High Availability Platform User's Guide

Release 2.0

DOC-0398-01

Contents

Preface

The *High Availability Platform User's Guide* contains information about LynuxWorks' High Availability Platform (HAP), including installation information, diagnostic tools, programmer's reference and supported hardware.

This manual assumes a basic understanding of Hot Swap and Domain Failover principles. It also assumes a basic familiarity with using, administering, and programming in a UNIX environment.

For More Information

For more information on the features of LynxOS, refer to the following printed and online documentation.

- *HAP Release Notes*

  This document contains late-breaking information about this release, including additional installation notes.

- *LynxOS Installation Guide*

  This manual describes the initial installation and configuration of LynxOS and the X Windows System.

- *LynxOS User's Guide*

  This document contains information about basic system administration and kernel-level specifics of LynxOS. It contains a "Getting Started" chapter and covers a range of topics, including tuning system performance and creating kernel images for embedded applications.

- *Writing Device Drivers for LynxOS*

  This document describes writing device drivers for the LynxOS kernel.

- *LynxOS Hardware Support Guide*

  This document describes the hardware supported by LynxOS. Hardware supported by HAP 2.0 is detailed in the HAP Release Notes.

- Online information

  The complete LynxOS documentation set is available on the Documentation CD-ROM. Books are provided in both HTML and PDF formats.

  Updates to these documents are available online at the LynuxWorks Website: `http://www.lynuxworks.com`.

  Additional information about commands and utilities is provided online with the **`man`** command. For example, to find information about the GNU gcc compiler, use the following syntax:

  ```
  man gcc
  ```

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to file names and commands are case sensitive and should be typed accurately.

Kind of Text	**Examples**
Body text; *italicized* for emphasis, new terms, and book titles	Refer to the *LynxOS User's Guide*.

Environment variables, file names, functions, methods, options, parameter names, path names, commands, and computer data	`ls`
	`-l`
	`myprog.c`
	`/dev/null`

Commands that need to be highlighted within body text, or commands that must be typed as is by the user are **`bolded`**.	`login:` **`myname`**
	`#` **`cd /usr/home`**

Text that represents a variable, such as a file name or a value that must be entered by the user	`cat` *`filename`*
	`mv` *`file1 file2`*

Kind of Text	**Examples**

Blocks of text that appear on the display screen after entering instructions or commands	`Loading file /tftpboot/shell.kdi`
	`into 0x4000`
	`.....................`
	`File loaded. Size is 1314816`
	`Copyright 2000 LynuxWorks, Inc.`
	`All rights reserved.`
	
	`LynxOS (ppc) created Mon Jul 17`
	`17:50:22 GMT 2000`
	`user name:`

Keyboard options, button names, and menu sequences	**Enter**, **Ctrl-C**

Special Notes

The following notations highlight any key points and cautionary notes that may appear in this manual.

NOTE: These callouts note important or useful points in the text.

CAUTION! Used for situations that present minor hazards that may interfere with or threaten equipment/performance.

Technical Support

LynuxWorks Technical Support is available Monday through Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (Europe).

The LynuxWorks World Wide Web home page provides additional information about our products.

LynuxWorks U.S. Headquarters

Internet: `support@lnxw.com`
Phone: (408) 979-3940
Fax: (408) 979-3945

LynuxWorks Europe

Internet: `tech_europe@lnxw.com`
Phone: (+33) 1 30 85 06 00
Fax: (+33) 1 30 85 06 06

World Wide Web

`http://www.lynuxworks.com`

CHAPTER 1

Introduction

High Availability Platform for LynxOS

The LynxOS High Availability Platform (HAP) 2.0 allows users to create highly available system and application services. HAP 2.0 provides support for both Hot Swap and Warm Domain Switchover applications.

This *High Availability Platform User's Guide* describes:

- Introduction to High Availability
- Supported Hardware
- Installation
- High Availability Tools
- Writing Applications for HAP
- Basic Terminology

Product Overview

The High Availability Platform for LynxOS provides the following basic functionality:

- Response to ENUM signals
- Dynamically loads or unloads device drivers
- Appends or prunes the Device Resource Manager (DRM) resource tree, which represents the system topology

- Response to application commands to transition domain states in a multiple domain system

In addition, the following major features are available in HAP 2.0:

- Full Hot Swap application support (both Hot Insertion and Hot Extraction) for Motorola PowerPC and Intel x86 Processors in standard Hot Swap-capable chassis (including Motorola 82xx, 22xx and 12xx families). HAP 2.0 informs applications of all connection states if they use the HAP 2.0 driver model.
- Warm Domain Switchover application support; HAP 2.0 allows applications to perform Warm Domain Switchover, provided they use the HAP 2.0 driver model. For more information, please see “Warm Domain Switchover” on page 14.
- Tool utilities and libraries to assist manual intervention in diagnosing and facilitating Hot Swap and Warm Domain Switchover events; for more information, please see Chapter 3, “High Availability Tools”

This release runs on the Motorola MCP750 PowerPC and Motorola CPV5350 Intel Pentium CompactPCI system controllers in 82xx, 22xx, and 12xx chassis.

About High Availability

High Availability is the term used to describe systems built with reliable components, redundant elements, and the means to communicate system and application states.

Problem detection and system recovery in *High Availability* systems can be either manual or automatic. Manual detection and recovery requires an operator to detect or anticipate a failure and replace the failing component. Automatic detection and recovery is controlled by system hardware, software extensions, and in some cases, the applications themselves.

It is possible to build systems with commercial hardware components and Open System software extensions. By providing redundant elements and switchover applications, the system can remain in service for a high percentage of time. The components involved are assumed to be only moderately reliable, so there must be a heavy reliance on software and manual intervention to anticipate and handle failures.

System Availability

Availability is defined as the duration of time a computer system provides services to applications in proportion to the duration of time the system is unavailable. A highly available system provides services to applications most of the time.

The downtime of a computer system can be predicted from the *mean time to failure (MTTF)* of the components and the *mean time to repair (MTTR)* those components. The following is the formula for Computing System Availability. In a system of *n* components, each having a mean time to failure and a mean time to repair, the availability is calculated as such:

$$\text{Availability} = \frac{\sum_{i=1}^{n} \text{MTTF}_i}{\sum_{i=1}^{n} (\text{MTTF}_i + \text{MTTR}_i)}$$

For example, a computer system of five components has these MTTF and MTTR specifications:

Table 1-1: MTTF/MTTR Specifications

Component	MTTF	MTTR
Chassis	1000 days	2 days
Processor	500 days	1 day
Board A	800 days	1 day
Board B	300 days	1 day
Board C	750 days	2 days

Component availability for this configuration is calculated as:

$$\text{Availability} = \frac{(1000 + 500 + 800 + 300 + 750)}{(1002 + 501 + 801 + 301 + 752)}$$

In this example, the system is expected to be up 99.79% of the time. Some computer applications require an availability of 99.999%. This is referred to as *Five-Nines Availability.*

Availability Impacting Events

In a system that supports application intervention on a live system, there are two different types of events that require the platform to change its component topology:

- An operator decides that a device in the system needs to be exchanged
- A device fails

The first event is characterized as a maintenance operation, and the second as a fault. Support for these different modes are discussed below.

Hot Swap

In this and following sections, all references are to CompactPCI system architecture.

Hot Swap is described in terms of three processes:

- *Physical Connection Process*

  Includes:

  - Hot Insertion (installing a board in a live system)
  - Hot Extraction (removing a board from a live system)

- *Hardware Connection Process*

  Describes the electrical connection and disconnection of hardware to a live system

- *Software Connection Process*

  Describes the connection and disconnection of the software layer(s) to a live system

These processes are a set of states and transitions. These states depend on the transitions and prior states for their characteristics. The following figure details these states and transitions:

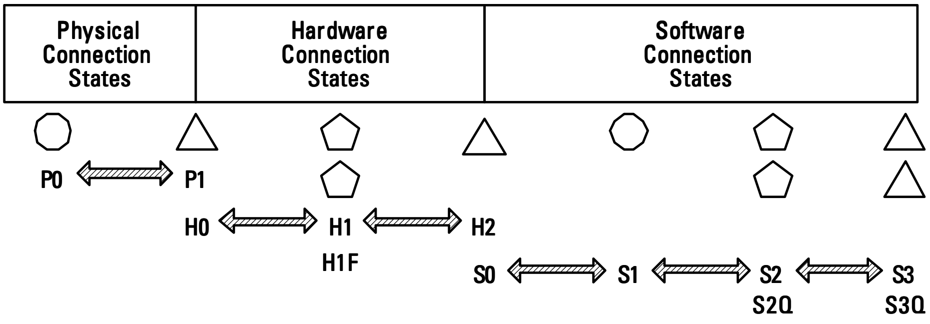

Physical Connection States
Hardware Connection States
Software Connection States
P0
P1
H0
H1
H1F
H2
S0
S1
S2
S2Q
S3
S3Q

Figure 1-1: Hot Swap States

The following are descriptions of Hot Swap States in Figure 1-1:

- *P0* –The board is physically separate from the system.
- *P1 / H0* – The board is fully seated, but not powered and not active on the PCI bus. Note that at this point, the physical layer is in P1 and the hardware layer is in H0.
- *H1* – The board is powered up and is sufficiently initialized to connect to the PCI bus.
- *H1F* – The board is commanded to power up and initialize and has failed, or the board detected an error and disconnects itself from the PCI bus. The board is not suitable for connection to the PCI bus.
- *H2 / S0* – The board is powered and enabled for access by the PCI bus in configuration space only. The board's configuration space is not yet initialized. Here, the hardware layer is in H2 and the software layer is in S0.
- *S1* – The board is configured by the system.
- *S2* – The necessary supporting software (drivers, etc.) are loaded. The board is ready for use by the operating system and applications, but no operations involving the board are active.
- *S2Q* – This state is the same as state S2, but no new operations are allowed to start. The board is quiesced.

- *S3* – The board is engaged in software operations.
- *S3Q* – The software is completing current operations, but is not allowed to start new ones.

The following are components of a Hot Swap system in the CompactPCI environment.

- *Board*

  A circuit board in the system (other than the system host)
- *System Host*

  The central resource that provides configuration of the CompactPCI bus; it may also provide arbitration and clocking for the CompactPCI bus.
- *Platform*

  The platform providing infrastructure for the boards; this includes (but is not limited to) the backplane, system host, cooling, and power supplies.

The diagram below depicts the functional elements and their relationship in Hot Swap architecture. Bold areas indicate the unique components for Hot Swap that are a superset of conventional CompactPCI systems.

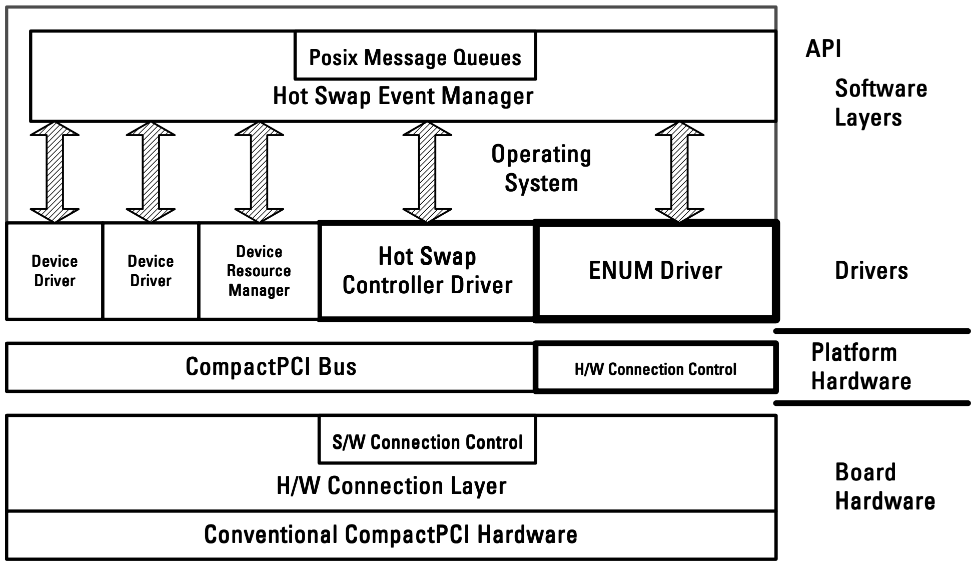


Figure 1-2: CompactPCI Hot Swap Architecture

At the lowest level, a Hot Swap board contains a *Hardware Connection Layer*, which supports the *Physical Connection Process* to the CompactPCI bus. Controlled by the platform, the *Hardware Connection Layer* performs the *Hardware Connection Process*. This layer can optionally contain special hardware resources to facilitate the *Software Connection Process*.

The Hot Swap architecture facilitates dynamic configuration of a users's system by interacting with the OS and user supplied device drivers.

System Models

There are four different degrees of Hot Swap capability:

- Non-Hot Swap

  Systems that do not have Hot Swap capabilities
- Basic Hot Swap

  Systems meeting the basic requirements for Hot Swap
- Full Hot Swap

  Systems utilizing the features of Full Hot Swap boards
- High Availability

  Systems utilizing features of High Availability platforms for greater hardware control

There are several ways for applications to implement High Availability. Depending on the system configuration, it is not always advisable to take advantage of High Availability capabilities. In the following table, various possibilities are shown in order of increasing complexity.

Table 1-2: System Models

System Type	Hardware Connection	Software Connection
Basic Hot Swap	Automatic in hardware	Manually by operator

Table 1-2: System Models

System Type	Hardware Connection	Software Connection
Full Hot Swap	Automatic in hardware	Controlled automatically by software
High Availability Hot Swap	Controlled by software	Controlled automatically by software

Fault detection and recovery requires either the Full Hot Swap or High Availability models. The Manual operator maintenance function is the only capability supported by Basic Hot Swap.

NOTE: *HAP 2.0 supports the High Availability Hot Swap model.*

Basic Hot Swap System

Basic Hot Swap consists of board hardware with the Hot Swap additions to the Hardware Connection Layer, and the Event Management Service. The Basic Hot Swap Model does not support software access to the Hardware and Software Connection Processes.

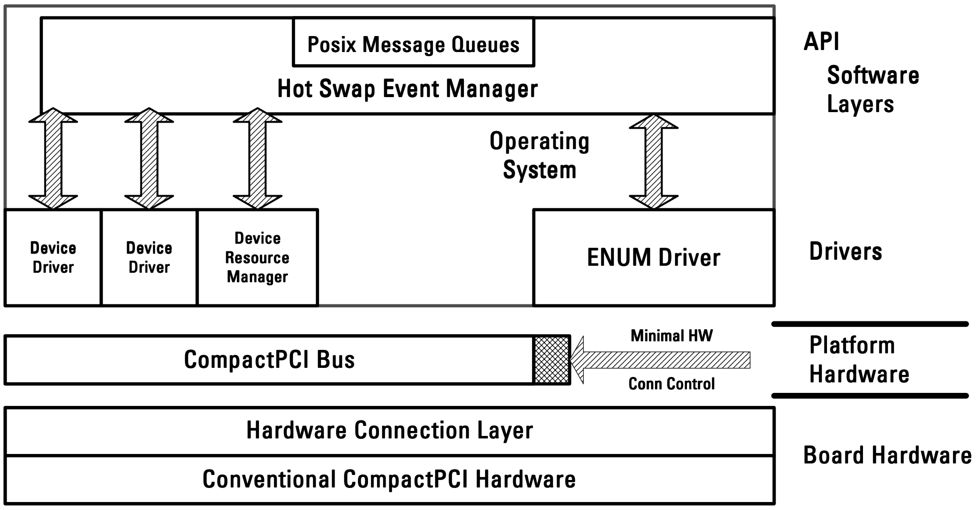


Figure 1-3: Basic Hot Swap System

The Hardware Connection Layer automatically brings a board to the H2 / S0 state upon insertion (see Figure 1-3). The Event Management Service interacts with the operating system to load drivers and configures the operating system to enable applications to access the new hardware. The Event Management Service performs the reverse function on an attempt to extract a board.

All Hot Swap actions must be initiated by an operator and performed in the correct sequence for proper system operation.

Full Hot Swap System

In addition to the Basic Hot Swap functionality, Full Hot Swap provides resources for controlling the Software Connection Process. These resources include Software Connection Control resources on the board, and the ability for the operating system to dynamically load drivers and configure new devices on a running system.

The Software Connection Control function on the board provides an electrical signal (ENUM) that notifies the system host of a service request, as well as other board state transitions. A Hot Swap board is expected to have a Hot Swap Switch. When pulled, the switch notifies the system that an extraction is desired. When set, the switch notifies the system that a new board has been inserted. There is also an LED indicator for the state of the board; if illuminated, it is safe to extract the board. This mechanism makes it possible to perform the necessary steps in the software to allow access to the board.

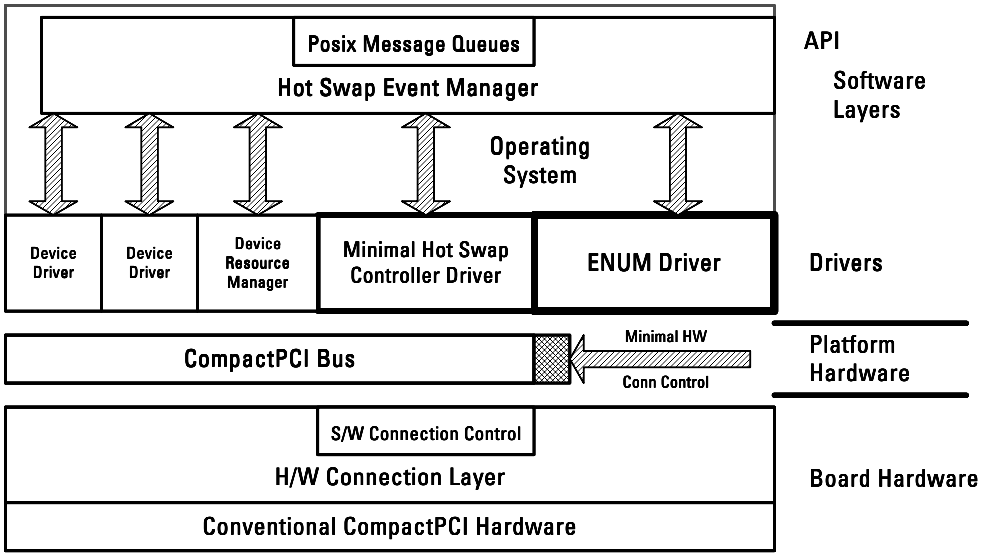


Figure 1-4: Full Hot Swap System

Full Hot Swap boards drive the ENUM signal to the system host to indicate a service request. The system host responds to the ENUM signal by adding software drivers for newly inserted boards, or unloading drivers for boards to be extracted. An operator action indicating a need to extract a board allows the system to quiesce and unload any drivers associated with the board. The board can then be safely removed from the system.

High Availability System

High Availability requires even more control than simply handling insertion and extraction. HA systems are able to control the Hardware Connection Process. This is accomplished by adding support for additional electrical signals, namely PRESENT and HEALTHY.

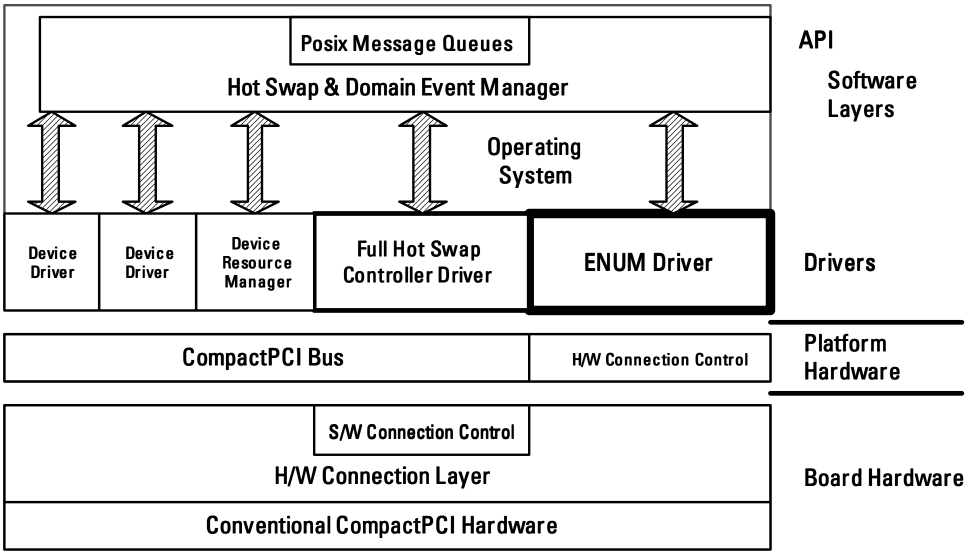


Figure 1-5: High Availability Hot Swap Mode

In this model, the system performs additional services to ensure more than just the inserted or extracted state of the board. In particular, prior to applying power, it detects the board's presence and whether or not the board is reporting that it is working properly.

High Availability Frameworks

For High Availability, a system must be able to be reconfigured (in the event of maintenance requirements or active faults) with minimal system impact. While reconfiguration does have some impact, it must still meet the given availability specification for Highly Available systems.

In order to design a system that accommodates a number of different architectures for High Availability, frameworks that provide High Availability benefits are required. As a rule, more specialized (and hence weaker) frameworks provide the maximum benefit.

For High Availability systems, there are two basic models for which the common framework is, as yet, undeveloped: The first model for High Availability is network based (what may be called interchassis redundancy). In this model, there are two or more duplications of complete platforms, and system hosts processors with the full complement of the necessary boards. Application software monitors the health and reliability of each platform and shifts responsibilities accordingly.

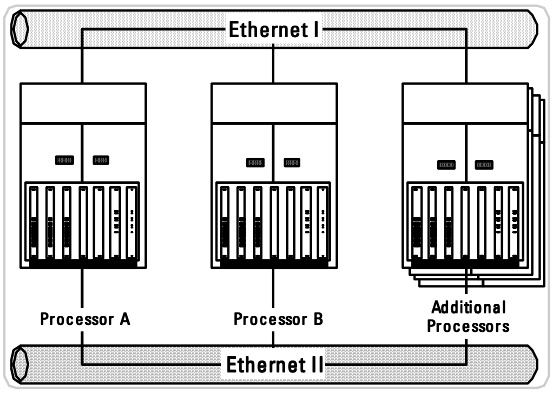


Figure 1-6: Network-Based Redundancy

In this case, availability is guaranteed by ensuring that one complete system is functional at all times using an ethernet network to share application and device states. In a more sophisticated setting, three or more systems running the same applications use an election mechanism to determine the most likely correct result.

The second model is a common backplane (or intrachassis redundancy). One such backplane is Compact PCI. Compact PCI is a commercial bus, backplane, and card specification with many features making it ideal for High Availability systems. The Compact PCI bus is electrically and logically the same as the motherboard based PCI bus. Interface chips, CPU, and I/O card designs are inexpensive and readily available. Compact PCI supports Hot Swap of I/O cards and Hot Swap of system controllers.

Like PCI, Compact PCI supports the dynamic assignment of I/O space, interrupt lines, and peripheral memory space. Boards are identified by the physical slot they occupy and are not operational until they are initialized. This is in contrast with the VME bus, where boards are not identified by their slot, but rather I/O space is configured with jumpers on the I/O boards themselves.

Compact PCI extends the PCI specification by adding an enumeration signal (ENUM#) and status bits to indicate insertion (INS) and extraction (EXT). These additional signals, along with support for applying power before any signal lines make contact, make it possible to Hot Swap Compact PCI boards. Compact PCI boards can be extracted and inserted while the system is on and operational, allowing replacement of a failed board without interrupting service. Also, additional boards can be added to the computer while it is operational, to upgrade the system without interrupting service. A PCI bus must be re-enumerated to make room for new boards. This changes the hardware topology of the system. It is not a simple task, because a board may contain more than one PCI device and even may contain bridge chips that introduce new PCI buses to the system.

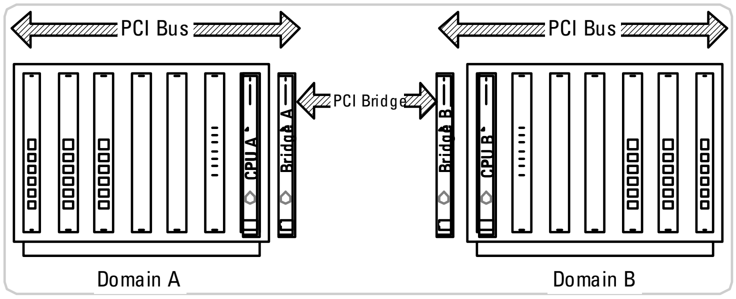


Figure 1-7: Compact PCI Bus Conceptual Diagram

Due to electrical characteristics, only eight devices can be plugged into a single Compact PCI bus. Through the use of PCI bridge chips, a computer can have multiple Compact PCI buses, and extend the number of PCI devices the CPU can access. Although Compact PCI allows only for one system board, Compact PCI buses (each with its own system board) can be connected through bus bridges. One way to make use of a bus bridge is to create a dual Compact PCI system with one CPU card/system controller for each half of the system. In case of failure of an I/O card, a redundant card plugged into the other Compact PCI bus can be used. In the case of a CPU/system card failure, the PCI bridge can be used to allow the working CPU card to take over control of the I/O cards that the failed. This allows for great flexibility in reconfiguring the system.

Today, the Compact PCI specification leaves the definition of the second set of connectors on 6U Compact PCI boards to the vendor. These can be used to bring signals to I/O transition boards or for other auxiliary buses. These signals can also be used for additional Hot Swap and HA support.

Domain Switchover

In Domain Switchover, an active domain on one side may have its application and driver states moved to the second domain. As with individual cards, a domain can be transitioned because of an operator-initiated act, or as a direct result of the failure of the system host. As with cards, the application may be responsible for controlling the transition.

Warm Domain Switchover

HAP 2.0 includes limited Domain Switchover support. At system startup, both Domains enter a Cold Stand-By state. Upon application commands, HAP will:

- change from Cold Stand-By state to Active state.
- change from Cold Stand-By state to Warm Stand-By state.
- change from Active state to Warm Stand-By state.
- change from Warm Stand-By to Active state.

