM chips, 15
PROM information structure, 19

R

RTOS, 15

S

show-sensors command, 51
sleep system call, 39
SMC, 1, 35
SMC switch, 15

T

temperature, 38, 41, 45 to 51

U

user data storage, 15
user flash
  application program, 27
  device, 18
  device files, 17
  driver, 15
  header file, 17
  interface structure, 19
  node properties, 17

V

voltage controller, 45

W

watchdog timer, 1
watchdog-enable?, 11
watchdog-timeout?, 11
WD1, 2
WD2, 2
WIOCGSTAT, 7
WIOCSTART, 7
WIOCSTOP, 7