Driver Model

In order for HAP 2.0 to respond correctly to the above, each driver of interest to a Warm Domain Switchover application must implement the following states and provide scripts to perform the necessary driver state transitions as shown in the following figure:

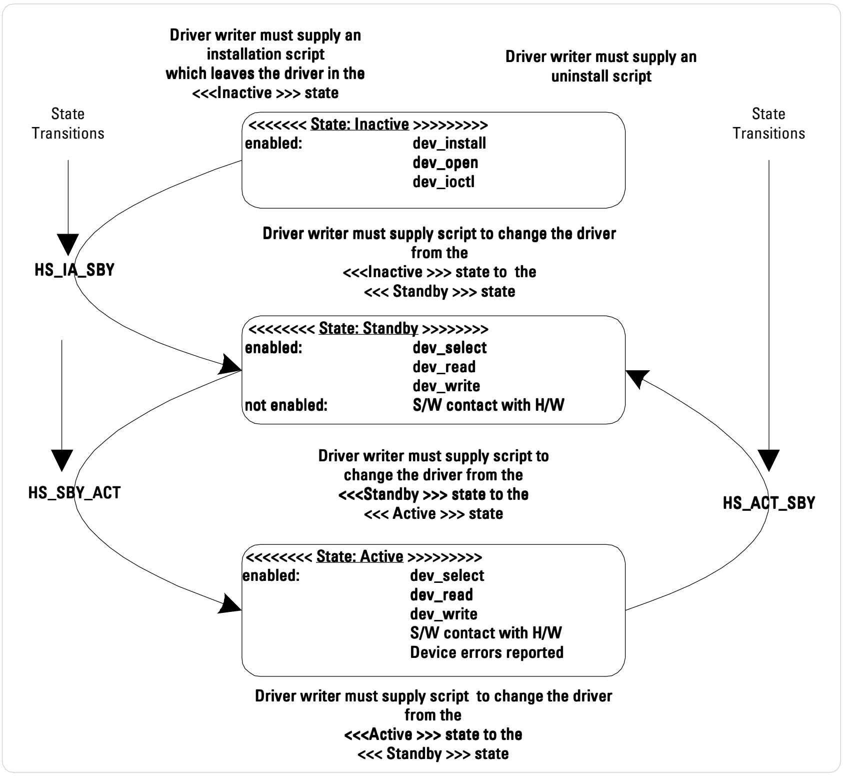

Driver writer must supply an installation script which leaves the driver in the <<<Inactive >>> state
Driver writer must supply an uninstall script
State Transitions
State Transitions
<<<<<<< State: Inactive >>>>>>>>>
enabled: dev_install dev_open dev_ioctl
Driver writer must supply script to change the driver from the <<<Inactive >>> state to the <<< Standby >>> state
HS_IA_SBY
<<<<<<<< State: Standby >>>>>>>>
enabled: dev_select dev_read dev_write
not enabled: S/W contact with H/W
Driver writer must supply script to change the driver from the <<<Standby >>> state to the <<< Active >>> state
HS_SBY_ACT
HS_ACT_SBY
<<<<<<<< State: Active >>>>>>>>>
enabled: dev_select dev_read dev_write S/W contact with H/W Device errors reported
Driver writer must supply script to change the driver from the <<<Active >>> state to the <<< Standby >>> state

Figure 1-8: Driver Model

CHAPTER 2

Installing the High Availability Platform

This chapter describes the steps and prerequisites required to install the High Availability Platform (HAP) 2.0 package.

System Requirements

HAP 2.0 must be installed on a Compact PCI system with either a PowerPC or Intel x86 system controller.

Installing the High Availability Platform

Follow these steps to install HAP 2.0:

1. Mount the installation CD-ROM media to an available mount point on the system. For example,

   ```
   mount /dev/cdrom /mnt/cdrom
   ```

2. Change directory to the mount point. For example,

   ```
   cd /mnt/cdrom
   ```

3. Install HAP 2.0 by executing this script:

   ```
   sh> Install.HAP
   ```

   Detailed configuration and system changes made by this script are described in “Install.HAP Configuration Specifications” on page 19.

4. When prompted, specify the HAP installation location. This directory must have a complete kernel build environment in a `/sys` tree. The installation does not use `ENV_PREFIX`.
5. When prompted, supply the correct chassis type for your system. Several configurations are installed on the system. In addition to different configuration files, a single domain system (CPV12xx or CPX22xx) must be configured to run the Hot Swap Event Manager from `ha_sim`. For more information, see See "ha_sim - HA System Initialization Manager" on page 24.. For a multiple domain system (CPX 82xx) `stmd` is configured to run by default.
6. When prompted, select to reconfigure the driver library automatically or manually. If manual is selected, the driver library must be updated by the user, and the kernel rebuilt before HAP will function.
7. After the driver library is rebuilt, the system must be restarted as follows:

```
reboot -aN
```

Previous kernel, nodetab, device and driver libraries are saved in this directory: `$INSTALL_ROOT/usr/local/kits/HAP`.

NOTE: Old kernel and device libraries can be restored from this directory should the system need to be restored to its previous state. This directory should not be deleted.

The following scripts and files are saved in `/usr/local/kits/HAP:`

- `Install.HAP`
- `Uninstall.HAP`
- `binary.filelist`
- `Distribution_HAP20.tar.gz`

Uninstalling

To uninstall, or revert to a previous installation of HAP, follow these instructions:

1. Change to the HAP script directory as follows:

```
cd /usr/local/kits/HAP
```

2. Execute the uninstall script as follows:

```
sh> Uninstall.HAP
```

Post-Installation Tasks

If not already completed by the installation script, device drivers must be added and the kernel rebuilt before HAP can be enabled.

Install.HAP Configuration Specifications

When the installation script runs, these changes are made to the system:

1. A directory is created for the installation scripts and working files in: `$INSTALL_ROOT/usr/local/kits/HAP.`

   If this directory already exists, it is renamed: `$INSTALL_ROOT/usr/local/kits/HAP.prev.`

2. The installation files are copied to `$INSTALL_ROOT/usr/local/kits/HAP.`

   These files include:

   - `Distribution_HAP20.tar.gz`
   - `binary.filelist`
   - `Install.HAP`
   - `Uninstall.HAP`

3. The following files are saved in `$INSTALL_ROOT/usr/local/kits/HAP/HAP_pre_install.tar.gz:`

   - `$INSTALL_ROOT/lynx.os`
   - `$INSTALL_ROOT/etc/nodetab`
   - `$INSTALL_ROOT/sys/lib/libdevices_cpci_<`*`platform`*`>.a`
   - `$INSTALL_ROOT/sys/lib/libdevices_cpci_<`*`platform`*`>_d.a`
   - `$INSTALL_ROOT/sys/lib/libdevices_cpci_<`*`platform`*`>_uk.a`
   - `$INSTALL_ROOT/sys/lib/libdrivers_cpci_<platform>.a`
   - `$INSTALL_ROOT/sys/lib/libdrivers_cpci_<`*`platform`*`>_d.a`
   - `$INSTALL_ROOT/sys/lib/libdrivers_cpci_<`*`platform`*`>_uk.a`
   - any file in `binary.filelist` if it exists

   Where `<`*`platform`*`>` is `drm` for PowerPC and `x86` for Intel.

4. The following files, if they exist, are saved in the `$INSTALL_ROOT/usr/local/kits/HAP/save` directory, so they can be restored after the HAP 2.0 Installation:
   - `$INSTALL_ROOT/etc/hasw/ha_sim.con`
   - `$INSTALL_ROOT/sys/cfg/enum.cfg`
   - `$INSTALL_ROOT/etc/hasw/hsem.conf`
   - `$INSTALL_ROOT/etc/hasw/tm.conf`
5. The distribution is copied to the installation location. The files copied are those listed in `binary.filelist`. The following directories are created if they do not already exist:
   - `$INSTALL_ROOT/usr/sbin/hasw`
   - `$INSTALL_ROOT/etc/hasw`
   - `$INSTALL_ROOT/usr/include/hasw`
   - `$INSTALL_ROOT/usr/lib/hasw`
   - `$INSTALL_ROOT/usr/man/cat1`
   - `$INSTALL_ROOT/usr/man/cat5`
   - `$INSTALL_ROOT/sys/drivers.cpci_`<*platform*>
   - `$INSTALL_ROOT/sys/devices.cpci_`<*platform*>
   - `$INSTALL_ROOT/usr/src/hasw/util`

   where <*platform*> is `drm` for PowerPC and `x86` for Intel.
6. Chassis type configuration files (`tm`, `hsem`, and `ha_sim`) are added to the system.
7. The files saved in step 4 are restored to their original location.
8. `$INSTALL_ROOT/sys/bsp.cpci_`<*platform*>/`CONFIG.TBL` is checked to ensure that the `I:enum.cfg` is present and uncommented.
9. The driver library is rebuilt if the user requests it.

CHAPTER 3 High Availability Tools

Introduction

The High Availability Platform 2.0 includes a set of tools that provide command line interaction with Compact PCI and board hardware. This chapter details these commands.

cpxload - cpxunload

Scripts for loading and unloading the Hot Swap Controller Driver.

cpxtool

An interactive tool for accessing Hot Swap Controller Driver statistics.

drm_stat - Display all device nodes in a system

Synopsis

```
drm_stat
```

Description

This command uses no arguments. A status of all the devices in the DRM tree and Internal Address Allocation information is displayed.

Table 0-1: Sample Session

```
bash# drm_stat
Device Resource Manager
Device ID = 1
Vendor ID = 1
Primary Buslayer ID = -1
Secondary Buslayer ID = 0
Node type = 5
State = 5
Interrupt Controller = -1
Interrupt Line = -1
Device ID = 2
Vendor ID = 1
Primary Buslayer ID = 0
Secondary Buslayer ID = 1
Node type = 5
State = 4
Interrupt Controller = -1
Interrupt Line = -1
Device ID = 4801
Vendor ID = 1057
Primary Buslayer ID = 1
Secondary Buslayer ID = 0
Node type = 10
State = 5
Interrupt Controller = 0
Interrupt Line = -5
BusNo = 0
DevNo = 0
FuncNo = 0
0: Vaddr = 0, Paddr = 80040000, Baddr = 40000, Size = 40000, Al = 40000
1: Vaddr = c0000000, Paddr = c2000000, Baddr = 2000000, Size = 40000, Al =
40000
2: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
3: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
4: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
5: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
```

Table 0-1: Sample Session (Continued)

```
Device ID = 586
Vendor ID = 1106
Primary Buslayer ID = 1
Secondary Buslayer ID = 2
Node type = 6
State = 5
Interrupt Controller = 0
Interrupt Line = 0

Device ID = 571
Vendor ID = 1106
Primary Buslayer ID = 1
Secondary Buslayer ID = 0
Node type = 10
State = 4
Interrupt Controller = 0
Interrupt Line = 0
BusNo = 0
DevNo = 11
FuncNo = 1
0: Vaddr = 0, Paddr = 80004000, Baddr = 4000, Size = 8, Al = 1000
1: Vaddr = 0, Paddr = 80005000, Baddr = 5000, Size = 4, Al = 1000
2: Vaddr = 0, Paddr = 80006000, Baddr = 6000, Size = 8, Al = 1000
3: Vaddr = 0, Paddr = 80007000, Baddr = 7000, Size = 4, Al = 1000
4: Vaddr = 0, Paddr = 80008000, Baddr = 8000, Size = 10, Al = 1000
5: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
Device ID = 9
Vendor ID = 1011
Primary Buslayer ID = 1
Secondary Buslayer ID = 0
Node type = 10
State = 5
Interrupt Controller = 0
Interrupt Line = 2
BusNo = 0
DevNo = 14
FuncNo = 0
0: Vaddr = 0, Paddr = 80009000, Baddr = 9000, Size = 80, Al = 1000
1: Vaddr = c0040000, Paddr = c2040000, Baddr = 2040000, Size = 80, Al = 1000
2: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
3: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
4: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
5: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
```

Table 0-1: Sample Session (Continued)

```
Device ID = 3
Vendor ID = 1000
Primary Buslayer ID = 1
Secondary Buslayer ID = 0
Node type = 10
State = 5
Interrupt Controller = 0
Interrupt Line = 3
BusNo = 0
DevNo = 16
FuncNo = 0
0: Vaddr = 0, Paddr = 8000a000, Baddr = a000, Size = 100, Al = 1000
1: Vaddr = c0041000, Paddr = c2041000, Baddr = 2041000, Size = 100, Al = 1000
2: Vaddr = 0, Paddr = c2042000, Baddr = 2042000, Size = 1000, Al = 1000
3: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
4: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
5: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0

Device ID = 26
Vendor ID = 1011
Primary Buslayer ID = 1
Secondary Buslayer ID = 1
Node type = 6
State = 4
Interrupt Controller = 0
Interrupt Line = 0
BusNo = 0
DevNo = 20
FuncNo = 0
SecBusNo = 1
SubBusNo = 41
0: Vaddr = 0, Paddr = 80080000, Baddr = 80000, Size = 1200000, Al = 1000
1: Vaddr = 0, Paddr = c2100000, Baddr = 2100000, Size = 6000000, Al = 100000
2: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
3: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
4: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
5: Vaddr = 0, Paddr = 0, Baddr = 0, Size = 0, Al = 0
```

ha_sim - HA System Initialization Manager

Synopsis

```
ha_sim -i | -s [-d -n <mddPublicQueueName> -q
<PrivateQueueName> -f <ConfigFileName>]
```

Description

ha_sim is responsible for starting (at system init time) HA processes and stopping (at system shut down time) HA sub-systems. ha_sim is started with a command line argument indicating the mode as *init* or *shutdown*. ha_sim starts during system initialization from the /bin/rc file run by LynxOS init. When started, ha_sim reads a configuration file to determine the HA processes or sub-systems to start or to stop. ha_sim can also be started at any time from the command line.

Dependencies

To receive the SHUTDOWN message, processes must register with hsem/stmd to receive notification of these messages:

```
type  = SIM
class = SYS_CLASS_1
```

There is no response to the SHUTDOWN message.

The SIM, SYS_CLASS_1 messages are not targeted to a single named process, but rather, are broadcast to all processes registered to receive such messages. Therefore, a process must filter received messages and shut down only if specific messages are received:

```
type  = SIM
class = SYS_CLASS_1
event = SHUTDOWN
```

In addition, the process must also test the first two bytes of the attached data field to determine if the shutdown message refers to its own sub-system. It is a matter of convention that two characters are used. Any sequence of characters may be used as long as they do not conflict with other sub-system designations. For example, AP refers to the Application sub-system, and HS refers to the Hot Swap Event Manager sub-system. These characters come from the SubSystemName portion of the SHUTDOWN line in ha_sim.conf. As necessary, new letter combinations can be used, and the only coding change needed is localized to the process supporting the new SubSystemName.

SubSystemName is (optionally) followed by a single space and then by SubSystemArguments, if present. SubSystemArguments can be any combination of characters that has meaning to the receiving process. The data field is `NULL` terminated. The data field of a `SHUTDOWN` message has the following form:

Table 3-1: Shutdown Message Data Field

Field Element	Description
`SubSystemName`	(2 bytes by convention).
Space char	(only if `SubSystemArguments` is present). `SubSystemArguments` (optional, variable length - can be used to pass sub-system specific information to the `SubSystemName` process.
NULL char.	

Initialization

At init time, `ha_sim` starts all processes referenced in the configuration file. Processes are started via fork and `execv`. `ha_sim` cannot determine if a process starts successfully. The only means of determining the health of a started process is the `SIGCHLD` signal generated when the child process either forks itself to become a daemon, or dies. If the `SIGCHLD` signal is caught by `ha_sim` before the timeout period, the process is assumed to be running OK. If the timeout occurs before the `SIGCHLD` signal is caught by `ha_sim`, an error is assumed and `ha_sim` logs an error and continues to the next line in the config file. After all processes start, `ha_sim` logs a success message and exits.

`ha_sim` does not send or receive any messages when running in *init* mode.

Shutdown

At shut down time, `ha_sim` runs until all appropriate sub-systems referenced in the config file are sent a `SHUTDOWN` message. If a timeout period is specified, `ha_sim` waits the specified timeout period after sending the `SHUTDOWN` message and before processing the next config file entry.

`ha_sim` sends and receives messages only when running in `SHUTDOWN` mode.

Default

The default `ha.sim.conf` file contains commands to start the appropriate components of HAP 2.0. In most circumstances, users should not need to modify this file.

Options

Table 3-2: ha_sim Options

Token	Meaning
`-i`	Either `-i` or `-s` must be specified, but not both. This option instructs `ha_sim` to run in *init* mode: Start process entries in the config file with `INIT` entries, and start the associated HA processes.
`-s`	Either `-i` or `-s` must be specified, but not both. This option tells `ha_sim` to run in *shutdown* mode: Stop processes in the config file with `SHUTDOWN` entries, and terminate the associated HA sub-systems.
`-d`	Optional. Run with Debug mode `ON`. Function calls are traced and internal variables are displayed. Useful for unit testing. Output is sent to `STDOUT`. Default is `OFF`.
`-n`	Optional. Use `mddPublicQueueName` as the queue name to send messages to `mdd`. Useful if `mdd` specifies a non-default public queue name with its `-n` argument. Default is `/mdd_public_queue`.
`-q`	Optional. Use `PrivateQueueName` as the queue name on which to receive messages from `mdd`. Default is `/sim_queue`.
`-f`	Optional. Use `ConfigFileName` as the full path name of the configuration file. Default is `/etc/hasw/ha_sim.confg`.

hscmd - Hot Swap Command Utility

Synopsis

```
hscmd [-t timeout] [-q queue_name] [-p public_queue] action
slot
```

Description

The Hot Swap Command utility, `hscmd`, issues an ENUM message to the Hot Swap Event Manager containing the action to occur and the slot number to which it applies.

Timeout, provided with the `-t` option, indicates the time to wait for a response from the Hot Swap Event Manager. The default is 5 seconds, the minimum 1 second, and maximum of 60 seconds.

The `-q` option is used to override the default message queue name, `/hscmd_queue`, used by `hscmd` with `queue_name`.

The `-p` option is used to override the default public message queue name, `/mdd_public_queue`, provided by `mdd` and used by `hscmd` with `public_queue`.

Action may be either `insert` or `extract`.

`slot` is the slot number of the affected slot and must identify a valid slot in the domain in which `hscmd` is executed.

`hscmd` waits for an acknowledgement from the Hot Swap Event Manager indicating that the action completed or an error occurred. If the action succeeded, `hscmd` exits with a status of 0. In the case of an error or timeout, the error is reported to stderr and `hscmd` exits with a non-zero status.

hsem - Hot Swap Event Manager

Synopsis

```
hsem -f <config file> <options>
```

Description

The Hot Swap Event Manager provides basic support for Hot Swap events, for example, Hot Insertion and Hot Extraction. Running as a daemon, it communicates information about these events to applications via message queues.

Options

Table 3-3: hsem Options

Token	Meaning
`-x`	Dump the current slot table and exit.
`-f`	Use the following argument as the comfiguration file.
`-i`	If the following arg is "on" run with enum interrupts enables, and if the arg is "off" run with them disabled.
`-d`	Debug mode.
`-l`	Use the config file to download Compact PCI slot table to `enum`.
`-r`	Read the Compact PCI slot table from `enum`.
`-q`	Register with `mdd` and accept messages from queues.
`-t`	Use the arg for the select time out (poll) value (otherwise 2 secs).

hsi - Hot Swap Insert

Synopsis

`hsi <`*`slot`*`>`

Description

This command uses slot numbers as arguments. This command inserts a recently inserted board in slot <*slot*> of a CPX8216 chassis (Domain A Side Only). Insert the board into the slot, and execute the command `hsi` <*slotnumber*>, for example: `hsi 3`, if slot 3 is used for the new board. This command probes the slot and adds the device to the DRM device tree. All devices are selected, and resources are allocated and made ready for use.

Sample Session

```
bash# hsi 5

Hotswap Insert in slot 5
HotSwap Insert Board
HotSwap insert of board successful
```

hsls - List Non-Bridge Devices

Synopsis

`hsls`

Description

This command uses no arguments. `hsls` probes all the nodes below the Domain A bridge and extrapolates the slot occupancy on a CPX8216 chassis. This program prints the occupied status of slots. `hsls` works only with the Domain A backplane.

Sample Session

```
bash# hsls

Slot#   State
0001    empty
0002    empty
0003    empty
0004    empty
0005    empty
0006    empty
0007    System Controller
0008    Not Present
0009    System Controller
0010    Not Present
0011    empty
0012    empty
0013    empty
0014    empty
0015    empty
0016    empty
```

hsx - Hot Swap Extract

Synopsis

`hsx <slot>`

Description

This command is similar to `hsi`, except that the devices in a particular slot are removed. Once this command runs, the board in the slot can be physically removed. If any of the devices are in use (ACTIVE), then this command prints a message, and the extraction should be done after releasing the device.

Sample Session

```
bash# hsx 5

Hotswap Extract in slot 5
Hot Swap Extract Board
HotSwap extract of board successful
```

mdd - Message Distributor Daemon

Synopsis

mdd <*options*>

Description

The message distributor sets up a public queue and waits for applications and daemons to register. Part of the registration is the identification of a private message queue by which `mdd` and others pass information to the registered process. Applications also express interest in message classes during registration. `mdd` runs as a daemon.

Options

Table 3-4: `mdd` Options

Token	Meaning
`-a`	The maximum applications allowed to connect
`-c`	The maximum number of message classes
`-l`	The maximum data length
`-d`	Run as daemon
`-q`	Queue depth
`-m`	Queue id
`-o`	Output file
`-t`	Trace on
`-n`	Queue name
`-p`	Queue permissions

stmd - State Transition Management Daemon

Synopsis

stmd <*options*>

Description

STMD is the State Transition Management Daemon. It assists Warm Domain Switchover applications in managing a multiple domain system and provides a means of transitioning a domain from Active to Standby and vice versa.

Options

Table 3-5: `stmd` Options

Token	Meaning
`-x`	Dump the current slot table and exit.
`-f`	Use the following argument as the configuration file.
`-i`	If the following argument is used, run with enum interrupts enables. If off, run with them disabled.
`-d`	Debug mode
`-r`	Read the Compact PCI slot table from `enum`.
`-q`	Register with `mdd` and accept messages from queues.
`-t`	Use the argument for the select time out (poll) value (otherwise 2 secs).
`-o`	Usr argument as logfile name.
`-b`	Variable to indicate whether `busno`, `devno`, `funcno` should be separated with blank spaces or underscores when passed to the driver install/uninstall scripts.

tm - Topology Manager Script

Synopsis

```
tm [-d device_name] [-f filename]

# include <tm.h>
```

Description

The topology manager script is invoked to configure the topology of the High Availability chassis. The configuration process is accomplished by reading in the topology configuration file, processing it and setting up the High Availability hardware accordingly.

The topology manager runs until it has finished configuring the hardware, or encounters an error. Any error condition is treated as fatal, causing the manager to halt processing and log the error message.

The default topology manager file is located in `/etc/hasw` and is described by manpage `/usr/man/cat5/tm.conf.5`. The file is specified by the user, and more than one can be used when initializing the HSC hardware. The default filename is `tm.conf`, but any name (and path) can be specified using the `-f` option.

The topology manager configuration file describes host processors, non-host slots, peripheral bays, and power supplies. The topology manager reads this file in and initializes a data structure. After reading in the file, the data structure is then used to configure the hardware by initializing the relevant Hot Swap controller registers via calls to the device driver.

Options

`tm` requires no options to be specified, however, the following options modify the behavior of `tm`.

Table 3-6: Topology Manager Options

Token	Meaning
`-d`	`device_name` Specify the full path name of the Hot Swap controller device to open, read/write to, and close. The default is `/dev/hsc0`.
`-f`	`filename` Specify the full path name of the topology configuration file to be read in. The topology configuration can be changed at any time by invoking the topology manager script. The default file is `/etc/hasw/tm.conf`.

Error Handling

Two types of errors can occur with the topology manager: The first is associated with the host swap controller device, and the second type is associated with the topology file.

The Hot Swap controller device driver can cause errors by returning an error status when opening or closing the device. Any device driver call (to set a bit, take control of a domain, etc.) can return an error status if the hardware does not respond correctly to the requested action.

The topology configuration file can also cause errors when opening or closing the file. In addition, if the format of the topology file is incorrect, then errors will occur.

When either type of error occurs, the topology manager cleans up by optionally closing the Hot Swap controller device and the topology file before writing out an error message and exiting with a non-zero value.

CHAPTER 4

Writing High Availability Platform Applications

Overview

The High Availability Platform enables application writers to develop highly available applications. The High Availability Platform does not itself provide high availability. Application developers are responsible for a number of components of the High Availability model, including:

- Drivers conforming to the HAP 2.0 driver model. See Figure 1-8, "Driver Model," on page 15
- Applications that respond to Hot Swap events, such as insertion and extraction
- Applications transitioning the HAP 2.0 state from Cold Standby to Active on startup of a single system host system, or one side of a dual system host system bringing the inactive side to Warm Standby
- Applications responsible for all error recovery, including driver state change failures, `DRM` failures and card unhealthy failures
- Detecting and managing hardware problems
- Managing a specific message IPC interface with `hsem/STMD`

In addition, if an application requires Warm Domain Switchover, the application must:

- Develop the overall Warm Domain Switchover policy
- Communicate all `DRM` information from the Active to the Standby domain
- Command Domain state transitions. For example, the system commands one Domain to Active and the other to Warm Standby
- Manage Domain state transition failures

The application interface to HAP 2.0 consists of standard Message Queue IPC. Applications must connect to this queue and retrieve messages from it. This interface allows applications to command state transitions.

There are three types of messages that occur on this interface:

- Application commands to HAP 2.0
- HAP 2.0 responses to application commands
- HAP 2.0 asynchronous notifications of system events related to High Availability

The following diagram details the generalized command, status and state flow in the system. It also shows which elements (i.e. application, HAP 2.0 or the kernel) are responsible for which functions.

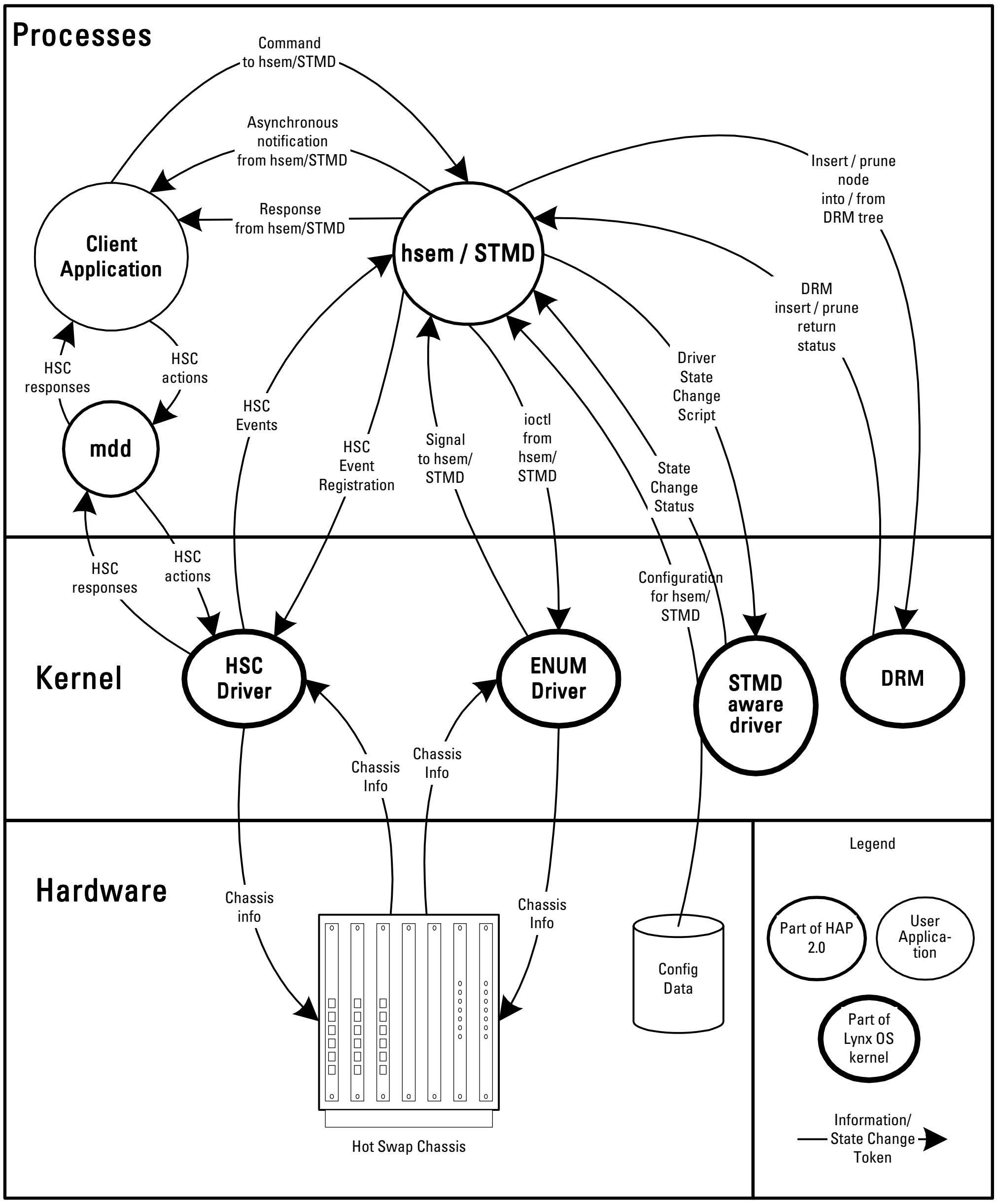


Figure 4-1: HAP 2.0 (Warm Domain Switchover) State and Information Flow

Client Application

A client application can range from simple Hot Swap Controller actions (register accesses) to a full Warm Domain Switchover application. HAP enables applications running on a dual domain system to make one domain Action the other Standby, and to reverse the roles as needed.

An application can be a Hot Swap application on a single domain using `hsem` or a Warm Domain Switchover application using `stmd`.

NOTE: A client application must not attempt to use `hsem` and `stmd` simultaneously.

NOTE: Application developers must be aware that application intervention is required on a dual domain system (CPX8216). Without it, the system comes up and remains in Cold Standby state.

hsem

Basic Hot Swap actions such as card extraction and card insertion are supported by a Hot Swap Event Manager (`hsem`). It informs a registered application when a Card Insert or a Card Extract takes place. It also enables an application to read and write Hot Swap Controller registers.

mdd

The Message Distributor Daemon (`mdd`) is the mechanism by which Hot Swap applications interact with `hsem` and `hscd`. `mdd` is also available to Warm Domain Switchover applications.

stmd

For HAP 2.0, `hsem` (Hot Swap Event Manager) enables application Warm Domain Switchover. `stmd` (State Transition Management Daemon) allows applications greater control of domain states than was previously possible.

`stmd` is the only interface to the HAP 2.0 platform a Warm Domain Switchover application is required to support. All state transitions and Hot Swap events are managed by this module. Tasks include:

- Transitioning internal state to Cold Standby on startup

- Receiving application commands on a message queue, determining the correct type of action to be taken and initiate the necessary state transitions to satisfy the application request
- Responding to signals from `enum` or `hsc` drivers, determining next function (either an `ioctl` to `enum` driver), device adding or deleting to `drm`, changing state of a driver or responding to a notification from an application
- Responding to `drm` failures by returning the appropriate error notification to the application
- Responding to driver state change failures by returning the appropriate error notification to the application

enum

The `enum` driver is responsible for taking appropriate action when the hardware generates an `ENUM#` interrupt.

hscd

The Hot Swap Controller driver (`hscd)` is responsible for reporting basic Compact PCI events to the application. It also provides applications direct access to slot registers.

LynxOS Elements Used

In addition to the standard POSIX system calls, applications that use High Availability Platform must provide drivers for each card supported in the Hot Swap scenarios (i.e, Hot Extraction and Hot Insertion). The drivers must conform to the standard LynxOS driver model, as well as the special state and transition requirements set by HAP 2.0 as illustrated in “Driver Model” on page 15 and in “The Device Driver Model” on page 44.

As indicated in Figure 4-1, “HAP 2.0 (Warm Domain Switchover) State and Information Flow,” on page 39, the main elements of High Availability Platform are the application drivers located in kernel space: the `enum` driver and the `hsc` driver. In addition, the Device Resource Manager (`DRM`) is an important part of activating slot devices.

The following example provides a basic `stmd.conf` configuration file. For more information, please see the `stmd.conf(5)` man page, orAppendix C, “stmd.conf file example”.

```
; stmd.conf
;
; For more detailed information, see the man page.
;
; Copyright (c) 1998,1999 MOTOROLA
; and
; Copyright (c) 2000 LynuxWorks
; All Rights Reserved
;

; The "slots" entry defines the slot table for the specific chassis.
; Field 1 = Physical Slot number
; Field 2 = Bus Number
; Field 3 = Device Number
; Field 4 = Function Number
;
; The slot table below is for the domain A side of an 8216 Chassis.
; It should not be modified.
;
slots
slot 1,1,14,0
slot 2,1,13,0
slot 3,1,12,0
slot 4,1,11,0
slot 5,1,10,0
slot 6,1,9,0
slot 7,0,0,0
slot 8,0,0,0
slot 9,0,0,0
slot 10,0,0,0
slot 11,42,14,0
slot 12,42,13,0
slot 13,42,12,0
slot 14,42,11,0
slot 15,42,10,0
slot 16,42,9,0
end_slots

; This is the drivers section of the config file.  In it will be
; sections for each driver.
drivers

driver

; This prefix will be used when the device is created.

device_name_prefix /dev/wan
```

Figure 4-2: stmd.conf file Exampe

```
; This signature is used during device parsing.
; It corresponds to the PCI device specific vendor ID.

signature
vendor 0x10b59080,0,255
end_signature

; The script invoked during device insertion transition.

device_install
system /etc/hasw/install.scr
end_device_install

; The script invoked during device removal transition.

device_uninstall
system /etc/hasw/uninstall.scr
end_device_uninstall
; The script invoked during domain transition to active.

active_command /etc/hasw/active.scr

; The script invoked during domain transition to standby.

standby_command /etc/hasw/standby.scr

end_driver

end_drivers
```

Figure 4-3: stmd.conf file Example (Continued)

The bus number in this example depends on the DRM bus number allocator. For `stmd`, bus and slot numbers can be described the same for either side A or B. Slots 1-6 are mapped via Bus number 1, while slots 11-16 are mapped via bus number 42.

If a processor has a PMC card containing a bridge, then 41 should be added to the above bus numbers.

Device numbers are given by (15 - slot) for 0 < slot < 7 and by (25 - slot) for 10 < slot < 17 for the CPX8216 chassis.

Function numbers are always 0.

For additional information, please see the `stmd.conf(5)` man page.

Device Resource Manager

Device Resource Manager (`DRM`) is a LynxOS module that manages device resources. `DRM` assists device drivers in identifying, setting up, and managing devices and device address space. Using `DRM` services, device drivers can use devices without requiring information about board-specific configurations. `DRM` is enhanced to support Basic Hot Swap services.

The Device Driver Model

Previous releases of HSP provided a model for Hot Swap-aware drivers. Previous states and transitions are reorganized to draw parallels with the new HAP 2.0 model.

The Inactive Driver state is a superset of:

- No Driver
- Loaded
- Idle
- Software Init
- Offline
- Software Closing
- Zombie
- Released

The Standby state consists of:

- Software Quiesce
- Hardware Stop

The Active state consists of:

- Software Ready
- Hardware Init
- Hardware Ready
- Hardware Standby

The device driver writer supplies scripts that transition the driver from one superset state to another. These are:

1. **I-S**, which must include the following legacy state changes:
   - Software Init to Quiesce
   - Idle to Hardware Stop (`dev_open` part1)
2. **S-A**, which must include the following legacy state changes:
   - Software Quiesce to Software Ready (`hs_enable`)
   - Hardware Stop to Hardware Init (`dev_open` part 2)
   - Hardware Stop to Hardware Standby (`hs_standby`)
3. **A-S,** which must include the following legacy state changes:
   - Software Ready to Software Quiesce (`hs_quiesce`)
   - Hardware Ready to Hardware Stop (`hs_stop`)

Figure 4-4 shows the correspondence between that model and the one required by the HAP 2.0 release.

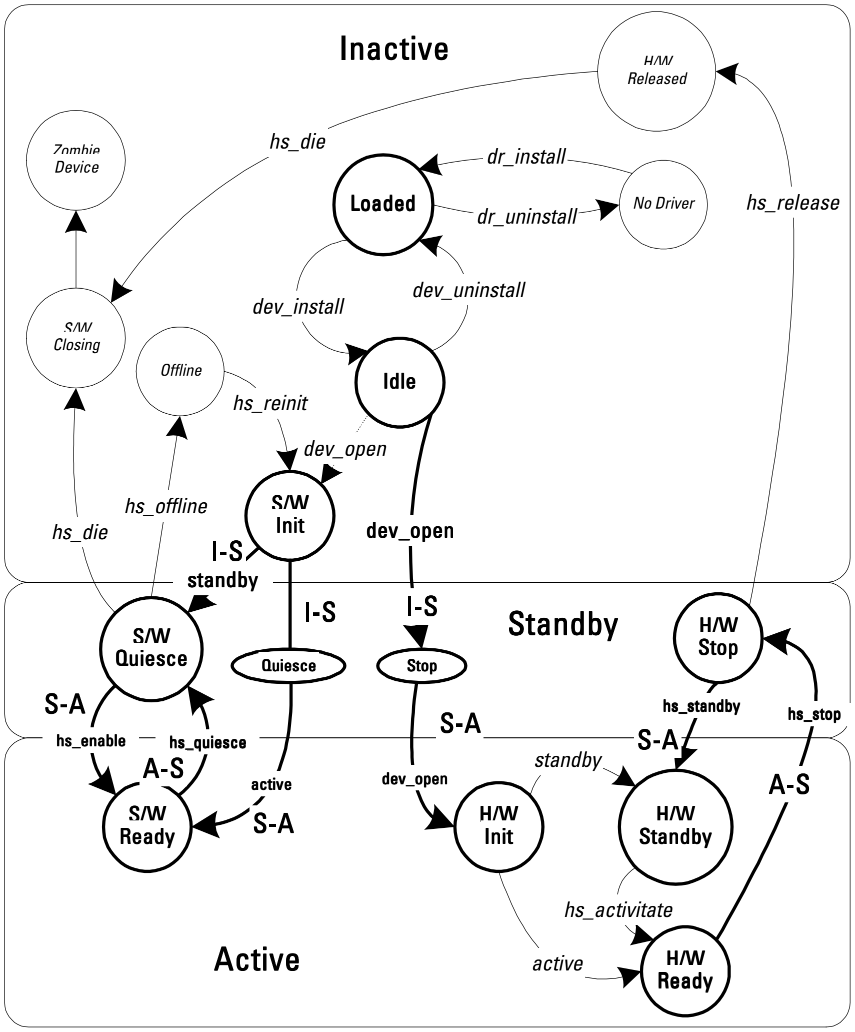


Figure 4-4: HSAD Driver Model with corresponding HAP 2.0 states

STMD Driver Model

Drivers must be written to accommodate these states: Inactive, Standby and Active (see Figure 1-8, "Driver Model," on page 15). The new driver model does not attempt to label actual `IOCTL` calls to a driver. Only the notion of state is used.

- Inactive = Does nothing
- Standby = Ready to perform function, but not accessing hardware
- Active = Ready to perform function, accessing hardware

Entry points for these transitions must be filled by user-defined code, which is called during `stmd` operation. See "stmd - State Transition Management Daemon" on page 32.

Writing Hot Swap Applications

Hot Swap-only applications are hosted on single bus CompactPCI systems (CPV1204 and CPX2208), or on dual bus systems (CPX8216), where Domain Switchover is not implemented. In this case, it is necessary to use `hsem` (See "hsem - Hot Swap Event Manager" on page 28. for usage) as well as `mdd` (See "mdd - Message Distributor Daemon" on page 31.).

During installation, the user is queried for the chassis type, whether CPX 8216 or CPX2208/CPV1204. If the user specifies a single domain chassis, then `hsem` is set as the daemon. It is not required that the user to know how to start `hsem` or `mdd`. The daemon appropriate for the chassis is installed in the `ha_sim`.conf configuration file and `ha_sim` starts the correct daemon when invoked.

Initialization and Registration

Hot Swap applications uses the following header files to facilitate the interface to `hsem`:

```
# include <ha_msg.h>
# include <hsem.h>
```

In addition to providing device drivers that conform to the HAP 2.0 model, applications need to open the `mdd` message queue and establish a command message queue to `mdd`. In order to receive notification of Hot Insertion and Hot Extraction events, as well as issue other commands to `hscd`, the application must register with `mdd`.

The following shows the call to open the correct message queues:

```
mdd_q = mq_open ( MDD_QUEUE, O_WRONLY );
mq_getattr ( mdd_q, &attr );
app_q = mq_open ( “/myQ”,
 O_CREAT|O_RDONLY,
 creat_mod,
 mq_attr, );
```

The application must then send a message registering with `mdd`. This also allows `mdd` to be aware of the application’s private message queue.

```
Msg smsg;
ReqData *data_p;
unsigned charlen= aizeof(Msg)
- MAX_DATA_LEN
+ sizeof(ReqData);
smsg.class = SYS_CLASS_1;
smsg.type = DISTR;
smsg.event = REGISTER;
data_p = (ReqData *) smsg.data;
strcpy( data_p->queue_name, “/myQ” );
mq_send( app_q, (char *) smsg, len, 1 );
The message structure is
typedef struct Msg
{
unsigned charorigin_id;
unsigned charmd_id;
unsigned charlen;
unsigned charclass;
unsigned longtype;
unsigned shortevent;
unsigned chardata[MAX_DATA_LEN];
} Msg;
The data field (ReqData) structure is
typedef struct RegData
{
unsigned charqueue_name[MAX_QUEUE_NAME_LEN];
unsigned charclass;
unsigned longtype;
unsigned shortstatus;
} RegData;
```

Operation

The following table details Hot Swap-related message types and the actions required:

Table 4-1: Hot Swap Messages

Message Type	Data			Actions
	Field	Contents	Possible Results	
HS_INSERT	origin_id		SUCCESS	Open (application) the device and perform needed operations
	class	SYS_CLASS_1	FAILURE_INVALID_SLOT	
	type	HS_UTIL	FAILURE_UNHEALTHY_CARD	
	event	USER_ENUM	FAILURE_DRM_DEVICE_CONFIG	
	EnumData.action	HS_INSERT	FAILURE_DRIVER_STATE_CHANGE	
	EnumData.slot	*<slot no.>*		
	EnumData.result	<see below>		
HS_EXTRAC T	origin_id		SUCCESS	Complete pending operations and close (application) the device
	class	SYS_CLASS_1	FAILURE_INVALID_SLOT	
	type	HS_UTIL	FAILURE_DRM_DEVICE_CONFIG	
	event	USER_ENUM	FAILURE_DRIVER_STATE_CHANGE	
	EnumData.action	HS_EXTRACT		
	EnumData.slot	*<slot no.>*		
	EnumData.result	*<see below>*		

The following table displays the various error notifications and their causes:

Table 4-2: Hot Swap-Related Error Notifications

Error Name	Error Num	Meaning
SUCCESS	0	The action completed successfully
FAILURE_INVALID_ACTION	1	Invalid action requested
FAILURE_INVALID_SLOT	2	Action requested for invalid slot
FAILURE_UNHEALTHY_CARD	3	Inserted Card sets its unhealthy bit

Table 4-2: Hot Swap-Related Error Notifications (Continued)

Error Name	Error Num	Meaning
`FAILURE_DRM_DEVICE_CONFIG`	4	DRM failure on Insert/Prune Device
`FAILURE_DRIVER_STATE_CHANGE`	5	Driver failed to change to requested state

Bottom Up Hot Insertion

The following figure details the state transitions that take place in a Hot Insertion event.

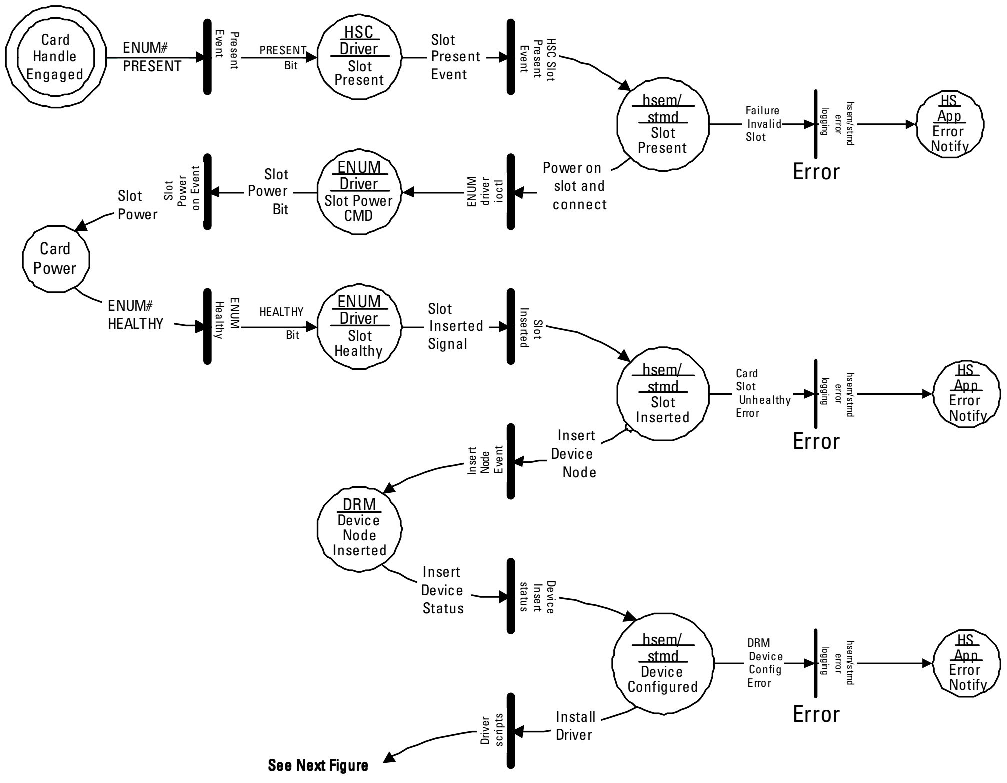


Figure 4-5 part 1: Successful Hot Insertion State

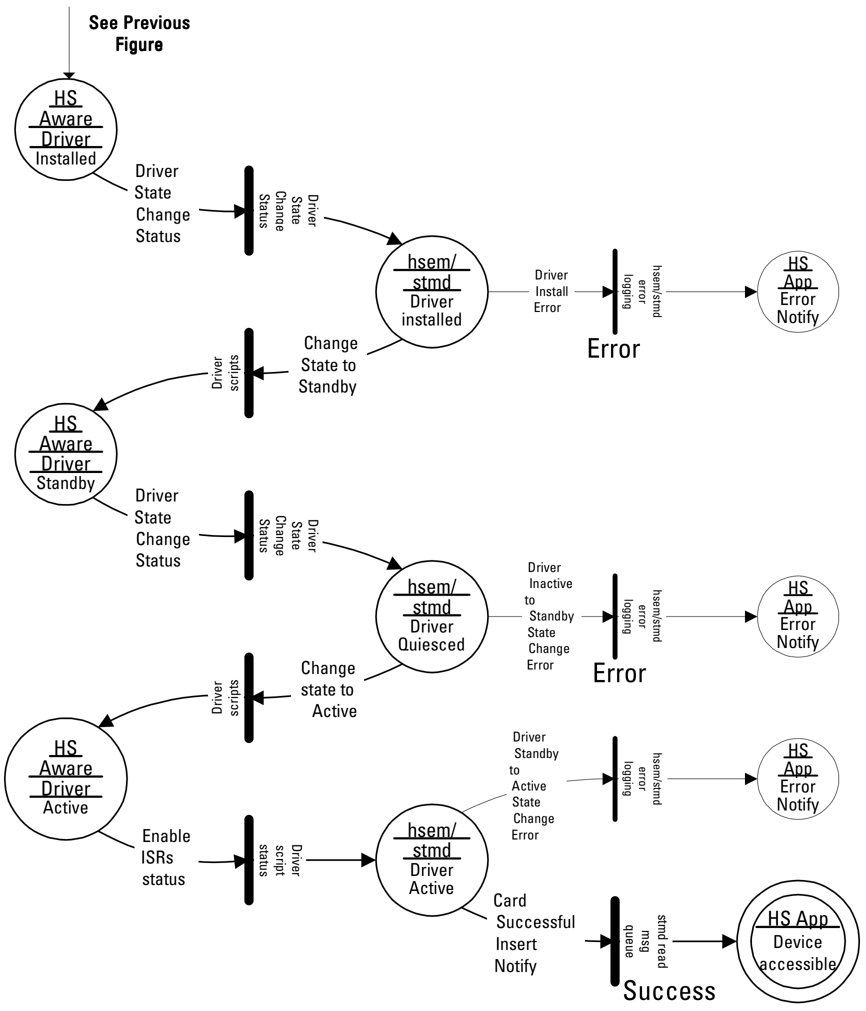


Figure 4-5 part 2: Successful Hot Insertion State

Error conditions are reported:

- After power is applied to the card
- After `DRM` is requested to configure the device information

- After the device driver is commanded to transition from Inactive to standby and from Standby to Active
- The types of messages that an application can receive are detailed above. In this example, the message received would be `HS_INSERT`.

Hot Extraction

The following diagram shows the state transitions for a card extraction:

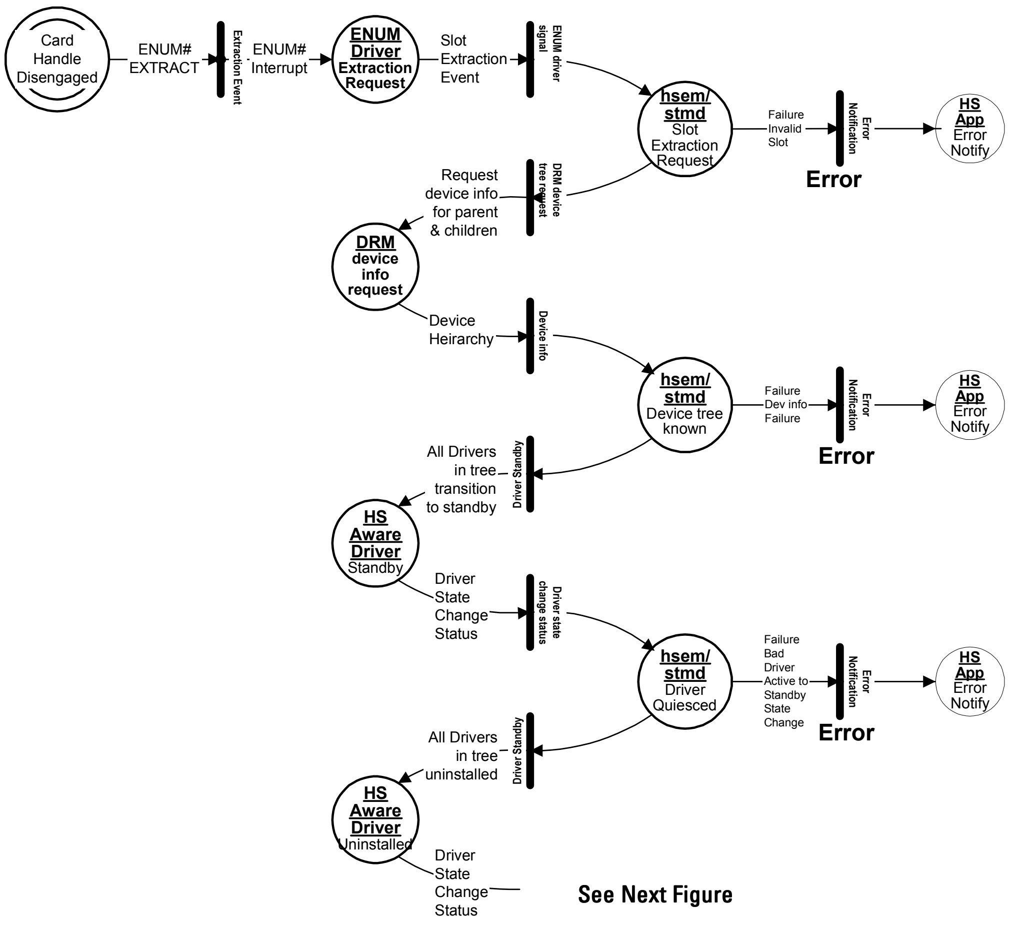


Figure 4-6 part 1: Card Extraction State

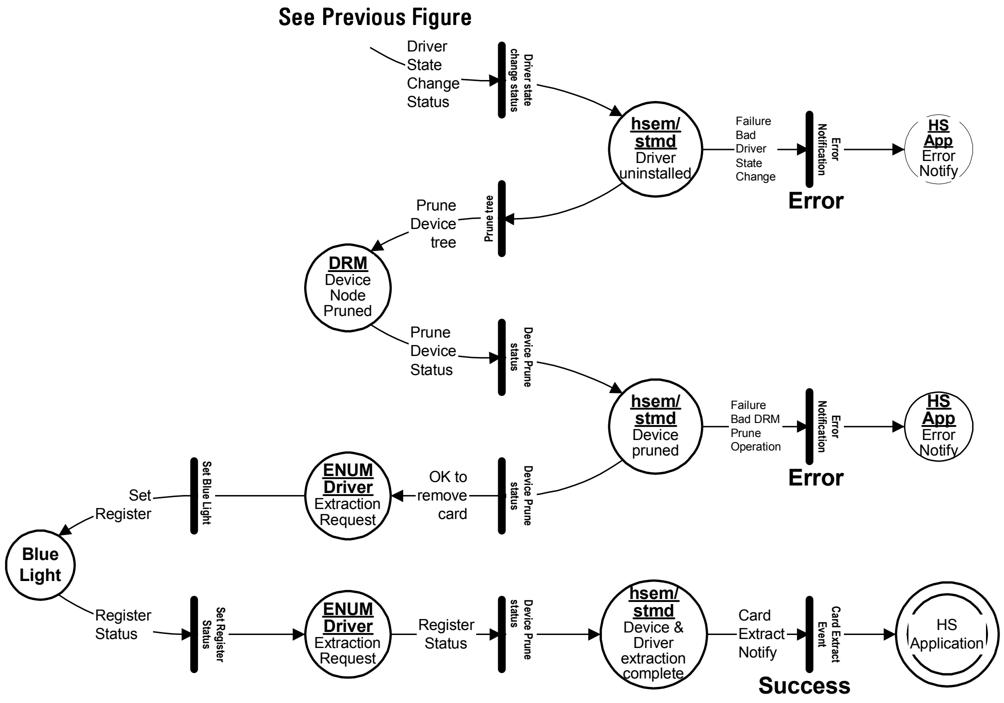


Figure 4-6 part 2: Card Extraction State

Error conditions are reported:

- After `Extract` interrupt (with invalid slot)
- After `DRM` is requested to return the device information
- After the device driver is commanded to transition from active to standby
- After `DRM` returns an error condition on a Device prune operation

Writing Warm Domain Switchover Applications

Warm Domain Switchover extends the types of messages that are sent and received, and changes nomenclature. Warm Domain Switchover can only be run on a dual domain chassis (Motorola CPX8216).

In this case, it is necessary to use `stmd` (See “stmd - State Transition Management Daemon” on page 32. for usage) as well as `mdd` (See “mdd - Message Distributor Daemon” on page 31.).

During installation, the user is queried for the chassis type, whether CPX82xx or CPX22xx/CPV12xx. If the user specifies a multiple domain chassis, `stmd` is the default daemon. It is not necessary for the user to know how to start `stmd` or `mdd`. The daemon appropriate for the chassis is installed in the `ha_sim.conf` configuration file and `ha_sim` starts the correct daemon when invoked.

Warm Domain Switchover framework in HAP 2.0 provides only a framework for developers to create High Availability software. The following are responsibilities of the system designer to develop a Highly Available system:

- Selection algorithm for Active/Standby state assignment on startup
- Detection algorithm for initiation of failover
- Checkpoint and reliable communication of slot and chassis information across domains
- Error resolution and recovery (HAP halts transitions on errors, and provides all available information to the user for resolution
- Scripts/applications to execute during driver state transition period
- Driver code following stateful driver model (conceptual, no forced syntax
- PICMG non-compliance issues (typical here are non-standard enum or non-standard Hot Swap model issues)

Initialization and Registration

Initialization proceeds in similar fashion as Hot Swap applications, except there are additional queues for talking to `stmd`. There is an additional header file to be included as well.

```
# include <ha_msg.h>
# include <hsem.h>
# include <stmd.h>
```

Setting up the message queues must take this form:

```
mdd_q= mq_open( MDD_QUEUE, O_WRONLY );
app_q= mq_open(“/myQ”,
O_CREAT|O_RDONLY,
creat_mod,
mq_attr, );
stmd_wr_q= mq_open(STMD_MSGQ_WR, O_RDONLY );
stmd_rd_q= mq_open(STMD_MSGQ_RD, O_WRONLY);
```

The message structures are the same as Hot Swap applications and it is necessary to register with the Message Distributor (`mdd`).

The following state transition diagram shows how, upon startup, `STMD` transitions to Cold Standby:

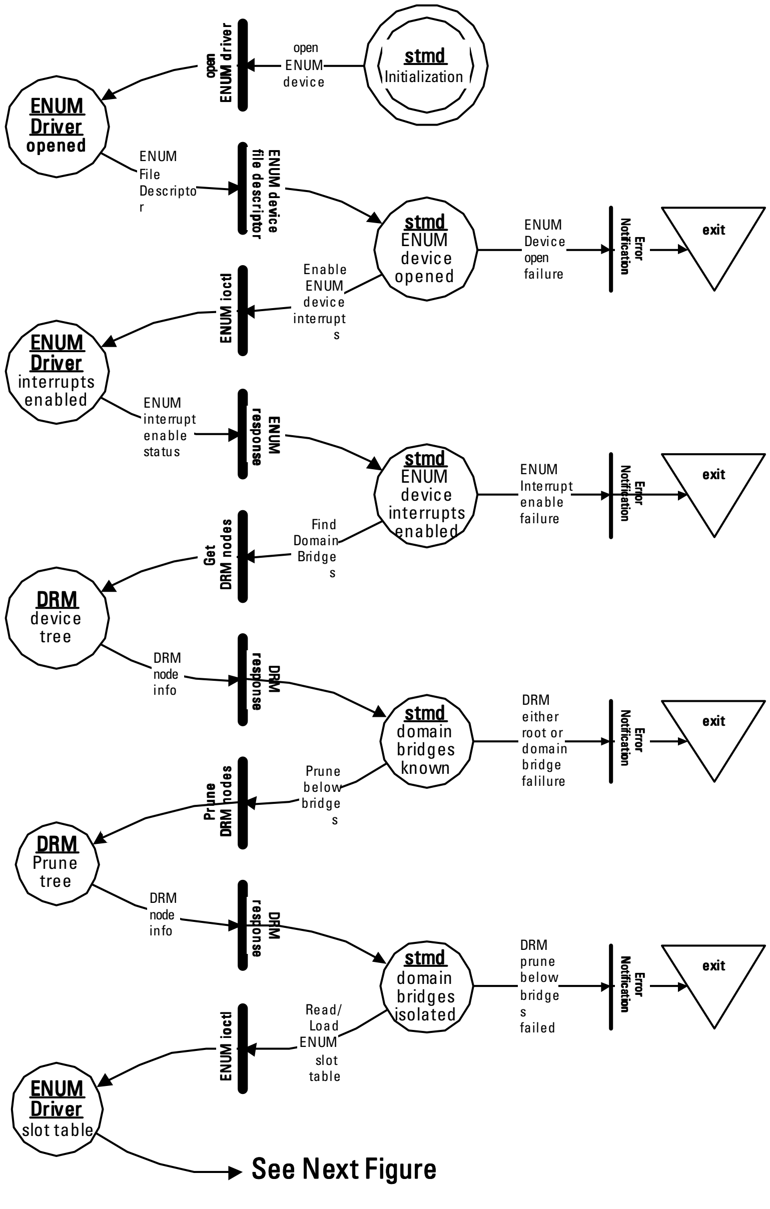


Figure 4-7: STMD Initialization (Transition to Cold Standby)

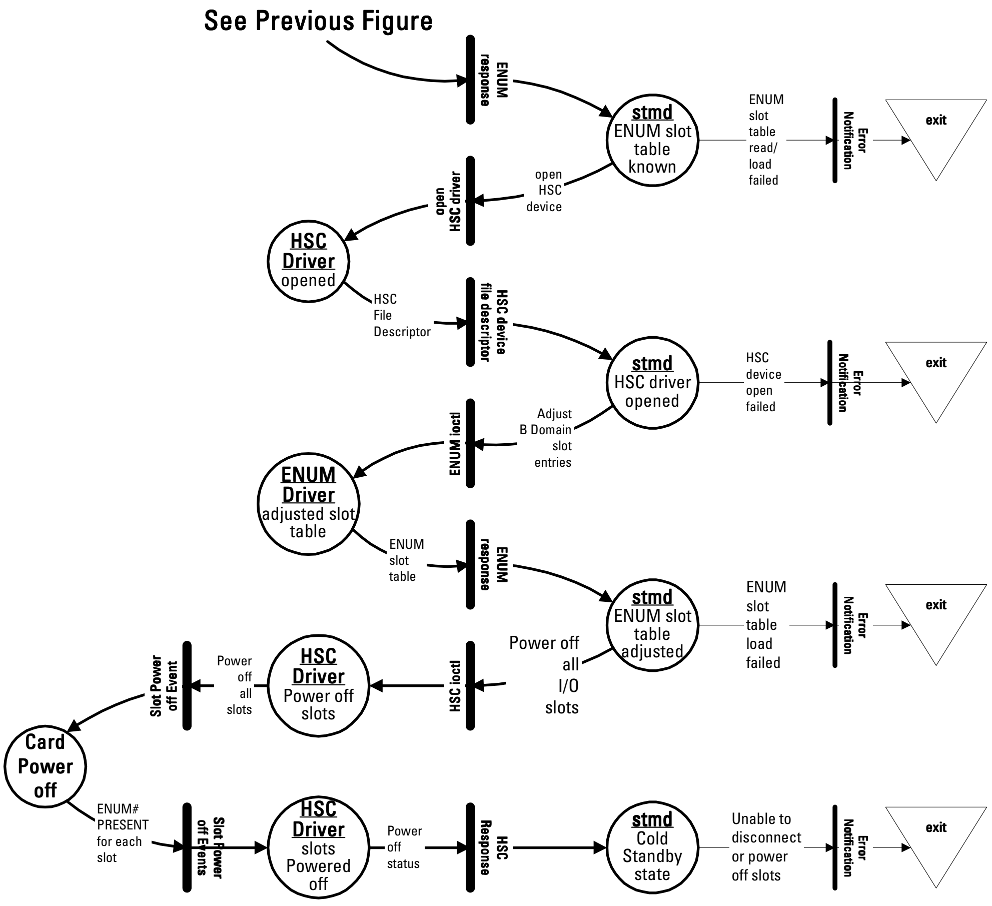


Figure 4-7 part 2 : STMD Initialization (Transition to Cold Standby)

Operation

The following table denotes STMD-related message types and the actions required. Note that in most cases the actual names begin with STMD:

Table 4-1: State Transition Messages

Message Type	Data Field	Data Content	Possible Results	Actions
CS_TO_ACTIVE_CMD	msg	CS_TO_ACTIVE	SUCCESS (E_NOERR)	Transition the domain from Cold Standby to Active; on errors, switch other domain to active, correct problem, and restart system controller.
	type	COMMAND	E_ENABLE_ENUM	
	msg_data	none	E_TAKE_BUS_SLOTS	
CS_TO_ACTIVE_RESP	msg	CS_TO_ACTIVE	E_PWRON	
	type	RESPONSE	E_GET_DRM_INFO	
	msg_tag		E_HSC_REGISTER	
	status		E_ENUM_REGISTER	
	errnum		E_PWROFF	
	errstr		E_DRM_INSERT	
	slot_mask			
	num_cards			
CS_TO_WS_CMD	msg	CS_TO_WS	SUCCESS (E_NOERR)	Domain state is changed but no error conditions are possible (errors, if any, come on insert actions).
	type	COMMAND		
	msg_data	none		
CS_TO_WS_RESP	msg	CS_TO_WS		
	type	RESPONSE		
	msg_tag			
	status			
	errnum			
	errstr			

Table 4-1: State Transition Messages (Continued)

Message Type	Data Field	Data Content	Possible Results	Actions
WS_TO_ACTIVE_CMD	msg	WS_TO_ACTIVE	SUCCESS (E_NOERR)	Domain state is changed; on error, correct problem and reset system controller.
	type	COMMAND	E_ENABLE_ENUM	
	msg_data	none	E_DISABLE_ARBITRATION	
WS_TO_ACTIVE_RESP	msg	WS_TO_ACTIVE	E_HSC_REGISTER	
	type	RESPONSE	E_ENUM_REGISTER	
	msg_tag		E_TAKE_BUS_SLOTS	
	status		E_SYSCTL	
	errnum		E_DRIVER_ENABLE	
	errstr		E_ENABLE_ARBITRATION	
			E_DRIVER_ENABLE_ISR	
ACTIVE_TO_WS_CMD	msg	ACTIVE_TO_WS	SUCCESS (E_NOERR)	Domain state is changed; on error correct problem and or reset system controller, in either case, accompany with other side transitioning from Warm Standby to Active.
	type	COMMAND	E_HSC_DEREGISTER	
	msg_data	none	E_ENUM_DEREGISTER	
ACTIVE_TO_WS_RESP	msg	WS_TO_ACTIVE	E_DRIVER_STANDBY	
	type	RESPONSE	E_DRIVER_DISABLE_ISR	
	msg_tag		E_DISABLE_ARBITRATION	
	status		E_ENUM_DISABLE	
	errnum			
	errstr			
GET_SYSTEM_STATE_ CMD	msg	GET_SYSTEM_ST ATE	SUCCESS (E_NOERR)	Application use information
	type	COMMAND	msg data is STATE_COLD, STATE_WS, or STATE_ACTIVE	
	msg_data	none		
GET_SYSTEM_STATE_ RESP	msg	GET_SYSTEM_ST ATE		
	type	RESPONSE		
	msg_data			

Table 4-1: State Transition Messages (Continued)

Message Type	Data Field	Data Content	Possible Results	Actions
CARD_INSERT_CMD	msg	CARD_INSERT	SUCCESS (E_NOERR)	Open (application) the device and perform needed operations; on error, extract card, correct problem, and re-insert. Switch to backup domain if operation is critical.
	type	COMMAND	E_INVALID_SLOT	
	msg_data	card_inserted_s	E_UNHEALTHY_CARD	
	slot_num	*<slot no.>*	E_PUSH_DRM_INFO	
	vendor_id		E_SYSCTL	
	device_id		E_DRIVER_NOT_FOUND	
CARD_INSERT_RESP	msg	CARD_INSERT	E_DRIVER_INSTALL	
	type	RESPONSE	E_DRIVER_STANDBY	
	msg_data	card_id_s	E_PWRON	
	slot_num	*<slot no.>*		
CARD_REMOVE_CMD	msg	CARD_REMOVE	SUCCESS (E_NOERR)	Complete pending operations and close (application) the device.
	type	COMMAND	E_INVALID_SLOT	
	msg_data	*<slot no.>*	E_DRIVER_STANDBY	
CARD_REMOVE_RESP	msg	CARD_REMOVE	E_DRIVER_DISABLE_ISR	
	type	RESPONSE	E_DRIVER_NOT_FOUND	
	msg_data	card_id_s	E_DRIVER_UNINSTALL	
	slot_num	*<slot no.>*	E_DRM_PRUNE	
GET_SLOT_ENTRY_CMD	msg	GET_SLOT_ENTRY	SUCCESS (E_NOERR)	Application use information
	type	COMMAND		
	msg_data	*<slot no.>*		
GET_SLOT_ENTRY_RESP	msg	GET_SLOT_ENTRY		
	type	RESPONSE		
	msg_data	slot_entry_s		
	device_ident[16]	device_ident_s		
	busno			
	funcno			
	vendor_id			
	device_id			
	drm_info	char*		

Table 4-1: State Transition Messages (Continued)

Message Type	Data Field	Data Content	Possible Results	Actions
GET_VERSION_CMD	msg	GET_VERSION	SUCCESS (E_NOERR)	Application use information
	type	COMMAND		
	msg_data	none		
GET_VERSION_RESP	msg	GET_VERSION		
	type	RESPONSE		
	msg_data	version_s		
	unused	char		
	major_version	char		
	minor_version	char		
	maint_version	char		
TRACE_CMD	msg	TRACE	SUCCESS (E_NOERR)	Application gets messages tracing STMD operations
	type	COMMAND		
	msg_data	=1 trace on, 0, off		
TRACE_RESP	msg	TRACE		
	type	RESPONSE		
	msg_data	none		
FAILURE_NOTIFY	msg	FAILURE	event can be one of the following:	Application may handle the conditions listed in Table on page 44.
	type	NOTIFICATION	ASYNC_INSERTION	
	msg_data	consists of	ASYNC_EXTRACTION	
	event	int	CS_TO_ACTIVE_TRANS	
			INIT	
			PROCESS_MSGQ	
			ERROR_INTERNAL	
			E_ENUM_COMPLETE_EVENT	
			E_ENUM_READ_SLOT_TABLE	
			ERROR_GET_DRM_INFO	
			CATCHALL	

The following table shows the various error notifications and their causes:

Table 4-2: State Transition Management Error Notifications

Error Name	Error Strings	Meaning
`SUCCESS (E_NOERR)`	`none`	The action completed successfully (no error occurred).
`E_PWROFF`	`“Unable to power off slot”` `“Unable to disconnect slot”`	During initialization, `CS_TO_ACTIVE` state transition, hot insert or extract , if `STMD` is unable to power off or disconnect a slot, this message error is issued.
`E_PWRON`	`“Unable to power on slot”` `“Unable to connect slot”`	In `CS_TO_ACTIVE` or `CARD_INSERT`, `STMD` returns this status if it is unable to power on or connect.
`E_RELEASE_BUS_SLOTS`	`none`	Unused
`E_TAKE_BUS_SLOTS`	`none`	In `CS_TO_ACTIVE` and `HS_TO_ACTIVE`, if `STMD` is unable to command hsc to take control of given bus slots.
`E_HSC_OPEN`	`“stmd_wds_init: Unable to open /dev/hsc0. Possible problems: permissions, non-root user, HSC driver not loaded”`	See error string.
`E_DRM_TRAVERSE`	`none`	Unused
`E_DRIVER_INSTALL`	`"install_drivers_for_slot_nodes: error installing driver for vid nn did nn slot nn”`	The driver installation script returned an error.
`E_DRIVER_ENABLE`	`The character special file name`	The driver enable script returned an error.
`E_DRIVER_NOT_FOUND`	`none`	The `STMD` tables did not contain driver quiesce and/or uninstall scripts.
`E_MKNOD`	`none`	Unused
`E_DRIVER_OPEN`	`none`	Unused
`E_ENABLE_ARBITRATION`	`“Error clearing LOCK bit in HSC”`	`STMD` failed to enable arbitration in `HS_TO_ACTIVE` state change.
`E_DISABLE_ARBITRATION`	`“Error setting LOCK bit in HSC”`	In `HS_TO_ACTIVE` and `ACTIVE_TO_HS` state changes, `STMD` failed to disable arbitration.
`E_DRIVER_QUIESCE`	`none`	Unused
`E_FLUSH_FIFOS`	`“Error from get_drm_root() call”` `“Error from read_pci_node() call”`	In `ACTIVE_TO_HS`, there was an error in flushing bridge FIFOs.

Table 4-2: State Transition Management Error Notifications (Continued)

Error Name	Error Strings	Meaning
E_SYSCTL	"process_insertion: get_drm_root returned error" "process_insertion: probe_drm_node returned error" "process_insertion: Unable to select DRM subtree" "process_insertion: Unable to alloc DRM subtree" "process_insertion: pci_get_complete_node for slot node returned error" "find_new_node: get_drm_root returned error" "find_new_node: get_next_drm_node returned error" "find_new_node: pci_get_complete_node returned error" "find_new_node: domain bridge has no child" "find_new_node: get_drm_child returned error" "find_new_node: pci_get_complete_ node returned error" "install_drivers_for_slot_nodes: pci_get_complete_node returned error" "main: prune domainA returned error" "main: prune domainB returned error" find_domain_bridges: get_drm_root returned error" "drm_program_pci_bridges: get_drm_root returned error" "drm_program_pci_bridges: get_next_drm_node returned error" "drm_program_pci_bridges: pci_get_complete_node returned error" "drm_program_pci_bridges: PCI_PROGRAM_BUSNODE returned error"	process_insertion: find_new_node: install_drivers_for_slot_nodes: main: find_domain_bridges: drm_program_pci_bridges: error return from a sysctl() (DRM) call
E_DRIVER_BOOT	none	Unused
E_GET_DRM_INFO	"process_st_insertion: get_drm_partial_tree call failed"	process_insertion: Failed to get DRM information about the inserted line card.
E_PUSH_DRM_INFO	"Possible problem: wrong length"	stmd_hs_card_insert: Failed to push DRM information about a line card into the standby.

Table 4-2: State Transition Management Error Notifications (Continued)

Error Name	Error Strings	Meaning
E_DISABLE_CPCI_INTR	"Possible cause: sysctl not up to date"	stmd_wds_hs_to_active: unable to disable cPCI interrupts.
E_ENABLE_CPCI_INTR	"Possible cause: sysctl not up to date"	stmd_wds_hs_to_active: unable to enable cPCI interrupts.
E_RESET_CPU	none	Unused
E_ERROR_LEVEL	none	Unused
E_DISABLE_ENUM	“Unable to turn off PROP ENUM” “Unable to turn off ENUM A MASK” “Unable to turn off ENUM B MASK”	Initialization: stmd_wds_active_to_hs: Failed to turn off ENUM interrupts in the HSC.
E_ENABLE_ENUM	“Unable to turn on PROP ENUM” “Unable to turn on ENUM A MASK” “Unable to turn on ENUM B MASK”	stmd_wds_cs_to_active: stmd_wds_hs_to_active: Reason - Failed to turn on ENUM interrupts in the HSC.
E_DRIVER_UNINSTALL	command line of the driver uninstall script	uninstall_drivers_in_slot: Driver uninstall script returned error.
E_DRM_PRUNE	slot number	DRM PRUNE operation failed.
E_DRM_INSERT	“Error inserting local domain bridge” “Error inserting remote domain bridge”	stmd_wds_cs_to_active: Inserting either the local or the remote domain bridge failed during CS_TO_ACTIVE.
E_HSC_REGISTER	none	In CS_TO_ACTIVE & HS_TO_ACTIVE, the call to hsc_SetEventFunction() to set an event handler failed.
E_HSC_DEREGISTER	none	In ACTIVE_TO_HS, the call to hsc_SetEventFunction() to clear the event handler failed.
E_INSERT_CARD	none	Unused
E_NEW_CARD	“find_new_node: Unable to find node”	STMD was unable to find in DRM and nodes that corresponded to the inserted line card.
E_UNKNOWN_MESSAGE	none	Unused
E_NOT_CMD	none	The msg_type field in the received message was not STMD_CMD.
E_SYSTEM_STATE	“active state” “cold state” “warm standby state”	The requested operation is not possible in the state STMD is in.
E_WRONG_SLOT	none	A wrong slot number is passed in.

Table 4-2: State Transition Management Error Notifications (Continued)

Error Name	Error Strings	Meaning
E_DOMAIN_BRIDGE	"find_new_node: could not find domain bridge"	STMD failed to find the correct domain bridge while traversing the DRM tree after a line card insertion.
E_MQ_NOTIFY	"Unable to set up message queue notification signal: errno = <errno>"	The call to mq_notify() to set up message queue notification failed.
E_SIGACTION	output of strerror	The call to sigaction() failed.
E_SYNC_SIGNAL	"Received signal <signo>"	STMD received a SIGILL, SIGFPE, SIGBUS, or SIGSEGV.
E_CFG_OPEN	none	stmd_hs_init: Failed to open the STMD config file
E_GET_SLOT_TBL	none	stmd_hs_init: Failed to read the slot table from the STMD config file.
E_GET_DRV_TBL	none	stmd_hs_init: Failed to read the drivers table from the STMD config file.
E_MQ_RECEIVE	none	process_stmd_msg: The call to mq_receive() failed.
E_HSC_GET_DOMAIN_ID	none	Unused
E_ENUM_SCAN	none	Unable to scan for ENUM event after receiving SIGALRM.
E_ENUM_GET_EVENT	none	ioctl to get ENUM event in the ENUM event handler returns error.
E_DRIVER_STANDBY	Special device file name of driver	Driver standby script returned error.
E_DRIVER_ENABLE_ISR	Special device file name of driver	Driver enable ISR script returned error.(no longer used).
E_DRIVER_DISABLE_ISR	Special device file name of driver	Driver disable ISR script returned error.(no longer used).
E_ENUM_REGISTER	none	stmd_wds_cs_to_active: stmd_wds_hs_to_active attempt to register for ENUM interrupts failed.
E_ENUM_DEREGISTER	none	stmd_wds_active_to_hs: Attempt to deregister for ENUM interrupts failed.
E_ENUM_READ_SLOT_TABLE	none	Initialization: unable to read slot table from enum driver.
E_ENUM_LOAD_SLOT_TABLE	none	Initialization: stmd_hs_card_insert: unable to download slot table to enum driver.
E_ENUM_COMPLETE_EVENT	none	ENUM ioclt failed during insertion.

State Transitions

This section details various state transitions used.

Cold Standby to Active

The following state diagram describes the Cold Standby to Active domain state transition:

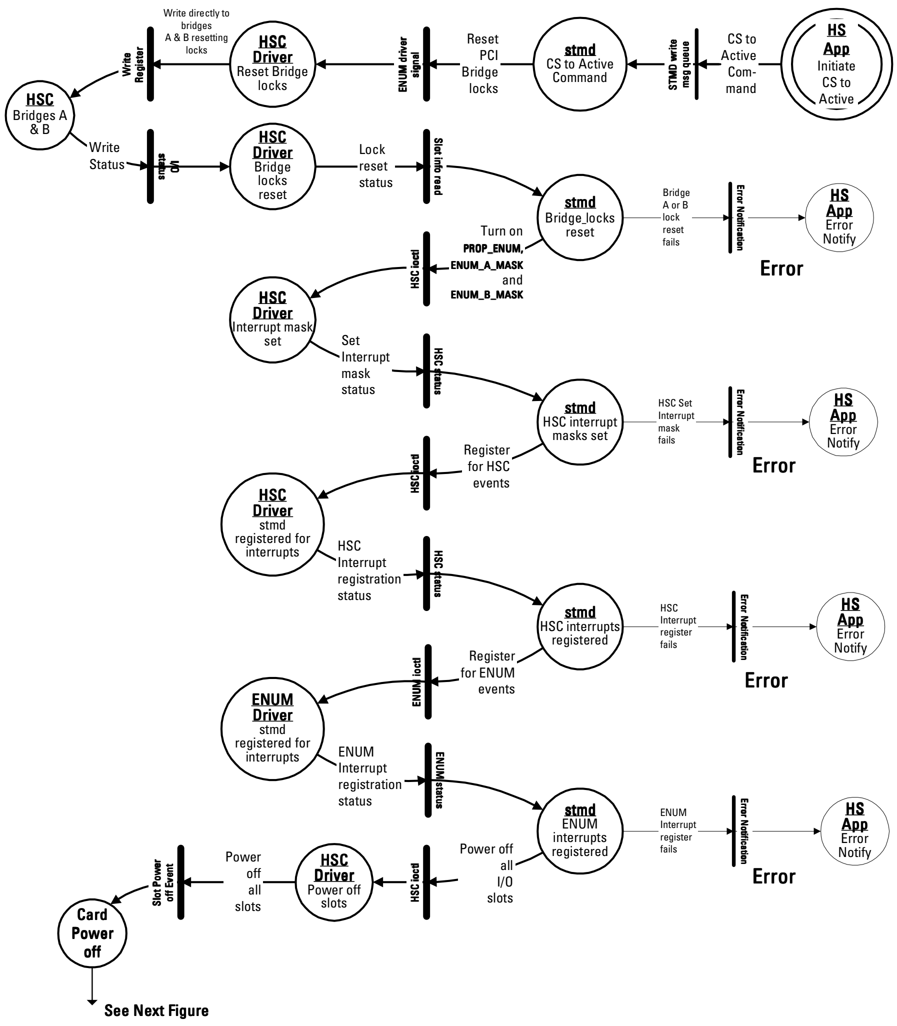

HS App Initiate CS to Active
CS to Active Command
STMD write msg queue
stmd CS to Active Command
Reset PCI Bridge locks
ENUM driver signal
HSC Driver Reset Bridge locks
Write directly to bridges A & B resetting locks
Write Register
HSC Bridges A & B
Write Status
I/O status
HSC Driver Bridge locks reset
Lock reset status
Slot info read
stmd Bridge_locks reset
Bridge A or B lock reset fails
Error Notification
HS App Error Notify
Error
Turn on PROP_ENUM, ENUM_A_MASK and ENUM_B_MASK
HSC ioctl
HSC Driver Interrupt mask set
Set Interrupt mask status
HSC status
stmd HSC interrupt masks set
HSC Set Interrupt mask fails
Error Notification
HS App Error Notify
Error
Register for HSC events
HSC ioctl
HSC Driver stmd registered for interrupts
HSC Interrupt registration status
HSC status
stmd HSC interrupts registered
HSC Interrupt register fails
Error Notification
HS App Error Notify
Error
Register for ENUM events
ENUM ioctl
ENUM Driver stmd registered for interrupts
ENUM Interrupt registration status
ENUM status
stmd ENUM interrupts registered
ENUM Interrupt register fails
Error Notification
HS App Error Notify
Error
Power off all I/O slots
HSC ioctl
HSC Driver Power off slots
Power off all slots
Slot Power off Event
Card Power off
See Next Figure

Figure 4-8 part 2: Cold Standby to Active State Transition Diagram

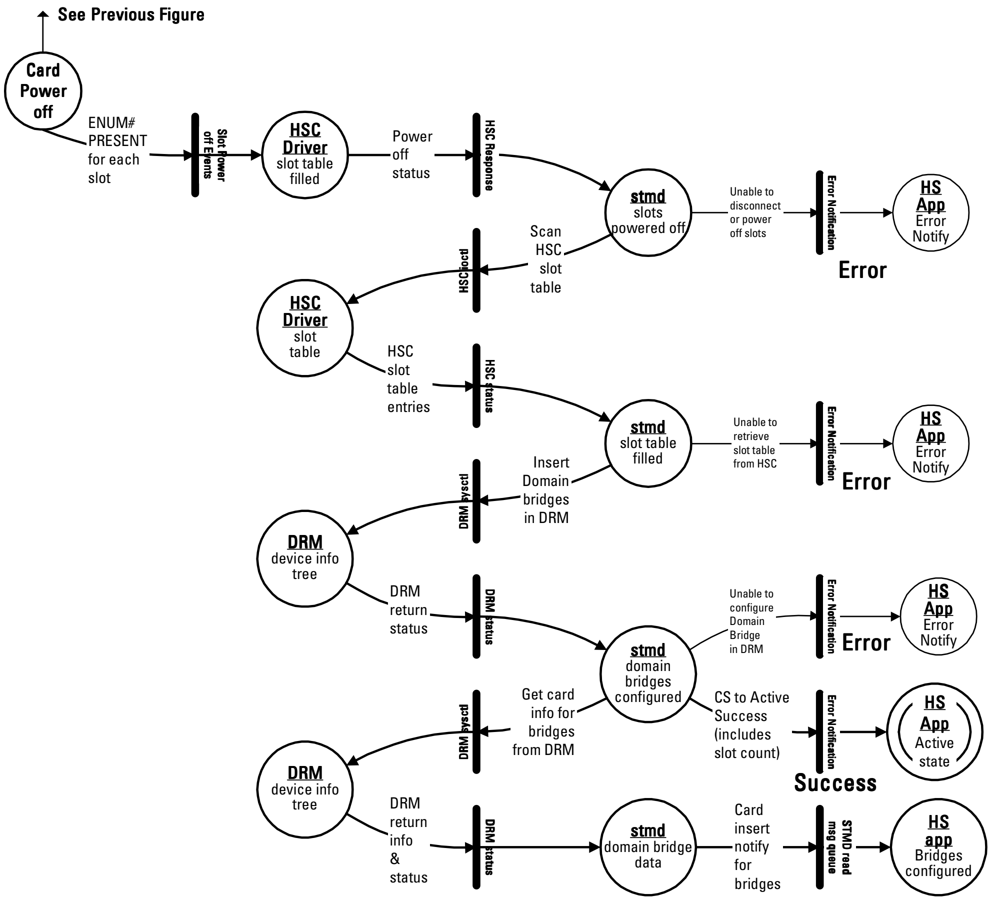


Figure 4-8 part 2: Cold Standby to Active State Transition Diagram

On receiving a `STMD_CS_TO_ACTIVE` message, the system:

- Enables arbitration
- Enables CompactPCI interrupts
- Sets `PROP_ENUM, ENUM_A_MASK`, and `ENUM_B_MASK` to 1
- Takes control of both domains
- Powers all slots on

- Registers for HSC events
- Registers with the ENUM driver
- Adds the enum driver signal and the HSC driver signal to the signal list
- Powers all slots off. Since the slots were previously powered on, this causes the PRESENT bit to raise an interrupt
- Scans all slots for the PRESENT bit. Builds slot mask and number of cards. This is sent with the return status in the STMD_CS_TO_ACTIVE response message
- Inserts nodes into the DRM tree for the local and remote domain bridges
- Sends the STMD_CS_TO_ACTIVE response message
- Sends STMD_CARD_INSERT messages for the domain bridges, if unable to get DRM node, sends STMD_ERROR_GET_DRM_INFO notification

Cold Standby to Warm Standby

The following state diagram describes the Cold Standby to Warm Standby domain state transition:

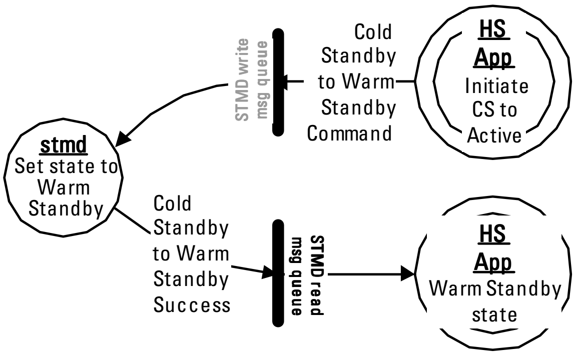

STMD write msg queue
Cold Standby to Warm Standby Command
HS App
Initiate CS to Active
stmd
Set state to Warm Standby
Cold Standby to Warm Standby Success
STMD read msg queue
HS App
Warm Standby state

Figure 4-9: Cold Standby to Warm Standby State Transition Diagram

This is completed on receiving a STMD_CS_TO_WS message.

There is no explicit Cold to Warm Standby transition. Slot and DRM information is passed in parts from Active to Standby. The Active side notifies the Standby side that it has the entire static configuration and that it must now transition into the Standby state. It uses the message STMD_CS_TO_WS to do this. The Standby STMD then sets an internal variable to indicate that it is in Warm Standby state and sends a `STMD_CS_TO_WS` response with the return status.

Active to Warm Standby

Figure 4-10 part 1 describes the Active to Warm Standby domain state transition:

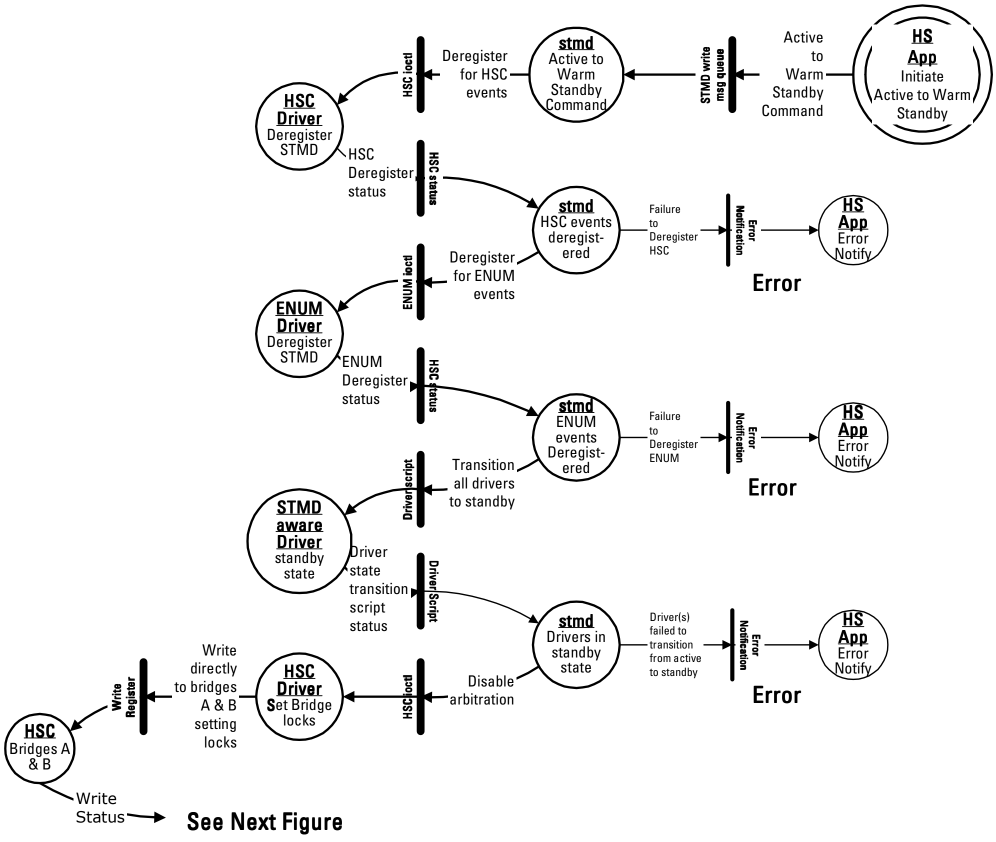

HS App Initiate Active to Warm Standby
Active to Warm Standby Command
STMD write msg queue
stmd Active to Warm Standby Command
Deregister for HSC events
HSC ioctl
HSC Driver Deregister STMD
HSC Deregister status
HSC status
stmd HSC events deregist-ered
Failure to Deregister HSC
Error Notification
HS App Error Notify
Error
Deregister for ENUM events
ENUM ioctl
ENUM Driver Deregister STMD
ENUM Deregister status
HSC status
stmd ENUM events Deregist-ered
Failure to Deregister ENUM
Error Notification
HS App Error Notify
Error
Transition all drivers to standby
Driver script
STMD aware Driver standby state
Driver state transition script status
Driver Script
stmd Drivers in standby state
Driver(s) failed to transition from active to standby
Error Notification
HS App Error Notify
Error
Disable arbitration
HSC ioctl
HSC Driver Set Bridge locks
Write directly to bridges A & B setting locks
Write Register
HSC Bridges A & B
Write Status
See Next Figure

Figure 4-10 part 1 : Active to Warm Standby State Transition

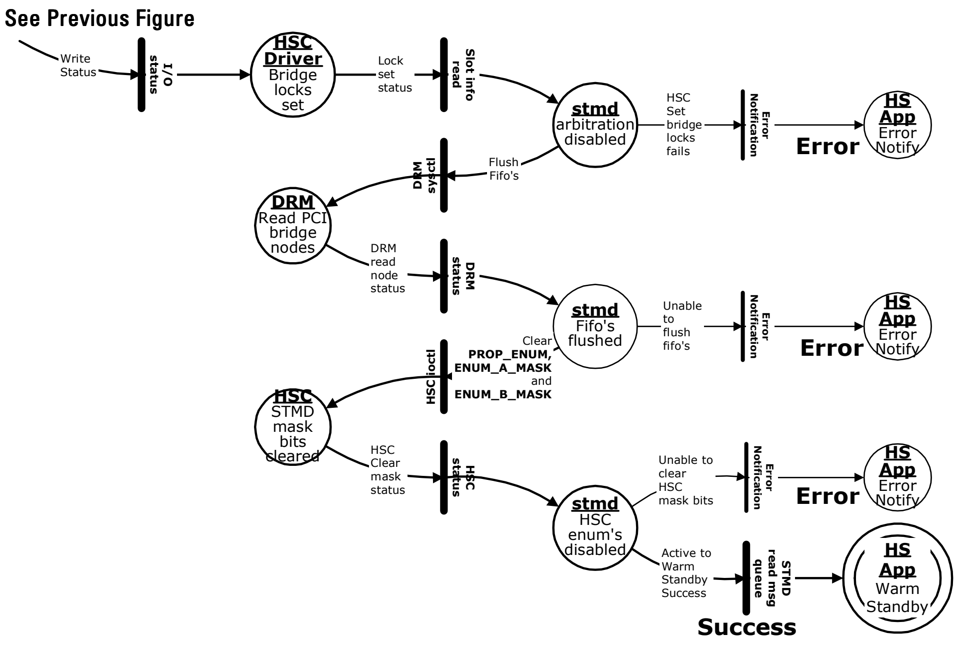


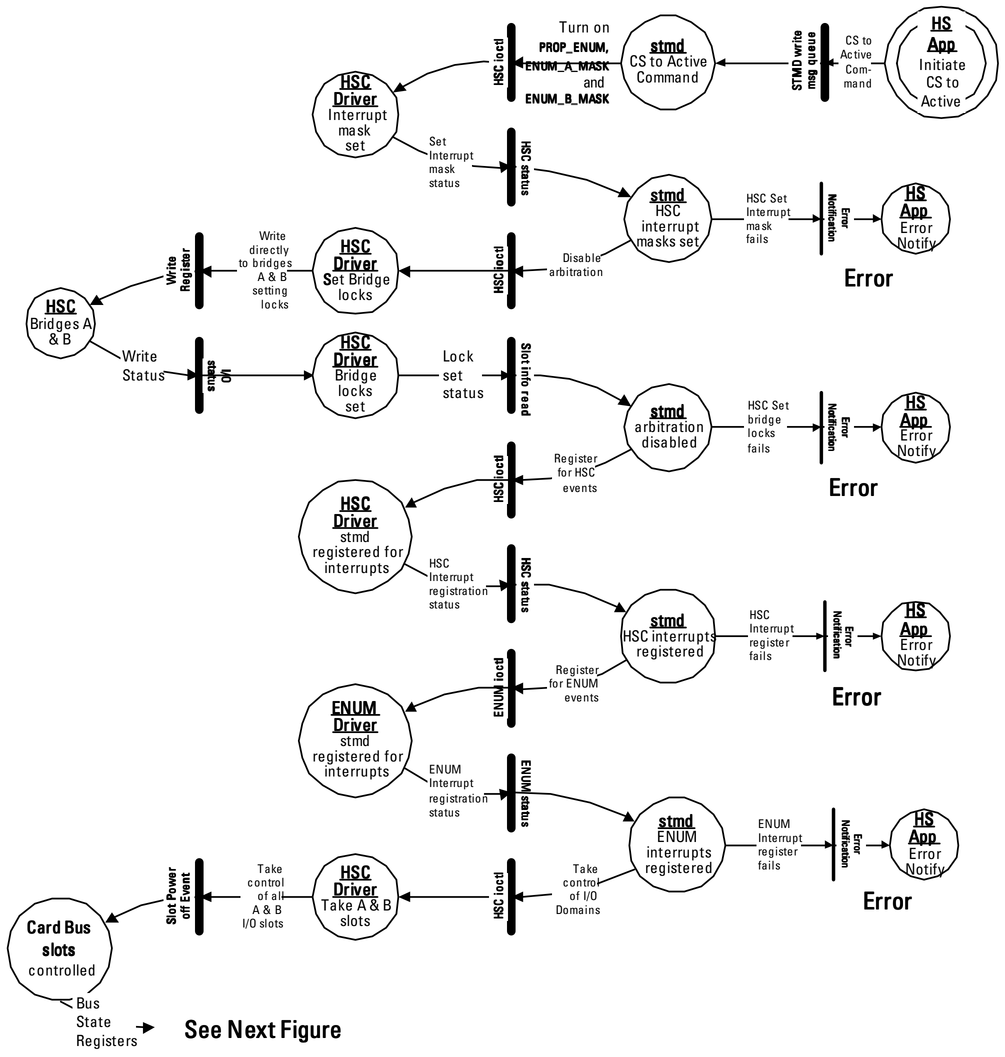
Figure 4-10 part 2 : Active to Warm Standby State Transition

The system performs the following on receipt of STMD_ACTIVE_TO_WS message:

1. Deregisters for HSC events
2. Deregisters for ENUM events
3. Removes the HSC driver signal and the ENUM driver signal from the signal list
4. Quiesces the list of installed drivers by calling the standby script for each driver
5. Disables arbitration
6. Flushes the FIFOs in all the PCI-to-PCI bridges by traversing the DRM tree and reading the bridge's config space
7. Sends a STMD_ACTIVE_TO_WS response with the return status
8. Sets PROP_ENUM, ENUM_A_MASK, and ENUM_B_MASK to 0

Warm Standby to Active

The following diagram describes the Warm Standby to Active domain state transition::



HS App Initiate CS to Active
CS to Active Command
STMD write msg queue
stmd CS to Active Command
Turn on PROP_ENUM, ENUM_A_MASK and ENUM_B_MASK
HSC ioctl
HSC Driver Interrupt mask set
Set Interrupt mask status
HSC status
stmd HSC interrupt masks set
HSC Set Interrupt mask fails
Error Notification
HS App Error Notify
Error
Disable arbitration
HSC ioctl
HSC Driver Set Bridge locks
Write directly to bridges A & B setting locks
Write Register
HSC Bridges A & B
Write Status
I/O status
HSC Driver Bridge locks set
Lock set status
Slot info read
stmd arbitration disabled
HSC Set bridge locks fails
Error Notification
HS App Error Notify
Error
Register for HSC events
HSC ioctl
HSC Driver stmd registered for interrupts
HSC Interrupt registration status
HSC status
stmd HSC interrupts registered
HSC Interrupt register fails
Error Notification
HS App Error Notify
Error
Register for ENUM events
ENUM ioctl
ENUM Driver stmd registered for interrupts
ENUM Interrupt registration status
ENUM status
stmd ENUM interrupts registered
ENUM Interrupt register fails
Error Notification
HS App Error Notify
Error
Take control of I/O Domains
HSC ioctl
HSC Driver Take A & B slots
Take control of all A & B I/O slots
Slot Power off Event
Card Bus slots controlled
Bus State Registers
See Next Figure

Figure 4-11 part 1 : Warm Standby to Active State

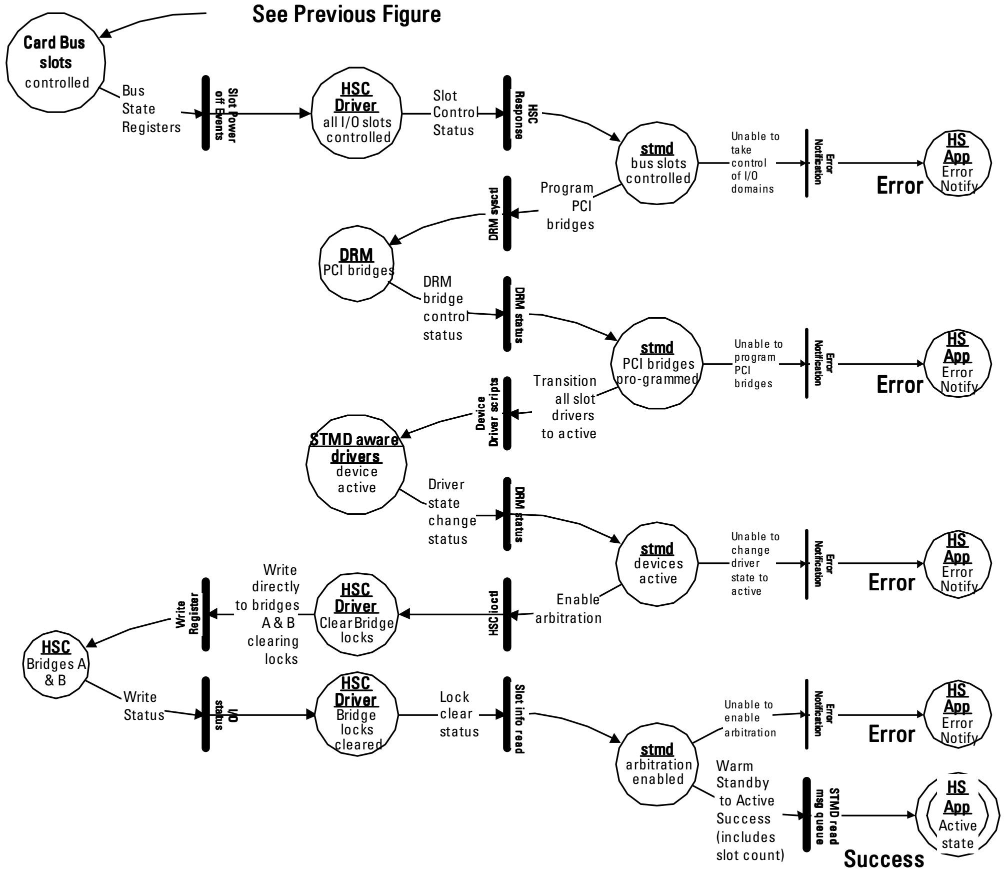


Figure 4-11 part 2 : Warm Standby to Active State

The system performs the following on receipt of `STMD_WS_TO_ACTIVE` message:

1. Sets `PROP_ENUM`, `ENUM_A_MASK`, and `ENUM_B_MASK` to 1
2. Disables arbitration
3. Disables interrupts from the CompactPCI domains by turning them off in the `MPIC`
4. Registers for `HSC` events. This is to capture `PRESENT` events
5. Registers for `ENUM` events

6. Adds the `HSC` driver signal and the `ENUM` driver signal to the signal list
7. Takes control of both domains. The domain bridges now become visible
8. Programs the domain bridges with the information already in the `DRM` nodes that were pushed before
9. Goes through the list of installed drivers and enables all of them by calling the active script for each driver
10. Enables arbitration
11. Enables interrupts from the CompactPCI domains by turning them on in the MPIC
12. Sends a `STMD_WS_TO_ACTIVE` response

NOTE: If any errors occur during the transition from Warm Standby to Active, the system will be left in an undefined state. No attempt is made to restore the system to its original state.

Hot Insertion

Hot Insertion can take these different forms:

- Bottom Up Hot Insertion on the Active Domain
- Application initiated Hot Insertion on a Warm Standby Domain

Bottom Up Hot Insertion on the Active Domain

The state transition for this transition are presented in Figure 4-5 part 1, “Successful Hot Insertion State,” on page 50.

1. `STMD` gets a signal from the `ENUM` driver; this can happen only on the active side.
2. Get the event from the `ENUM` driver.
3. Check the `HEALTHY` bit for the slot; if it is not set, power down the slot, set the slot status to `CARD_UNHEALTHY`, send a `STMD_CARD_UNHEALTHY` message and return.
4. Have `DRM` find all the devices and bridges in the slot by calling `CMD_PROBE`.

5. Starting with the root of the subtree corresponding to the card in the slot, do the following for each node in the tree:
   - `CMD_SELECT` and `CMD_ALLOC` the node.
   - Install the driver corresponding to the node by calling the driver installation script.
   - Bring the driver to standby by calling the driver standby script.
   - Enable the driver by calling the driver active script.
   - Update the `drv_info` structure.
   - Update the slot table.
6. Set the slot state to `CARD_INSERTED`.
7. Send a `STMD_CARD_INSERT` message to `stmd_msgq_rd`.

Application-Initiated Hot Insertion on a Warm Standby Domain

Because there can be no bottom up hot insertion on the Standby Side, insertion must be initiated by the application itself. The following shows the state transitions requirements:

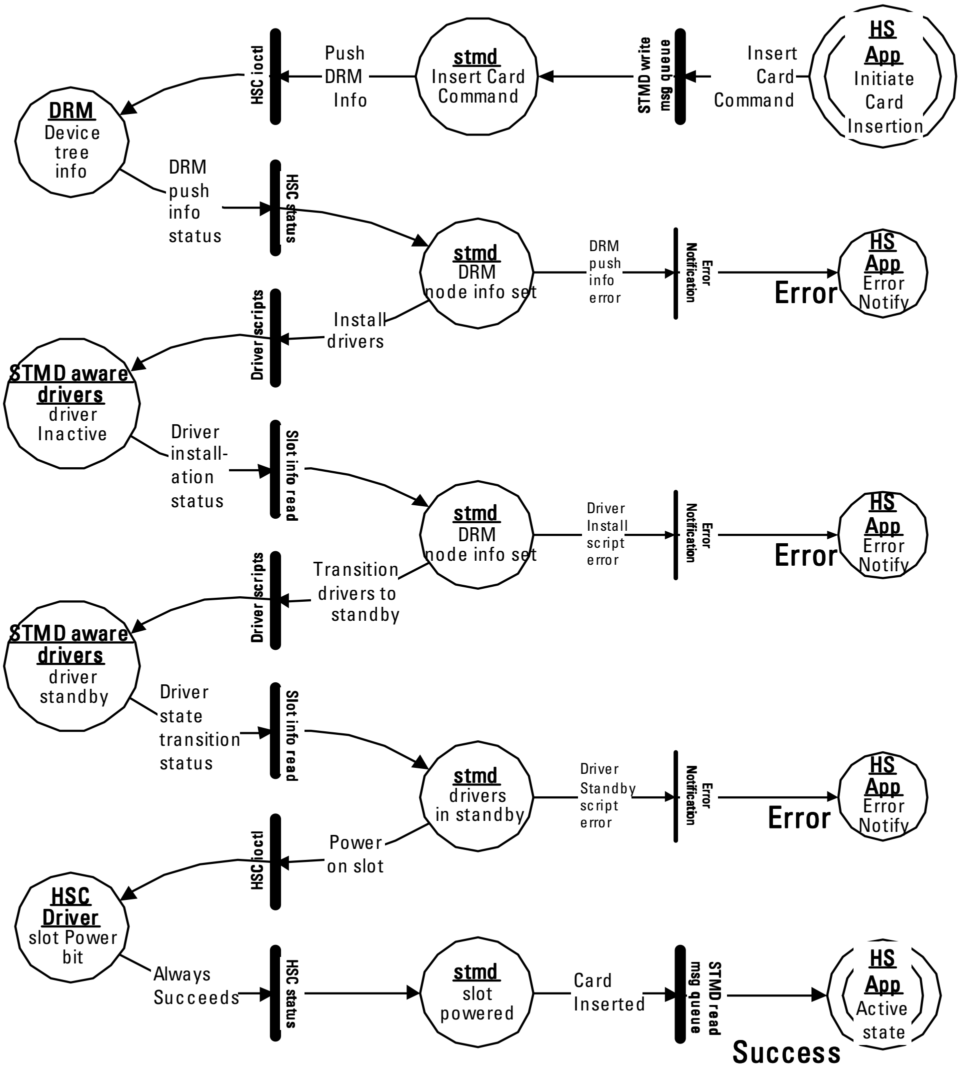


Figure 4-12: Application-Initiated Hot Insertion

These steps are completed on receiving a `STMD_CARD_INSERT` command message:

1. Update the slot table entry
2. Call `push_drm_info()` to push the `DRM` node into the standby tree. `push_drm_info()` returns the root of the pushed subtree
3. Starting with the root of the subtree corresponding to the card in the slot, do the following for each node in the tree:
   - Look for a driver for the `vendor_id`, `device_id` pair in the node. Install the driver corresponding to it. Execute the standby script in order to bring the driver to standby state
   - Update the `drv_info` structure
   - Update the slot table
4. Send a `STMD_CARD_INSERT` response with the return status

Hot Extraction

Hot Extraction can take these different forms:

- Bottom Up Hot Extraction on the Active Domain.
- Application initiated Hot Extraction on a Warm Standby Domain.

Bottom Up Hot Extraction on the Active Domain

The state transitions for this transition is presented in Figure 4-6 part 1, “Card Extraction State,” on page 52.

1. `STMD` gets a signal from the `ENUM` driver. This can happen only on the active side.
2. Get the event from the `ENUM` driver.
3. Starting with the root of the subtree corresponding to the card in the slot, do the following for each node in the tree:
   - Call driver standby script. Wait for the response. If error, flag an asynchronous error `STMD_ERROR_QUIESCE` and return.
   - Uninstall the driver by calling the driver uninstallation script.
   - Update `drv_info`.

4. Prune the `DRM` subtree.
5. Set the slot state to `CARD_EXTRACTED`.
6. Update the slot table.
7. Turn off power to the slot.
8. Send message `STMD_CARD_REMOVE`.

Application-Initiated Hot Extraction on a Warm Standby Domain

Because there can be no bottom up Hot Extraction on the Standby Side, extraction must be initiated by the application itself. The following figure shows the required state transitions:

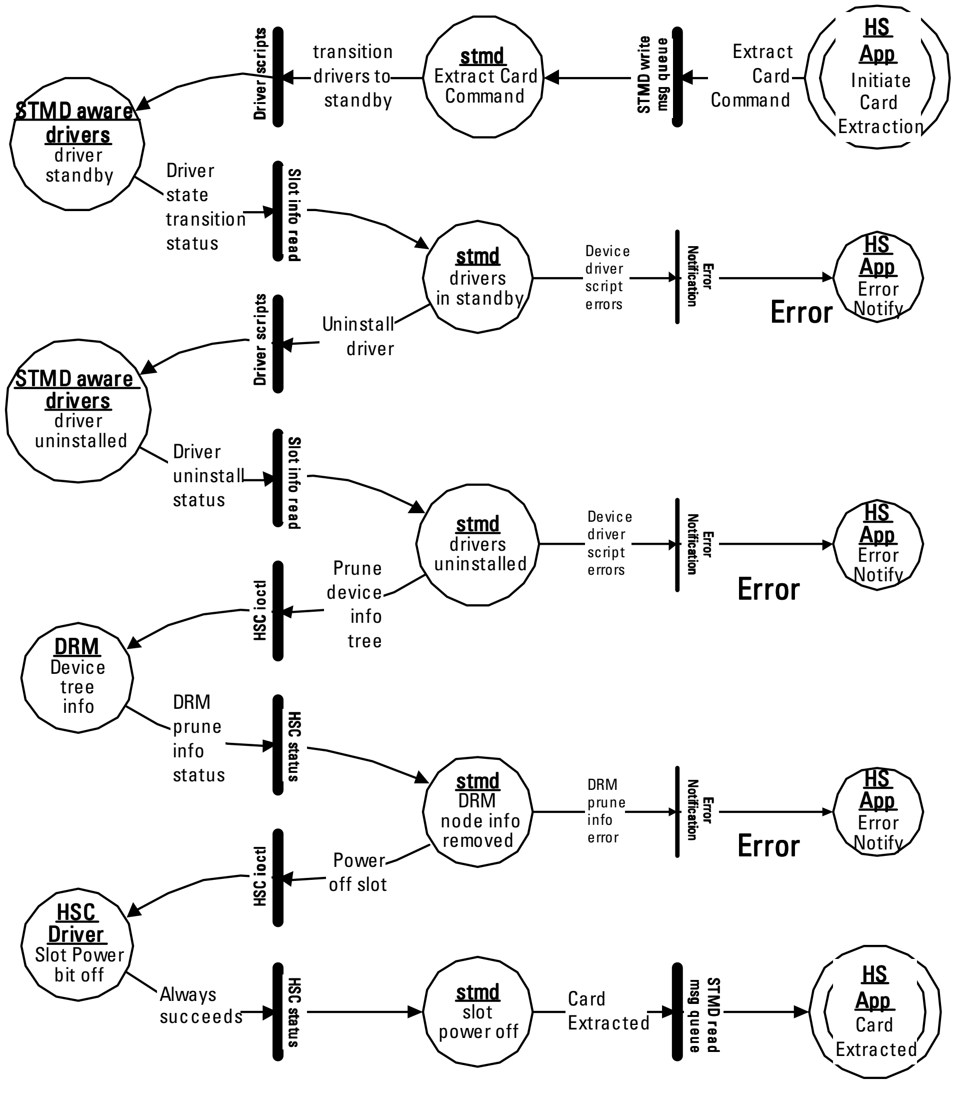


Figure 4-13: Application-Initiated Hot Extraction

These steps are completed on receiving a STMD_CARD_REMOVE command message:

1. Starting with the root of the subtree corresponding to the card in the slot, do the following for each node in the tree:
   - Uninstall the driver.
   - Update the drv_info structure.
2. Use the slot table entry with CMD_PRUNE to delete the DRM subtree corresponding to the line card that is extracted.
3. Update the slot table entry.
4. Send a STMD_CARD_REMOVE response with the return status.
5. For an unrecognizable message, send the message STMD_UNKNOWN_MESSAGE.

Asynchronous Events

There are certain events (other than bottom up Hot Insertion and Extraction) that alter system state and generate asynchronous notification.

HSC Events

The STMD HSC event handler gets called upon receiving a signal from the HSC driver. The HSC Event Handler:

1. Receives the event from the event queue.
2. If it is not a PRESENT event, discards it.
3. If the slot state is CARD_PRESENT or CARD_INSERTED, flags an asynchronous error STMD_ERROR_INTERNAL.
4. If the slot state is CARD_EXTRACTED and the event is PRESENT, ON, ignores it. When power is turned off to the slot, the PRESENT bit changes from 0 to 1 and generates an interrupt, but this is not a true PRESENT event, therefore it is ignored.
5. If the slot state is CARD_ABSENT and the event is PRESENT, ON, sets slot state to CARD_PRESENT and turns on power to the slot.
6. If the slot state is CARD_EXTRACTED and the event is PRESENT, OFF, sets slot state to CARD_ABSENT.

7. If the slot state is `CARD_ABSENT` and the event is `PRESENT`, `OFF`, flags an asynchronous error `STMD_ERROR_SPURIOUS_PRESENT.`

APPENDIX A Basic Terminology

These definitions and descriptions are derived from section 2.1.4 “Additional Terms” from the Compact PCI Hot Swap Specification R1.0 and several additional sources indicated in footnotes.[1]

Back End Logic

The portion of a Compact PCI board that is isolated from the system until the Hardware Connection Process is complete.

Back End Power

The power to the back end logic of a Compact PCI board. Back end power is applied to the back end logic through the power isolation circuitry.

Dynamic Configuration

A process whereby a hot swap board is allocated system resources (enumerated) by the system software following insertion of the board. In non hot swap systems, enumeration only takes place on system boots. The same resources are released prior to extraction of the board.

Enumeration

The action taken by the Compact PCI system host to poll the configuration spaces of all the PCI devices and assign (or release) the necessary resources (memory, I/O address space, interrupts, drivers).

1. Compact PCI Hot Swap Specification 2.1 R1.0 August 3, 1998 PCI Industrial Computers Manufacturer’s Group (PICMG) 1997. 1998

Fault Tolerance

A Fault Tolerant Computer System is a system that can continue to operate reliably by producing acceptable outputs in spite of occasional occurrences of component failures in both hardware and software components.[1]

One of the principal concepts of Fault Tolerance is that of handling a fault as additional system state.

Fault Tolerant computer systems can be created using two or more conventional computers that duplicate all processing, or having one system standby if the other fails. They can be built from the ground up from commercially available redundant processors. Such systems have several processors, control units, peripheral units and power supplies / sources combined into a modular integrated system.

Fault Tolerance has even been extended into the realm of handling deliberate sabotage; this is the so-called byzantine fault. Obviously, anticipating, detecting and handling this type of fault is very involved.

It is precisely the nature of these Fault Detection and Management systems, involving such things as multiple copy elections and lock step operation that seriously impacts system performance — even the time bounded qualities required of Real-Time systems.

Hardware Fault Tolerance has been matured for these reasons:

- Hardware cost has become relatively insignificant in comparison with overall system cost and as a result, hardware redundancy is easily justified.

Systems vendors have provided platforms with substantial fault tolerance capabilities.

The primary beneficiaries of traditional Fault Tolerance are financial applications. Because of the extreme cost and lack of COTS support, such systems have not been practical in most other applications.

1. “Design of Real-Time Fault-Tolerant Computing Stations” by K. H. Kim, University of California at Irvine from Real Time Computing NATP ASI Series ISBN 3-540-57558-8 Springer Verlag 1994

High Availability

The attribute of a system designed to keep running (maintain availability) in the event of a system component failure, or preventative action. To provide a higher degree of availability, a system requires a higher degree of control.

Hot Swap

The idea behind Hot Swap is to allow the orderly insertion and extraction of boards (usually in reference to Compact PCI boards) without adversely affecting system operation.

Hot Swap is described in terms of three processes:

- *Physical Connection Process*,

  Includes:

  - Hot Insertion (by which a board is installed in a live system).
  - Hot Extraction (by which a board is removed from a live system).

- *Hardware Connection Process*

  Describes the electrical connection and disconnection of hardware to a live system.

- *Software Connection Process*

  Describes the connection and disconnection of the software layer(s) to a live system.

There are four different degrees of Hot Swap capability. These are:

- Non Hot Swap
- Basic Hot Swap
- Full Hot Swap
- High Availability

For more detail on these different Hot Swap capabilities, see “Hot Swap” on page 4.

PCI Extended Capabilities Pointer (ECP)

A pointer to a linked list of additional configuration space registers. The mechanism allows additional configuration space registers to be added. A hot swap control and status register is added using the ECP mechanism to bring signals to I/O transition boards or for other auxiliary buses. These signals can also be used for additional hot swap and HA support.

PCI Mezzanine Card (PMC)

PMCs are modules which are modules installed on a Hot Swap Module PMC expansion Carrier or on the system controller CPU board. They provide additional I/O capabilities.

Quiesced

No operations are in progress or pending and there is no authorization to launch new operations.

Warm Domain Switchover

In contrast to Hot Swap, Warm Domain Switchover involves switching from one processor domain in the same chassis again to another processor domain, without seriously affecting overall system performance.

APPENDIX B *Motorola Hot Swap Controller/ Bridge API*

This Appendix is provided by Motorola, and details the Hot Swap Controller/Bridge API.

Introduction

This document addresses the Motorola Hot Swap Controller/Bridge Driver incorporated into various Motorola supported operating systems. This driver is required to manage the Motorola Compact PCI CPX82xx Computer system chassis and functions.

Definitions

Mesquite	Code name for the MCP750 single board computer.
DDI	Device Driver Interface
API	Application program interface. Equivalent to the DDI.
CPX8216	Dual-Mesquite capable cPCI computer w/ 12 payload card slots.
CPX8216A	Dual-Mesquite capable cPCI computer w/ 12 payload card slots and ATM bus.
CPX8216T	Dual-Mesquite capable cPCI computer w/ 12 payload card slots and H110 Reset.
CPX8221	Dual-Mesquite capable cPCI computer w/ 17 payload card slots.
HSC/B	Hot Swap Controller and Bridge board.
HSCD	Hot Swap Controller/Bridge Driver.

CPX82xx HA Programmable Resource Management

Purpose of this Document

This document describes the Application Program Interface(API), a collection of C-based library functions, which are used to manage the resources of Motorola's CPX8216, CPX8216A, CPX8216T and CPX8221 High Availability(HA), Hot Swapable, Compact PCI(cPCI) computer systems.

Where differences in hardware exist between these two systems, a notation of (CPX8216), (CPX8216A), (CPX8216T) or (CPX8221) will indicate that the resource is available to that hardware platform only.

Overview

A Set Of Manageable System Resources

All hardware resources addressed by the HA Resource Management API reside in some variant of the CPX8216 or CPX8221 system rack. These resources include four drive/tape peripheral bays (CPX8216), three power supplies, three cooling fans, two processors--each with its own bridge card, two extension bridges (CPX8221), eeprom programmer, system LEDs, four alarms, two hot swap controllers, and two buses with 6+6 cPCI payload card slots (CPX8216) or three buses with 6+6+5 cPCI payload card slots (CPX8221), respectively. Managing these resources allow user applications to provide the HA environment which keeps the work flow going, even if performance might, to some extent, be degraded temporarily. The software managing the HA environment facilitates, through finer gradation of control, the coordinated swapping of defective, new or upgraded boards and their drivers, fail over to redundant hardware, or even new OS or eeprom, software.

Modifiable Attributes for Each Resource

Each resource has a set of attributes which can be altered, i.e. LEDs being turned *on*, or power being turned *off*, etc. When this purposeful action is taken, the attribute's state is being changed. Altering one state may precipitate a different attribute's state to transition--change-- asynchronously. For example, changing the *power* attribute on a payload slot from *off* to *on* **may** result in the transition of the *healthy* attribute's state from *off* to *on*. A power loss to a payload slot would obviously cause the reverse transition of *healthy* to occur. In this example, note that the *healthy* attribute itself cannot be changed directly, but represents a new state that happens as a result of some other event. This type of attribute is classed as *non-modifiable*, and can only be *status'd*.

Whether scheduled (state changed by application code) or unscheduled (state transitions asynchronously), these *events* are available to a user's application through a function call which uses a signaling scheme. In this manner, the HA application(s) can monitor what other software is doing with the resources in addition to being notified of 'unexpected' events which indicate a change in status for any particular resource. Requesting these event notifications makes the application a *subscriber* as opposed to simply being a one-way *user* of the interface.

As of now, this interface will support 10 simultaneous subscribers, but an unlimited number of users (dependent only upon operating system limits and configuration). The number of event subscribers permitted can be adjusted higher or lower via a #define in the system header file *hscd.h*, which is currently not part of the distribution. Until end user modifyable, a request can be made of MCG Engineering to effect a change in this value.

Valid Attribute Values

Modifiable attributes may be set to *on* or set to *off*. In the case of LEDs, a third setable state is *blink*. Non-modifiable attributes, which transition asynchronously, cannot be set, but can be *status'd*. Naturally, all attributes of any particular resource may be *status'd* to determine their current states.

Brief Summary

The CPX82xx platform hardware resources are managed through the use of an API comprised of several C-language functions collected into a library accessible to the HA application software. Each software resource has a set of attributes which may be disabled or enabled through the use of this API, as well as a group of attributes whose states cannot be set, but which transition asynchronously in response to other system resource states. In either case, changes to ALL resource attributes result in events which may be received by the HA applications to assist in their HA environment's control.

All hardware addressed by this software resides in some variant of the CPX8216 or CPX8221 system cabinet. This includes peripheral drive bays (CPX8216 only), power supplies, fans, processors, extension bridges (CPX8221 only) and Compact PCI slots with boards.

Accessing CPX System API's

The software described in this document resides in a LynxOS, or other OS compatible library and is linked with the user's own applications. A single header file--cpxapi.h-- needs to be #include'd in your application's sources for successful compilation. The locations of this software is as follows:

- header - `cpxapi.h` in `/hasw/include`
- `library - libcpxapi.a or libcpxapi.o` in `/hasw/cpx`
- tool - `cpxtool.o` in `/hasw/cpx`
- driver `cpxHSCD` (LynxOS)
- device - `hscdev` (LynxOS binary file)

All functions and enumerations/defines are preceded by `cpx` or `hsc_`, trigraphs representing the compatible family of hardware it is used to manage. This is done so that no ambiguity or overlap occurs with other OS header files and libraries.

CPX82xx System Resource and Attribute Identifiers

Access to the CPX8000's hot swap controller is done through a functional interface. The enumerations found in this document's tables must be used as arguments for the various functions to control that interface. The hot swap controller/bridge permits control of the processor, bridge and cPCI payload slots [16 (12 payload) in the CPX8216 & 21 (17payload) in the CPX8221] and system functions such as power supplies, alarms and bus control options, etc.

Required Enumerations

The following enumerations are necessary for using the functions discussed later in this document. Virtually all enumerations come under the heading of command, status, and identification. Many are equivalenced for code readability, but are logically the same.

hsc_RESOURCE_ID Enumerations

For identification, the following register resource enumerations are available. Bold entries indicate either CPX8216 or CPX8221 specific implementations. *Note that these enumerations apply to* `hsc_`*-style API calls only.*

Table B-1: hsc_RESOURCE_ID Enums

Slot/Subsystem typedef:	Equivalent	Description	In Domain
`hsc_SLOT_01`	N/A	non-host cPCI slot	A
`hsc_SLOT_02`	N/A	non-host cPCI slot	A
`hsc_SLOT_03`	N/A	non-host cPCI slot	A
`hsc_SLOT_04`	N/A	non-host cPCI slot	A
`hsc_SLOT_05`	N/A	non-host cPCI slot	A
`hsc_SLOT_06`	N/A	non-host cPCI slot	A
`hsc_SLOT_07`	`hsc_PROC_A`	host processor slot	A
`hsc_SLOT_08`	`hsc_BRIDGE_B`	bridge to B	B
`hsc_SLOT_09`	`hsc_PROC_B`	host processor slot	B
`hsc_SLOT_10`	`hsc_BRIDGE_A`	bridge to A	A
`hsc_SLOT_11`	N/A	non-host cPCI slot	B
`hsc_SLOT_12`	N/A	non-host cPCI slot	B
`hsc_SLOT_13`	N/A	non-host cPCI slot	B
`hsc_SLOT_14`	N/A	non-host cPCI slot	B
`hsc_SLOT_15`	N/A	non-host cPCI slot	B
`hsc_SLOT_16`	N/A	non-host cPCI slot	B
`hsc_SLOT_17`	N/A	non-host slot **(CPX8221)**	B**(bus C)**
`hsc_SLOT_18`	N/A	non-host slot **(CPX8221)**	B**(bus C)**
`hsc_SLOT_19`	N/A	non-host slot **(CPX8221)**	B**(bus C)**
`hsc_SLOT_20`	N/A	non-host slot **(CPX8221)**	B**(bus C)**
`hsc_SLOT_21`	N/A	non-host slot **(CPX8221)**	B**(bus C)**
`hsc_PS_1`	N/A	Power Supply 1	A
`hsc_PS_2`	N/A	Power Supply 2	A
`hsc_PS_3`	N/A	Power Supply 3	A
`hsc_PBAY_1`	N/A	Peripheral Bay 1 **(CPX8216)**	A and/or B

Table B-1: hsc_RESOURCE_ID Enums (Continued)

Slot/Subsystem typedef:	Equivalent	Description	In Domain
hsc_PBAY_2	N/A	Peripheral Bay 2 **(CPX8216)**	A and/or B
hsc_PBAY_3	N/A	Peripheral Bay 3 **(CPX8216)**	A and/or B
hsc_PBAY_4	N/A	Peripheral Bay 4 **(CPX8216)**	A and/or B
hsc_BUS_A	N/A	Bus Control Reg A	N/A
hsc_BUS_B	N/A	Bus Control Reg B	N/A
hsc_BUS_C	N/A	Bus C **(CPX8221 only)**	N/A
hsc_EXT_BRIDGE_1	N/A	Extension Bridge to C-bus **(CPX8221)**	N/A
hsc_EXT_BRIDGE_2	N/A	Extension Bridge to C-bus **(CPX8221)**	N/A
hsc_ALARM_CTRL	N/A	Alarm Control Reg	A
hsc_EEPROM_CTRL	N/A	EEPROM Control Reg	local
hsc_INT_MASK	N/A	Interrupt Mask Reg.	N/A
hsc_SYS_LED	N/A	System LED Reg.	A
hsc_INT_STAT_MASK	N/A	All pending interrupts	N/A
hsc_INT_STAT_A	N/A	PCI_A active ints	N/A
hsc_INT_STAT_B	N/A	PCI_B active ints	N/A
hsc_INT_STAT_C	N/A	PCI_C active ints	N/A
hsc_ATM	N/A	ATM BUS control Reg. **(CPX8216A)**	N/A

cpx-Style Resource Enumerations

These enumerations are used with cpx-Style API calls *only* and have the ability to be *aggregated*. That is, they can be specified in groups of individual resources enumerations programatically, and arithmetically, *or*'d together when application of a change to one or more of their attributes is desired. For example, to connect payload slots one, two and four, the following call could be made:

```
cpxConnect(cpxSLOT1|cpxSLOT02|cpxSLOT04, cpxON);
```

Note that not all cpx-Style calls support aggregation. For example, *cpxTakeDomain()* takes arguments for specific resources specifying domain A, domain B or a combination of domains A & B--cpxDOMAIN_A, cpxDOMAIN_B and cpxDOMAIN_A_B respectively. Bold entries indicate either CPX8216 or CPX8221 specific implementations.

Table B-2: cpxRESOURCE Enums

cpx-Style Aggregatable Resources	Platforms
`cpxSLOT01`	All CPX8000-series
`cpxSLOT02`	
`cpxSLOT03`	
`cpxSLOT04`	
`cpxSLOT05`	
`cpxSLOT06`	
`cpxPROCA`	
`cpxPROCB`	
`cpxBRIDGEA`	
`cpxBRIDGEB`	
`cpxSLOT11`	
`cpxSLOT12`	
`cpxSLOT13`	
`cpxSLOT14`	
`cpxSLOT15`	
`cpxSLOT16`	
`cpxSLOT17`	**CPX8221**
`cpxSLOT18`	
`cpxSLOT19`	
`cpxSLOT20`	
`cpxSLOT21`	
`cpxALL_SLOTS`	Logical aggregate of all valid platform payload slots. All CPX8000-series

Table B-2: cpxRESOURCE Enums

cpx-Style Aggregatable Resources	Platforms
`cpxEB1`	**CPX8221**
`cpxEB2`	
`cpxSYSLED`	All CPX8000-Series
`cpxALARM`	
`cpxPS1`	
`cpxPS2`	
`cpxPS3`	
`cpxFAN1`	
`cpxFAN2`	
`cpxFAN3`	
`cpxBUS_A`	
`cpxBUS_B`	
`cpxBUS_C`	
`cpxPBAY1`	**CPX8216 only**
`cpxPBAY2`	
`cpxPBAY3`	
`cpxPBAY4`	

`hsc_ACTION` Enumerations

Bold entries indicate either CPX8216 or CPX8221 specific implementations.

Table B-3: hsc_ACTION Enums

hsc_ACTION typedef	Usage	Comments
NOTE: All action enumerations are *positive* in value. (see “hsc_Action” on page 155)		
`hsc_ON`	attribute change operations	also result of `hsc_STATUS` request
`hsc_OFF`	attribute change operations	also result of `hsc_STATUS` request
`hsc_STATUS`	attribute acquisition	
`hsc_WRITE`	*RegDirect()* or *RegByte()* write operation	also hsc_PUT (to become obsolete)
`hsc_READ`	*RegDirect()* or *RegByte()* read operation	also hsc_GET (to become obsolete)
`hsc_NCHG`	attribute operations - a result of `hsc_ON` or `hsc_OFF` if state already set	May occur in response to `hsc_ON` or `hsc_OFF` action. Never used as an action argument itself.

hsc_DOMAIN Enumerations

Bold entries indicate either CPX8216 or CPX8221 specific implementation

Table B-4: hsc_DOMAIN Enums

Enumeration	Meaning	Bus’s
`hsc_DOMAIN_A`	A	A
`hsc_DOMAIN_B`	B	B (& C if CPX8221)
`hsc_DOMAIN_A_B`	A & B	A, B (& C if CPX8221)
`hsc_DOMAIN_THIS`	Host’s Own Domain	obsolete - removed

cpx-Style Domain Enumerations

Bold entries indicate either CPX8216 or CPX8221 specific implementation.

Table B-5: cpx-Style Domain Enums

Enumeration	Meaning	Bus's
cpxDOMAIN_A	A	A
cpxDOMAIN_B	B	B (& C if CPX8221)
cpxDOMAIN_A_B	A & B	A, B (& C if CPX8221)
cpxDOMAIN_THIS	Host's Own Domain	(to become obsolete)

hsc_ATTRIBUTE Enumeration

Bold entries indicate either CPX8216 or CPX8221 specific implementations.s

Table B-6: hsc_ATTRIBUTE Enums

hsc_ATTRIBUTE typedef	Comment	Status
These objects can be used to set, clear or obtain status		trigraph hsc_
hsc_SOFT_XFR	enable/disable soft transfer for Bus A or Bus B	ON/OFF
hsc_XFR_CTL1	Bus control register bus transition	ON/OFF
hsc_XFR_CTL2	Bus control register bus transition	ON/OFF
hsc_LOCK	control of Bus access by Payload slot cards	ON/OFF
hsc_POWER	power enable/disable	ON/OFF
hsc_CONNECT	board connect/disconnect	ON/OFF
hsc_CONNECT_CTRL	extension bridge for bus C -- write only **(CPX8221)**	ON/OFF
hsc_LED_1	LED enable /disable	ON/OFF
hsc_LED_2	LED enable/disable	ON/OFF
hsc_LED_3	LED enable/disable	ON/OFF
hsc_LED_4	LED enable/disable	ON/OFF
hsc_PS_LED_1	LED enable /disable	ON/OFF
hsc_PS_LED_2	LED enable/disable	ON/OFF
hsc_FAN_LED_1	LED enable/disable	ON/OFF

Table B-6: hsc_ATTRIBUTE Enums (Continued)

hsc_ATTRIBUTE typedef	Comment	Status
`hsc_FAN_LED_2`	LED enable/disable	ON/OFF
`hsc_MINOR_ALARM`	alarm enable/disable	ON/OFF
`hsc_MAJOR_ALARM`	alarm enable/disable	ON/OFF
`hsc_CRIT_ALARM`	alarm enable/disable	ON/OFF
`hsc_RACK_ALARM`	alarm enable/disable	ON/OFF
`hsc_FAN_LOW`	power supply fan speed control	ON/OFF
`hsc_FORCE_HEALTHY`	use hsc_ power value as the healthy state	ON/OFF
`hsc_INT_STATUS`	slot status has changed	ON/OFF
`hsc_IMASK_A`	interrupt mask for PCI A interrupt	ON/OFF
`hsc_IMASK_B`	interrupt mask for PCI B interrupt	(no support)
`hsc_IMASK_C`	interrupt mask for PCI C interrupt	(no support)
`hsc_ENUM_A_MASK`	enable/disable ENUM interrupts for Domain A	ON/OFF
`hsc_ENUM_B_MASK`	enable/disable ENUM interrupts for Domain B	ON/OFF
`hsc_PROP_ENUM`	Propagate ENUMS between domains?	ON/OFF
`hsc_INSERTION`	HSC/Bridge card inserted?	ON/OFF
`hsc_REMOVAL`	HSC/Bridge cards only (cleared by SW)	ON/OFF
`hsc_EEPROM_REST_OE`	eeprom reset and output enable	ON/OFF
Objects which can be used to acquire status **only**		
`hsc_ENUM_A_STATE`	ENUM in Domain A currently asserted?	ON/OFF
`hsc_ENUM_B_STATE`	ENUM in Domain B currently asserted?	ON/OFF
`hsc_PRESENT`	board present? ALL slots!	ON/OFF
`hsc_PS_PRESENT`	power supply present?	ON/OFF
`hsc_EJECTOR`	Ejector handles on CPU/Bridge lock state	ON/OFF
`hsc_OWN_DEVICE`	peripherals only; set by present, cleared by reset	ON/OFF
`hsc_POWER_GOOD`	power supplies only	ON/OFF
`hsc_INSTALLED`	for HSC/Bridge card	ON/OFF
`hsc_COOLING_ALARM`	power supplies only	ON/OFF

Table B-6: hsc_ATTRIBUTE Enums (Continued)

hsc_ATTRIBUTE typedef	Comment	Status
`hsc_COOLING_FAULT`	power supplies only - critical	ON/OFF
`hsc_XFR_STAT1`	Bus control registers only	ON/OFF
`hsc_XFR_STAT2`	Bus control registers only	ON/OFF
`hsc_RESET_STATE`	reset is currently being asserted	ON/OFF
`hsc_STATE_BIT_0`	state bit for bus control & ext bridge registers	ON/OFF
`hsc_STATE_BIT_1`	state bit for bus control & ext bridge registers	ON/OFF
`hsc_STATE_BIT_2`	state bit for bus control & ext bridge registers	ON/OFF
`hsc_STATE_BIT_3`	state bit for bus control & ext bridge registers	ON/OFF
`hsc_CONNECTED`	board connection completed	ON/OFF
`hsc_HEALTHY`	board says it's healthy	ON/OFF
`hsc_WHAT_DOMAIN`	Bus Control Reg, this domain is; `hsc_OFF`=A `hsc_ON`=B	ON/OFF
`hsc_FAN_PRESENT`	PS Reg; is fan present?	ON/OFF
`hsc_FAN_FAULT`	power supplies only; power supply failure?	ON/OFF
`hsc_REG_ACTIVE`	this register's writes trigger immediate action	ON/OFF
`hsc_CLOCK_ENABLE`	ATM control register use	ON/OFF
`hsc_CLOCK_MASTER`	ATM control register use	ON/OFF
`hsc_A_FAIL`	ATM control register use	ON/OFF
`hsc_B_FAIL`	ATM control register use	ON/OFF
`hsc_PLL_LOCK`	ATM control register use	ON/OFF
`hsc_VTERM_OK`	ATM control register use	ON/OFF
`hsc_FORCE_A_FAIL`	ATM control register use	ON/OFF
`hsc_FORCE_B_FAIL`	ATM control register use	ON/OFF
`hsc_VTERM_ENABLE`	ATM control register use	ON/OFF
`hsc_GLOBAL_H110`	CPX8216T H110 reset only.	ON/OFF
`hsc_SLOT_H110`	CPX8216T H110 reset only.	ON/OFF
`hsc_BRIDGE_H110`	CPX8216T H110 reset only.	ON/OFF

Table B-7: cpx Action Enums

Attribute Name	Comment
`cpxOK`	valid operation
`cpxOFF`	request & notification status
`cpxON`	request & notification status
`cpxFASTBLINK`	for LEDs
`cpxSLOWBLINK`	for LEDs
`cpxMEDBLINK`	for LEDs
`cpxSTATUS`	all operations (read only)
`cpxHIGH`	for fans
`cpxLOW`	for fans
`cpxNOP`	status only
`cpxWRITE`	Word & Byte operations request or notification status
`cpxREAD`	
`cpxNCHG`	status only
`cpxERROR`	general failure
`cpxEARG1`	argument 1 in error
`cpxEARG2`	argument 2 in error
`cpxEARG3`	argument 3 in error
`cpxEARG4`	argument 4 in error
`cpxEARGN`	argument count(number of) in error
`cpxEOPEN`	*hsc_Open()* not yet called
`cpxELAMP`	lamp test already in progress
`cpxEALRM`	alarm test already in progress
`cpxMXNOT`	maximum subscribers exceeded
`cpxBADRES`	bad resource - invalid or non-existent
`cpxBADATT`	bad attribute - invalid or non-existent
`cpxBADACT`	bad action - invalid or non-existent
`cpxROATT`	read only attribute

Table B-7: cpx Action Enums (Continued)

Attribute Name	Comment
`cpxRORES`	read only resource
`cpxIERR`	internal error
`cpxNOFUNC`	no function - used in event notifications

`cpxACTION` Status & Error Return Value Enumerations

Table B-8: hsc_ACTION Error Return Value Enumerations

Error Enums	Meaning
note: All error enumerations are *negative*. (see “hsc_ACTION Enumerations” on page 95)	
`hsc_ERROR`	general error, no action taken by API library or HSC/B driver
`hsc_BAD_REGISTER`	slot/register specified out-of-range
`hsc_BAD_OBJECT`	bit object specified out-of-range
`hsc_BAD_ACTION`	action requested not defined or invalid
`hsc_BAD_DOMAIN`	domain specified is invalid
`hsc_ACTION_DENIED`	requested action cannot be performed
`hsc_NO_FUNCTION`	required function address for receiving signals is missing

Software Interfaces

The HSC access functions fall into five major categories:

1. Gaining HSCD Access - Before any control or status over the Hot Swap Controller/Bridge (HSC/B) can be exercised, the user must open and initialize a connection to its controller software.
2. Exercising Control - The HSC/B hardware is commanded to change its state or the state of some other device. This control extends to such items as applying board power, switching LEDs on and off, enabling/disabling interrupts and transferring cPCI bus control between processors.
3. Notification of Events - This software is designed to inform other user processes when *asynchronous* events (interrupts signifying a change of state for a resource) take place to which the HSC/B is privy and EVERY access (API call initiated) made of it. These include such items as board insertions or removals, alarm conditions, or other processes such as

altering an LEDs state, requesting a board connection, etc. The software provided allows the user to register for such events and to obtain the precise reason for the event via API calls. A tool (cpxtool) is provided to allow such monitoring on a casual basis as the developer debugs/unit tests his processes.

4. Obtaining Status - A process may at any time request the status of all items controllable by the HSC/B hardware. These include items such as board connection status, LED status, power status and overall HSC internal metrics.

NOTE: All functions described below will return an `hsc_ERROR` if any of the caller's input/output arguments are not within the caller's memory space (LynxOS only). Moreover, except for *hsc_Open()*, all hsc_-style functions will return an `hsc_ERROR` if a successful hsc_Open() has not been performed (all OS's), or in the case of cpx-Style calls, a `cpxEOPEN` is returned for the same reason.

CPX8000 System HSC Driver Control Functional Interface

What follows is a summary of the API functions which provide access to the HSC/B hardware.

- OPENING THE HOT SWAP CONTROLLER DRIVER
  hsc_Open()
- CLOSING THE HOT SWAP CONTROLLER DRIVER
  hsc_Close()
- IDENTIFYING THE DOMAIN
  hsc_GetDomainId()
- TAKING AND RELEASING THE DOMAINS/BUSES
  hsc_TakeDomain()
  hsc_ReleaseDomain()
  hsc_TakeBus()
  hsc_ReleaseBus()

  hsc_GetBusCEBridgeId() [CPX8221 only]
- CONTROLLING BUS ACCESS
  cpxLockDomain()
  cpxLockBus()
- ENUMS PROPAGATION
  cpxEnumPropagation()

- CONTROLLING POWER AND CONNECTIONS
  cpxPower()
  cpxConnect()
  cpxForceHealthy()
- CONTROLLING POWER SUPPLY FAN SPEED
  cpxFanHigh()
- CONTROLLING LED'S

  cpxLED() [see on page 117]

  cpxLampTest() [see on page 122]
- CONTROLLING ALARMS

  cpxAlarms() [see on page 123]

  cpxAlarmTest() [see on page 124]
- HARDWARE SUPPLIED PERSISTENT SCRATCH BYTES

  cpxInfoByte() [see on page 125]
- INTERRUPTS
  cpxProgramInterrupts()
  cpxEnableInterrupts()
  cpxProgramEnums()
- SIGNING UP FOR EVENTS AND EVENT CLASSES
  cpxSetEventFunction()
  cpxInstallEventFunction()
  cpxRemoveEventFunction()
  (change list of event classes of interest)
  cpxSetEventList()
  (retrieve an event, status & class)
  cpxGetEvent()
  (extract event information)
  cpxExtractEventReason()
  (pausing event notification)
  cpxBlockEventNotification()
- SETTING/GETTING BRIDGE STATES
  hsc_SetBridgeStates()
  hsc_GetBridgeStateSettings()

  hsc_SetEBridgeStates() [CPX8221 only]

  hsc_GetEBridgeStateSettings() [CPX8221 only]
- RETRIEVING STATUS ONLY

hsc_GetSlotStatus()
hsc_GetProcessorStatus()

hsc_GetPBayStatus() [CPX8216 only]

hsc_GetPSStatus()
hsc_GetBridgeStatus()

hsc_GetEBridgeStatus() [CPX8221 only]

- ACQUIRING VARIOUS REVISION LEVELS
  hsc_GetHSCRevision()
  hsc_GetPSRevision()
  hsc_GetAlarmPLDRevision()
  hsc_GetAlarmPanelRevision()
- LOW-LEVEL HSC/B ACCESS
  hsc_Action()
  hsc_RegDirect()
  hsc_RegByte()
  hsc_Interrupts()

OPENING THE HOT SWAP CONTROLLER DRIVER

The function *hsc_Open()* must be called once, and only once, prior to attempting any other HSCD operations. When an application has completed its session with the HSCD, then the *hsc_Close()* function *must* be called.

hsc_Open

NAME

`hsc_Open` - allows an application to open the hot swap controller

SYNOPSIS

```
#include <cpxapi.h>
```

int hsc_Open ()

DESCRIPTION

This function enables the CPX8000 HA API for command and status functions. This function must be called before any other functions which invoke the control and status gathering ability of the API can be used.

Open returns an `hsc_OK` if successful. Note that multiple *hsc_Open()*'s without a matching *hsc_Close()*'s are not supported. Two or more consecutive *hsc_Open()*'s return an `hsc_ERROR` result.

DIAGNOSTICS

An error return of `hsc_ERROR` signifies a failure to successfully initialize access to the API's controller driver. This will happen if the driver is not loaded (LynxOS only), the driver fails to install from its device (hscdev; LynxOS only), the HSC/B card is not found in cPCI space or the application already has an *hsc_Open()* to this driver in effect.

`hsc_OK` is returned for a successful open.

`hsc_ERROR` is returned:

•if the open fails.

•if the API's driver's not loaded.

•if the driver fails to install its device. (LynxOS only)

•if more than one *hsc_Open()* is attempted.

•Hot Swap controller hardware not found.

CLOSING THE HOT SWAP CONTROLLER DRIVER

hsc_Close

NAME

`hsc_Close` - closes an application's connection to the HSC/B board

SYNOPSIS

```
#include <cpxapi.h>
int hsc_Close (void)
```

DESCRIPTION

This function closes the program's open connection to the API's resources.

This function returns `hsc_OK` upon successful completion.

DIAGNOSTICS

This function will return an `hsc_ERROR` if the close fails. A close will fail only if an *hsc_Open()* for the calling process is not currently in effect.

IDENTIFYING THE DOMAIN

hsc_GetDomainId

NAME

`hsc_GetDomainId` - gets the domain id for the current process

SYNOPSIS

```
#include <cpxapi.h>
```

`hsc_DOMAIN_ID` *hsc_GetDomainId ()*

DESCRIPTION

This function returns the domain id for the current process.

ARGUMENTS

None.

DIAGNOSTICS

`hsc_DOMAIN_A` or `hsc_DOMAIN_B` are the only two possible domain values returned.

`hsc_ERROR` is returned if a successful *hsc_Open()* has not been performed.

TAKING AND RELEASING THE DOMAINS/BUSES

`hsc_TakeDomain`

NAME

`hsc_TakeDomain` - makes a domain's slots & functions active for this processor

SYNOPSIS

#include <cpxapi.h>

`hsc_ACTION hsc_TakeDomain`(domain)

`hsc_DOMAIN_ID` domain; /* grab slots in this domain */

DESCRIPTION

This function forcefully transfers control of the bus slots and functions in the domain specified to the current processor and process. This means all slot registers from that domain will become active for the process making the request, and any other process with an open connection to the HSC/B.

For domain A taken by either processor, its applications are now able to access system functions associated with domain A.

ARGUMENTS

domain - The `hsc_DOMAIN_ID` argument must be `hsc_DOMAIN_A`, `hsc_DOMAIN_B` or `hsc_DOMAIN_A_B`

An `hsc_OK` return code indicates a successful grabbing of the domain.

DIAGNOSTICS

`hsc_OK` is returned if this request succeeds.

`hsc_BAD_DOMAIN` is returned if the domain id provided is incorrect.

`hsc_ERROR` is returned if the request fails.

hsc_ReleaseDomain

NAME

`hsc_ReleaseDomain` - makes domain's slots & functions inactive for this process

SYNOPSIS

`#include <cpxapi.h>`

`hsc_ACTION hsc_ReleaseDomain` (domain)

`hsc_DOMAIN_ID domain`; /* grab slots in this domain */

DESCRIPTION

This function relinquishes control of the named domain for all processes runing with the current processor. This means all slot registers from that domain will become inactive (albeit still programmable). To emphasize, any other local process with a connection to the HSCD will also lose its "active" access to these registers!

ARGUMENTS

domain - The `hsc_DOMAIN_ID` argument must be `hsc_DOMAIN_A`, `hsc_DOMAIN_B` or `hsc_DOMAIN_A_B`

An `hsc_OK` return code indicates a successful jettisoning of the slot registers.

DIAGNOSTICS

`hsc_OK` is returned if this request succeeds.

`hsc_BAD_DOMAIN` is returned if the domain id is incorrect.

`hsc_ERROR` is returned if the release fails.

hsc_TakeBus

NAME

`hsc_TakeBus` - makes a bus's slots & functions active for this processor

SYNOPSIS

`#include <cpxapi.h>`

`hsc_ACTION hsc_TakeBus`(bus)

`hsc_RESOURCE_ID` bus; /* grab slots in this bus */

DESCRIPTION

This function forcefully transfers control of the bus slots and functions in the bus specified to the current processor and process. This means all slot registers on that bus will become active for the process making the request, and any other process with an open connection to the HSC/B.

For bus A taken by either processor, applications are able to access other system functions associated with bus A.

Taking bus A is equivalent to taking domain A. Taking bus B is not the same as taking domain B on the CPX8221. Taking domain B gets both B & C buses on that platform.

ARGUMENTS

bus - The `hsc_RESOURCE_ID` argument must be `hsc_BUS_A`, `hsc_BUS_B`, or `hsc_BUS_C`(CPX8221 only).

An `hsc_OK` return code indicates a successful grabbing of the bus registers.

DIAGNOSTICS

`hsc_OK` is returned if this request succeeds.

`hsc_BAD_REGISTER` is returned if the bus id provided is incorrect.

`hsc_ACTION_DENIED` is returned if the bus C is specified, but bus B is not currently controlled.

`hsc_ERROR` is returned if the request fails.

hsc_ReleaseBus

NAME

`hsc_ReleaseBus` - makes a bus's slots & functions inactive for this processor

SYNOPSIS

`#include <cpxapi.h>`

`hsc_ACTION hsc_ReleaseBus`(bus)

`hsc_RESOURCE_ID bus`; /* grab slots in this domain(or bus)*/

DESCRIPTION

This function forcefully relinquishes control of the bus slots and functions in the bus specified for the current processor and process. This means all slot registers on that bus will become inactive for the process making the request, and any other process with an open connection to the HSC/B.

If bus A is released by the bus B processor, it also disallows the application from accessing other system functions associated with bus A.

Releasing bus A is equivalent to releasing domain A. Releasing bus B is not the same as releasing domain B on the CPX8221. Releasing bus B does not release bus C bus on that platform as does releasing domain B.

ARGUMENTS

bus - The `hsc_RESOURCE_ID` argument must be `hsc_BUS_A`, `hsc_BUS_B`, or `hsc_BUS_C`(CPX8221 only).

An `hsc_OK` return code indicates a successful grabbing of the bus registers.

DIAGNOSTICS

`hsc_OK` is returned if this request succeeds.

`hsc_BAD_REGISTER` is returned if the bus id provided is incorrect.

`hsc_ERROR` is returned if the request fails(HW issue).

hsc_GetBusCEBridgeId()

NAME

`hsc_GetBusCEBridgeId` - gets id of extension bridge attached to bus C

SYNOPSIS

```
#include <cpxapi.h>
```

hsc_RESOURCE_ID hsc_GetBusCEBridgeId()

DESCRIPTION

This function returns the id of the extension bridge that was used to acquire bus C after an *hsc_TakeDomainBus()* call.

ARGUMENTS

None.

DIAGNOSTICS

`hsc_EXT_BRIDGE_1` is returned if this bridge indicates a bus active state.

`hsc_EXT_BRIDGE_2` is returned if this bridge indicates a bus active state.

`hsc_NO_FUNCTION` is returned if neither extension bridge is actively connected to bus C.

CONTROLLING BUS ACCESS

cpxLockDomain()

NAME

`cpxLockDomain` - status or control payload board's access to domain's bus(ses)

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxLockDomain`(domain_id, action)

```
int domain_id;
```

```
int action;
```

DESCRIPTION

This function allows or disallows payload boards within a domain to access its bus.

ARGUMENTS

Valid domain_id's are `cpxDOMAIN_A`, `cpxDOMAIN_B` or `cpxDOMAIN_A_B`.

Table B-9: Argument vs. Platform Detail

platform	**argument**	**buses locked**
all	`cpxDOMAIN_A`	`cpxBUS_A`
CPX8216	`cpxDOMAIN_B`	`cpxBUS_B`
CPX8221	`cpxDOMAIN_B`	`cpxBUS_B & cpxBUS_C`
CPX8216	`cpxDOMAIN_A_B`	`cpxBUS_A & cpxBUS_B`
CPX8221	`cpxDOMAIN_A_B`	`cpxBUS_A, cpxBUS_B & cpxBUS_C`

Valid actions consist of `cpxON` (to lock bus access) or `cpxOFF` (to allow bus access).

DIAGNOSTICS

`cpxOK` is returned for a domain successfully locked or unlocked.

`cpxBADRES` is returned if the domain specified is not valid.

`cpxBADACT` is returned if the action specified is incorrect.

cpxLockBus()

NAME

`cpxLockBus` - controls a payload's access to the bus or gets bus access status

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxLockBus`(bus_id, action)

```
int bus_id;
```

```
int action;
```

DESCRIPTION

This function allows or disallows payload boards within a particular bus to access it.

ARGUMENTS

Valid arguments for `bus_id` are `cpxBUS_A`, `cpxBUS_B` or `cpxBUS_C`**(CPX8221 only)**.

Action is either `cpxON` (to prevent bus access), `cpxOFF` (to allow bus access) or `cpxSTATUS` (to determine current lock state).

DIAGNOSTICS

`cpxOK` is returned for a bus successfully locked or unlocked.

`cpxON` is returned if the bus specified is currently locked from payload slot access.

`cpxOFF` is returned if the bus specified is currently unlocked and available for payload slot access.

`cpxBADRES` is returned if the bus specified is invalid.

`cpxBADACT` is returned if the action specified is incorrect.

`cpxERROR` is returned if the action specified could not be performed.

ENUM PROPAGATION

cpxEnumPropagation

CONTROLLING POWER AND CONNECTIONS

cpxPower()

NAME

`cpxPower` - power on or off multiple resources, or status a single resource

SYNOPSIS

```
#include <cpxapi.h>

int cpxPower(resources, action)

unsigned long resources;

int action;
```

DESCRIPTION

This function applies or removes power from one or more resources or gets the current power status from a *single* resource. More than one resource with a power attribute may be specified when setting the power to on or off. These resource *aggregates* are discussed in “cpx-Style Resource Enumerations” on page 92. When acquiring status, only one resource may be specified. If not, the first resource extracted from the aggregate will be used for the status. For obvious reasons, this approach is not recommended.

It is possible that if an error is returned, such as BADRES for a bad resource, that other valid resources have already been correctly handled.

ARGUMENTS

Valid register aggregates include all slots inclusive of cpxSLOT01 .. cpxSLOT06, cpxSLOT11 .. cpxSLOT16 and slots cpxSLOT17 .. cpxSLOT21 (CPX8221). All valid payload slots may be specified with the special enumeration of `cpxALL_SLOTS`. Also valid are cpxPROCA, cpxPROCB, cpxBRIDGEA, cpxBRIDGE_B, cpxEB1 .. cpxEB2 (CPX8221) cpxPS1 .. cpxPS3 and cpxPBAY1 .. cpxPBAY4(CPX8216).

Action is declared as `cpxON`, `cpxOFF` or `cpxSTATUS` *only.*

DIAGNOSTICS

`cpxOK` is returned for a successful `cpxON` or `cpxOFF` action.

`cpxON` or `cpxOFF` is returned if the action specified is `cpxSTATUS`.

`cpxEARG1` is returned if no resources are specified (i.e. a 0 is passed).

`cpxBADRES` is returned for an invalid resource.

`cpxBADACT` is returned for an invalid action.

`cpxEOPEN` is returned if an *hsc_Open()* has not been successfully performed.

cpxConnect()

NAME

cpxConnect - turn resources' connect on or off, or status a single resource

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxConnect`(resources, action)

```
unsigned long resources;
```

```
int action;
```

DESCRIPTION

This function connects or disconnects one or more resources or gets the current connect status from a *single* resource. It is analogous in all other respects to "`cpxPower()`" on page 113.

It is possible that if an error is returned, such as BADRES for a bad resource, that other valid resources have already been correctly handled.

cpxForceHealthy()

NAME

`pxForceHealthy` - control or status the board healthy overide for resource(s)

SYNOPSIS

`#include <cpxapi.h>`

`int cpxForceHealthy`(resources, action)

`unsigned long resources;`

`int action;`

DESCRIPTION

This function forces a resource's healthy status to track a payload board's power on or off condition. It is used for boards which are non-compliant with the hot plug PCI specifications for asserting healthy. It can also return the current force healthy status for a *single* resource. It is analogous in all other respects to "`cpxPower()`" on page 113.

It is possible that if an error is returned, such as `cpxBADRES` for a bad resource, that other valid resources have already been correctly handled.

CONTROLLING POWER SUPPLY FAN SPEED

cpxFanHigh()

NAME

`cpxFanHigh` - status or run a power supply's fan at high or temp. controlled speed

SYNOPSIS

`#include <cpxapi.h>`

`int cpxFanHigh`(fan_ids, action)

`unsigned long fan_ids;`

`int action;`

DESCRIPTION

This function causes one or more power supply fans to be run at it highest speed for additional cooling, or to be run at a speed as dictated by thermistor measurements. This action takes place immediately only if the processor owns domain A.

ARGUMENTS

`Fan_ids` is an aggregate of up to three fan identifiers arithmetically or'd together. They are `cpxFAN1`, `cpxFAN2` and `cpxFAN3`. As a convenience, the identifier `cpxALL_FANS` is enumerated to include all three fans.

Action is `cpxON` (run at high speed), `cpxOFF` (allow automatic variable speed setting of fan based on temperature conditions) or `cpxSTATUS` (is the fan currently running at high speed: `cpxON` == yes, `cpxOFF` == no).

If an aggregate of more than one fan is specified when `cpxSTATUS` is requested, then the first fan encountered--beginning with `cpxFAN1`--will have its status returned. It's better to specify only one fan at a time when requesting status.

DIAGNOSTICS

`cpxOK` is returned for an action of `cpxON` or `cpxOFF` being successfully taken.

`cpxBADRES` is returned if fan_ids has no valid fan identifiers.

`cpxEOPEN` is returned if *hsc_Open()* has not been performed to make the library available.

`cpxBADACT` is returned if the action requested is invalid.

`cpxON` is returned for a status request if the fan is running at high speed.

`cpxOFF` is returned for a status request if the fan is running at a temperature driven speed.

CONTROLLING LED'S

cpxLED()

NAME

`cpxLED` - turn on, off or blink various chassis LEDs

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxLED`(resources, leds, action)

```
unsigned long resources;

unsigned long leds;

int action;
```

DESCRIPTION

This function is used to light, extinguish or blink various chassis LEDs. More than one resource with available LEDs may be specified and more than one of it/their LEDs may be placed in the desired state. These resource and LED aggregates will be discussed further.

States for an LED is either on, off or blinking. If blinking, three blink rates are available. They are 1/4 second (cpxFASTBLINK), 1/2 second (cpxMEDBLINK) and 1 second (cpxSLOWBLINK).

ARGUMENTS

The first argument specifies a grouping of one or more resources for which LED attributes exist. Their enumerations are shown in the table below.

The second argument specifies one or more LED enumerations which specify the LEDs to be affected. These enumerations are `cpxLED1` through `cpxLED4`. Specifying `cpxALL_LEDS` will refer to all LEDs that any particular resource supports regardless of number. For instance, `cpxALL_LEDS` can be used for a power supply (like cpxPS1) and a payload slot (like cpxSLOT01) even though the power supply has four LEDs available while a payload slot has but three. In specifying aggregates of LEDs, you must mathematically *OR* them together. For example, to light LEDs one and three for slot6, you'd use cpxLED(cpxSLOT06, `cpxLED1` | `cpxLED3`, `cpxON`);

Resources with LEDs include the payload slots, processor cards, HSC/Bridge cards, system LEDs. alarm LEDs, power supplies, peripheral bays (CPX8216) and extension bridges (CPX8221).

The aggregate which specifies all payload slots (non-system board occupied slots) is cpxALL_SLOTS. Thus, to blink (at 1/4 second) all LED 1's for all payload cards, you'd use: cpxLED(`cpxALL_SLOTS`, `cpxLED1`, `cpxFASTBLINK`);

In another example, to extinguish all LEDs for the `cpxSYSLED` resource, processor A and slots 3 and 5, you'd use:

```
cpxLED(cpxSLOT03|cpxSLOT05|cpxPROCA|cpxSYSLED,
cpxALL_LEDS, cpxOFF);
```

The third argument specifies the LEDs new state or a status request. The arguments are `cpxON`, `cpxOFF`, `cpxFASTBLINK`, `cpxMEDBLINK` and `cpxSLOWBLINK` or `cpxSTATUS`. These arguments must not be OR'd together as the result would be indeterminate.

SPECIAL NOTE: Alarms can blink also. The first three alarms have an associated LED which lights as a by-product of enabling the alarm. The fourth does not, but it is blinked in order to remain consistent. All alarm blinking can be monitored externally. The alarm enumerations can be used, or, `cpxLED1`, `cpxLED2`, `cpxLED3` and `cpxLED4` may be used for `cpxCRIT`, `cpxMAJOR`, `cpxMINOR` and `cpxRACK` respectively. All four alarms may also be designated using the `cpxALL_ALARMS` enumeration.

DIAGNOSTICS

`cpxOK` is returned for a successful call to alter LED states.

`cpxON`, `cpxOFF`, cpxFASTBLINK, cpxMEDBLINK or cpxSLOWBLINK is returned in response to a `cpxSTATUS` request.

`cpxEARG1` is returned if NO resources are specified or the specified resource has no LEDs.

`cpxEARG2` is returned if NO valid LEDs were specified.

`cpxEARG3` is returned if NO valid action was specified.

`cpxEOPEN` is returned if the *hsc_Open()* call has not been made.

CPXTOOL USAGE

Here is an example of invoking *cpxLED()* using `cpxtool`:

```
usage: l)ed arg1 arg2 arg3
       arg1  = one or more slot/resource names-
               slot01..slot16 (cpx8216) or all_slots for all
               slot01..slot21 (cpx8221) or all_slots for all
               Separate by spaces or vertical bars '|'
               (also for ps, procs, bridges(all), alarms, sysleds)
       arg2  = any combination of led1, led2, led3 or led4
               or use all_leds for led1|led2|led3 grouping
       arg3  = on, off, fblink, mblink or sblink
                        (fast) (medium)   (slow)
example: l slot02 slot06 slot16 led1 led3 mblink
         l slot02|slot06|slot16 led1|led3 mblink
         l all_slots proca all_leds on
         l slot02 led2 status
```

LEDs available by resource..

Table B-10: LEDs Available per Resource

chassis exception	resource	cpxLED				comment
		1	2	3	4	
	`cpxSLOT01`	t	t	t		Payload slots 1 through 6 in Domain A, on Bus A.
	`cpxSLOT02`	t	t	t		
	`cpxSLOT03`	t	t	t		
	`cpxSLOT04`	t	t	t		
	`cpxSLOT05`	t	t	t		
	`cpxSLOT06`	t	t	t		

Table B-10: LEDs Available per Resource

chassis exception	resource	cpxLED				comment
		1	2	3	4	
	`cpxSLOT11`	t	t	t		Payload slots 11 through 16 in Domain B, on Bus B.
	`cpxSLOT12`	t	t	t		
	`cpxSLOT13`	t	t	t		
	`cpxSLOT14`	t	t	t		
	`cpxSLOT15`	t	t	t		
	`cpxSLOT16`	t	t	t		
	`cpxSLOT17`	t	t	t		Payload slots 17 through 21 in Domain B, on Bus C.
CPX8221 ONLY	`cpxSLOT18`	t	t	t		
	`cpxSLOT19`	t	t	t		
	`cpxSLOT20`	t	t	t		
	`cpxSLOT21`	t	t	t		
	`cpxALL_SLOTS`	t	t	t		All applicable payload slots.
	`cpxPROCA`	t	t	t		Chassis processors.
	`cpxPROCB`	t	t	t		
	`cpxBRIDGEA`	t	t	t		Chassis HSC & inter-domain bridges.
	`cpxBRIDGEB`	t	t	t		
	`cpxALARM`	t	t	t	t	System alarms (see Special Note on previous page).
	`cpxPS1`	t	t	t	t	`cpxLED1` & `cpxLED2` are for the power supplies. `cpxLED3` & `cpxLED4` are for the fans.
	`cpxPS2`	t	t	t	t	
	`cpxPS3`	t	t	t	t	

Table B-10: LEDs Available per Resource

chassis exception	resource	cpxLED				comment
		1	2	3	4	
IN CPX8216 CPX8216T CPX8216A ONLY	`cpxPBAY1`	t	t			
	`cpxPBAY2`	t	t			Peripheral Bays
	`cpxPBAY3`	t	t			
	`cpxPBAY4`	t	t			
CPX8221 ONLY	`cpxEBRIDGE1`	t	t	t		Extension Bridges
	`cpxEBRIDGE2`	t	t	t		

cpxLampTest()

NAME

`cpxLampTest` - light chassis LEDs for inspection purposes (except ALARMS)

SYNOPSIS

`#include <cpxapi.h>`

`int cpxLampTest`(duration)

`int duration`;

DESCRIPTION

This function lights all system chassis LEDs (`cpxALARM` LEDs notwithstanding), and in the case of a CPX8221 chassis, both extension bridge board's LEDs, for the amount of time specified. This allows for a visual inspection of the LEDs to ensure that they are working properly. Whatever state the LEDs are in, including blink mode, is preserved for the duration of the test, then restored after it completes.

If a lamp test or an alarm test is already in progress, then this lamptest is rejected with an appropriate error message.

ARGUMENTS

Valid arguments are between 1 and 60 seconds.

DIAGNOSTICS

`cpxOK` is returned for a successful call.

`cpxELAMP` is returned if a lamp test is already in progress.

`cpxEALRM` is returned if an alarm test is currently in progress.

`cpxEARG1` is returned if the duration argument is not within (1..60) seconds.

`cpxEOPEN` is returned if the *hsc_Open()* call has not been made.

CPXTOOL USAGE

Here is an example of invoking *cpxLampTest()* using cpxtool:

```
usage:  L)ampTest seconds; range = (1..60)
example:     L 10
```

CONTROLLING ALARMS

cpxAlarms()

NAME

`cpxAlarms` - turn on or off, one or more system alarms, or status one

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxAlarms` (alarm_ids, action)

```
unsigned long alarm_ids;
int action;
```

DESCRIPTION

This function enables or disables one or more of the system alarms, or gets the current status of a single alarm. The four alarms which may be aggregated (arithmetically or'd together) are `cpxMINOR`, `cpxMAJOR`, `cpxCRIT` and `cpxRACK`.

If a status is requested, but an aggregate of more than one alarm is specified, then the status of one of those alarms with be returned, but it will not be deterministic. If statusing an alarm, specify one alarm only.

If an alarm is to be "blinked", then use the *cpxLED()* function with the `cpxALARM` resource for all alarm on, off, blink and status related operations!

ARGUMENTS

Valid `alarm_ids` are `cpxMINOR`, `cpxMAJOR`, `cpxCRIT` and `cpxRACK`. The enumeration `cpxALL_ALARMS` is provided as a convenience.

Valid actions are `cpxON` (turn the alarm on), `cpxOFF` (turn the alarm off) and `cpxSTATUS` (for current state).

DIAGNOSTICS

`cpxOK` is returned for a successful call.

`cpxBADATT` is returned if no valid `alarm_ids` are specified.

`cpxBADACT` is returned if the action given is invalid.

`cpxON` is returned if the alarm specified is currently on.

`cpxOFF` is returned if the alarm specified is currently off.

`cpxEOPEN` is returned if the *hsc_Open()* call has not been made.

cpxAlarmTest()

NAME

`cpxAlarmTest` - light chassis Alarm LEDs for inspection purposes

SYNOPSIS

```
#include <cpxapi.h>
int cpxLampTest (duration)
int duration;
```

DESCRIPTION

This function enables all the `cpxALARM`'s four alarm attributes--`cpxMINOR`, `cpxMAJOR`, `cpxCRIT` and `cpxRACK`--which in turn also lights the LEDs associated with the first three alarms. Whatever state all the alarms are in, including blink mode, is preserved for the duration of the test, then restored after it completes.

If a lamp test or an alarm test is already in progress, then this alarm test is rejected with an appropriate error message--`cpxELAMP` or `cpxEALRM`.

ARGUMENTS

Valid arguments are between 1 and 60 seconds.

DIAGNOSTICS

`cpxOK` is returned for a successful call.

`cpxELAMP` is returned if a lamp test is currently in progress.

`cpxEALRM` is returned if an alarm test is already in progress.

`cpxEARG1` is returned if the duration argument is not within (1..60) seconds.

`cpxEOPEN` is returned if the *hsc_Open()* call has not been made.

CPXTOOL USAGE

Here is an example of invoking *cpxAlarmTest()* using cpxtool:

```
usage:  A)larmTest seconds; range = (1..60)
example:     A 10
```

HARDWARE SUPPLIED PERSISTENT SCRATCH BYTES

cpxInfoByte()

NAME

`cpxInfoByte` - write or read a resource's scratch byte

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxInfoByte` (resources,value,action)

```
unsigned long resources;
char * value;
int action;
```

DESCRIPTION

This function reads or writes a single byte to a resource's scratch byte area, located on the *HSC/B hardware*, for whatever application purpose deemed useful.

Most system resources have a scratch byte which is non-volatile as long as the computer system is under power. These bytes will survive computer resets and crashes without corruption. A software application may take advantage of these storage locations associated with the hardware if found useful.

It is possible that if an error is returned, such as BADRES for a bad resource, that other valid resources have already been correctly handled.

ARGUMENTS

Valid resources are an aggregate of one or more of the following system resources arithmetically OR'd together:

`cpxSLOT01` .. `SLOT06, cpxSLOT11` .. `cpxSLOT16` and for the `CPX8221` only, slots `cpxSLOT17` .. `cpxSLOT21.` Also valid are `cpxPROCA, cpxPROCB, cpxBRIDGEA, cpxBRIDGEB, cpxPS1` .. `cpxPS3, cpxPBAY1` .. `cpxPBAY4 (CPX8216), cpxEB1 and cpxEB2 (CPX8221), cpxALARM, cpxBUS_A, cpxBUS_B, cpxEEPROM, cpxIMASK` and `cpxSYSLED`.

The value to write must be stored in a variable of type char whose address is provided. For a read operation, the current value of the scratch byte for the resource specified is written to the byte address provided.

The action is either `cpxWRITE` or `cpxREAD`. If `cpxREAD`, then only *one* resource should be specified. If the aggregate contains more that one resource when reading, the resource identity from which the value is read and returned will be non-deterministic.

DIAGNOSTICS

`cpxOK` is returned for a successful read or write action.

`cpxEARG2` is returned if the byte's value address provided is zero (0).

`cpxBADRES` is returned if a resource is invalid or no valid resources were specified.

`cpxEOPEN` is returned if a successful cal to *hsc_Open()* has not been made.

INTERRUPTS

cpxProgramInterrupts

NAME

`cpxProgramInterrupts` - program resources to be able to generate interrupts

SYNOPSIS

```
#include <cpxapi.h>

int cpxProgramInterrupts (resources, action)

unsigned long resources;

int action;
```

DESCRIPTION

This function programs one or more resources to be capable of generating PCIA interrupts to the processor when interrupts are enabled. Programming is permitted only for those registers actually capable of generating interrupts.

Resources are one or more resources aggregated together using arithmetic OR'ing.

For example, to program slot1, slot2 and power supply2, the following should be used--

```
(cpxSLOT01|cpxSLOT02|cpxPS1)
```

--for the resources argument.

ARGUMENTS

As mentioned above, resource consist of any resource capable of generating an interrupt or'd together with other such resources as desired.

Action is `cpxON` (to program), `cpxOFF` (to deprogram) or `cpxSTATUS` (to get the current programmed state of the resource).

DIAGNOSTICS

An `cpxOK` is returned for successfully programming *all* resources specified.

An `cpxBADRES` is returned if *any* resource specified is not valid (cannot generate interrupts).

A `cpxON` is returned if the resource specified is currently programmed.

A `cpxOFF` is returned if the resource specified is currently not programmed.

A `cpxEOPEN` is returned if a successful call to *hsc_Open()* has not yet been made.

cpxEnableInterrupts

NAME

`cpxEnableInterrupts` - enable or disable all resource interrupts to the processor

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxEnableInterrupts`(action)

```
int action;
```

DESCRIPTION

This function enables or disables processor interrupts. If interrupts are not enabled, then no event notifications to applications can occur because resources programmed to interrupt cannot get their interrupts through to the processor. Hence, no event software comes alive to handle them and send notifications.

ARGUMENTS

Valid actions are `cpxON` (enable interrupts), `cpxOFF` (disable interrupts) and `cpxSTATUS` (return the current interrupt enable setting).

DIAGNOSTICS

`cpxOK` is returned if interrupts have been successfully enabled or disabled.

`cpxON` is returned for a status if interrupts are currently enabled.

`cpxOFF` is returned for a status if interrupts are currently disabled.

`cpxEOPEN` is returned if an *hsc_Open()* has not yet been performed.

cpxProgramEnums

NAME

`cpxProgramEnums` - program/deprogram a domain's ability to generate an interrupt

SYNOPSIS

```
#include <cpxapi.h>
```

`int cpxProgramEnums`(domain_id, action)

```
int domain_id;
```

```
int action;
```

DESCRIPTION

This function enables the enum event taking place in a domain to generate an interrupt, or deprograms it so that it cannot. This function must be called *in addition to cpxProgramInterrupts()* for enum events to actually generate processor interrupts.

ARGUMENTS

A `domain_id` of `cpxDOMAIN_A`, `cpxDOMAIN_B` or `cpxDOMAIN_A_B` must be provided.

An action of `cpxON` (program the Enum interrupt), `cpxOFF` (deprogram the Enum interrupt) or `cpxSTATUS` (get Enum's current programmed setting) is valid.

DIAGNOSTICS

`cpxOK` is returned for a successful `cpxON` or `cpxOFF` action.

`cpxON` is returned if the Enum for the specified domain is currently programmed and the action requested is `cpxSTATUS`.

`cpxOFF` is returned if the Enum for the specified domain is not currently programmed and the action requested is `cpxSTATUS`.

`cpxBADACT` is returned if an invalid action is specified.

`cpxBADRES` is returned if an invalid domain is specified.

`cpxEOPEN` is returned if a successful *hsc_Open()* has not been performed.

EVENT NOTIFICATION AND RETRIEVAL

cpxSetEventFunction

NAME

`cpxSetEventFunction` - remove or specify function for event signals

SYNOPSIS

```
#include <cpxapi.h>
int cpxSetEventFunction(func, action, classes, sigval)
void(*func)();
int action;
unsigned long classes;
int sigval;
```

DESCRIPTION

This function installs or removes a process's event function. Event functions are signaled when events corresponding to one of the classes specified in the classes aggregate occur.

After receiving a signal, the user's function must retrieve the events and examine them. Two additional functions are provided to assist in accomplishing this task. They are *cpxGetEvent()* in section on page 136 and *cpxExtractEventReason()* in section on page 138.

ARGUMENTS

action is either `cpxON`, `cpxOFF` or `cpxSTATUS`. If `cpxON`, then the function will be installed and the classes aggregate list used to determine which events will cause a signal to be sent to the func specified. If `cpxOFF` is specified, then the function specified will no longer be signaled. In the `cpxOFF` case, the values for *classes* and *sigval* may be 0 because they will not be used. The case is also true for `cpxSTATUS`, but a `cpxON` or `cpxOFF` is returned based upon the current event notification status of the proces.

func is the address of the function which receives a signal when a wanted event occurs. It is never passed as a NULL pointer or NULL value.

sigval is the value of the signal which will be used to cause invocation of **func** via a system *kill()*. If a value of zero is specified, then the default signal value of 28 will be used.

classes - If the action is `cpxOFF`, its value is immaterial because it is ignored. Otherwise, it is the aggregate of event classes being signed up for. When an event for one of the specified classes occurs, a signal is sent to the function specified. It is then up to that function to retrieve and examine the event. See the classes description in “cpxInstallEventFunction” on page 132.

DIAGNOSTICS

A `cpxEOPEN` is returned is the *hsc_Open()* call has not been made.

A `cpxEARG1` is returned if the func argument address is ever NULL. In this case, the call is ignored.

A `cpxEARG2` is returned if the action is not `cpxON`, `cpxOFF` or `cpxSTATUS`.

A `cpxBADACT` is returned if the action is `cpxOFF` with no previous `cpxON` having been performed.

A `cpxBADACT` is returned if the action is neither `cpxOFF` nor `cpxON`.

for a `cpxSTATUS` action

A `cpxON` indicates that the process is currently set up to receive events.

A `cpxOFF` indicates that the process is *not* currently set up to receive events.

for a `cpxON` or `cpxOFF` action

A `cpxOK` indicates that the request was accepted and acted upon.

cpxInstallEventFunction

NAME

`cpxInstallEventFunction` - declare function for event signals and classes wanted

SYNOPSIS

```
#include <cpxapi.h>

int cpxInstallEventFunction (func, classes, sigval)

void(*func)();

unsigned long classes;

int sigval;
```

DESCRIPTION

This function, in conjunction with *cpxGetEvent()* and *cpxExtractEventReason()*, form the backbone of managing the High Availability environment. The *cpxInstallEventFunction()* call allows the application to specify the user's own function which is to be invoked whenever an event takes place and also to provide an aggregate list of events for which notifications are wanted. Once this function is successfully invoked, the calling process is elevated to the status of a *subscriber*, as opposed to a simple *user*, of the HSCD.

The user's function, once signaled, must acquire the events and the reason(s) for the events from the resource controller using *cpxGetEvent()* and *cpxExtractEventReason()*. What this means, for example, is that when a payload board asserts `hsc_CONNECTED`, the user's function is signaled. Or if a board loses `hsc_HEALTHY`, the application is notified. These notifications are critical to managing the HA environment. In addition, anytime a user process makes a request of the controller to perform some operation, that notification is also made available to the any other subscriber (event-registered) process.

When calling this function, the user's event notification function address (a C-function) is specified, plus an aggregate list of classes the user want to be notified about and the value of the signal to be used in notifying the process. The class list is a grouping of classes mathematically OR'd together. When the user gets the events, they can switch on the class in order to identify and process the event properly. The signal value passed as the third argument will default to a 28 if a zero is passed. Otherwise, the value passed by the user will be used. No validation is made upon the signal value--whatever is passed is accepted.

ARGUMENTS

Func is the address of the function to receive a signal when an event occurs. It is never passed as a NULL pointer or a NULL value. Doing so will fail the call. Since only one function may ever be specified by any given process to receive event signals, making subsequent calls to this function will *replace* the previous function's address with the new one.

Sigval is the value of the signal which will be used to cause invocation of **func** via a system *kill()*. If a value of zero is specified, then the default signal value of 28 will be used.

Classes is a set of one or more class types mathematically OR'd together and specifies which events are to cause **func** to be invoked. The value `cpxALLCLS` will cause all controller events which are traceable (see list below) to be reported to the subscriber's function.

To request only `cpxINTRPT` & `cpxACTION` event reporting, using the default signal value of 28, the call would be:

```
cpxInstallEventFunction(my_func, cpxINTRPT | cpxACTION,
0);
```

Valid classes are:

Table B-11: Event Classes

Class	Events Captured
`cpxINTRPT`	all controller interrupt events
`cpxACTION`	all *hsc_Action()* requests made of the controller
`cpxREGDIR`	all *hsc_RegDirect()* requests
`cpxREGBYT`	all *hsc_RegByte()* requests
`cpxSETLED`	all *cpxLED()* requests
`cpxLEDTST`	all *cpxLampTest()* calls
`cpxALMTST`	all *cpxAlarmTest()* calls
`cpxNOTIFY`	all *cpxSetNotifyFunction()* requests [Install/RemoveNotifyFunction]
`cpxSETMSK`	all *cpxSetEventList()* requests

Table B-11: Event Classes (Continued)

Class	Events Captured
`cpxPRGINT`	all *hsc_Interrupt()* calls [Program/DeProgram interrupts]
`cpxALLCLS`	includes all the above.

DIAGNOSTICS

A `cpxEOPEN` is returned if the cpxapi library has not been opened using *hsc_Open()*.

A `cpxMXNOT` is returned if the maximum number of event subscribers has been reached. This configurable value is currently ten (10).

A `cpxARG1` is returned if the func argument address is ever NULL. In this case, the call is ignored.

A `cpxOK` indicates that the request was accepted and acted upon.

Note that a **class** aggregate value of 0 will not create an error condition, but will allow one or more subsequent calls to *cpxSetEventList()* to be made to create and alter the list of desired events.

cpxRemoveEventFunction

NAME

`hsc_RemoveEventFunction` - remove function for event signals

SYNOPSIS

```
#include <cpxapi.h>

int cpxRemoveEventFunction(func)

void(*func)();
```

DESCRIPTION

This function removes the user's previously specified function from the *driver's* list of functions to be signaled when Hot Swap controller events take place. For any given process, only one function can be specified to receive event signals.

ARGUMENTS

Func is the address of the function which received a signal when an event occurred. It is never passed as a NULL pointer or NULL value.

DIAGNOSTICS

A `cpxEOPEN` is returned if the *hsc_Open()* call has not been made.

A `cpxMXNOT` is returned if the maximum number of event subscribers has been reached. This configurable value is currently ten(10).

A `cpxBADACT` is returned if no *cpxInstallEventFunction()* has been called prior to this function.

A `cpxEARG1` is returned if the func argument address is ever NULL. In this case, the call is ignored.

A `cpxOK` indicates that the request was accepted and acted upon.

(CHANGE LIST OF EVENT CLASSES OF INTEREST)

cpxSetEventList

NAME

`cpxSetEventList` - re-declare event class list for which notifications are wanted

SYNOPSIS

```
#include <cpxapi.h>
int cpxSetEventList (classes)
unsigned long classes;
```

DESCRIPTION

This function is used to respecify the aggregate list of events for which the user's function declared in *cpxInstallEventFunction()* or *cpxSetEventFunction()* is to be signalled. See "cpxInstallEventFunction" on page 132 for a detailed list of classes which may be specified.

ARGUMENTS

Classes is a mathematically OR'd list of events for which signals are wanted.

DIAGNOSTICS

A `cpxEOPEN` is returned is the *hsc_Open()* call has not been made.

A `cpxOK` indicates that the request was accepted and acted upon.

(RETRIEVE AN EVENT, STATUS AND CLASS)

cpxGetEvent

NAME

`cpxGetEvent` - retrieve an event's class and status

SYNOPSIS

```
#include <cpxapi.h>
int cpxGetEvent (class, status)
unsigned long *class;
unsigned long *status;
```

DESCRIPTION

This function is the companion to *cpxInstallEventFunction()* and must be called immediately after the user's event notification function is invoked. This function retrieves the first, or next, class of the event and the status it resulted in.

Typically, an application that, for instance, asked a functional board to power on would anticipate a 'board healthy' event. Likewise, a board that was pulled or suddenly lost power would cause 'board healthy' to be de-asserted. In both cases, if the event is one which the application's function is being notified about, then it will be signaled to pick up that event for processing.

Since multiple events can occur nearly simultaneously, this function must be called repeatedly until '`cpxERROR`' is returned. By doing this, the event reasons list is exhausted. However, it is important to call "cpxExtractEventReason" on page 138 in between *cpxGetEvent()*'s in order to acquire the details of the event.

ARGUMENTS

The first argument is the address of an unsigned long which will receive the event class--the type of event it is. An event can be tracing requests placed by another peer or responding with interrupts occurring in response to these calls, or both. The class of the event identifies them.

The second argument is the address of another unsigned long to receive the status or results for the event. A cpxINTRPT event ALWAYS has a status of `cpxOK`. But the other events, such as `cpxACTION` could return `cpxEOPEN`, `cpxON` or even `cpxEARG1`. This information, used with the *cpxExtractEventReason()* allows very precise understanding of what is transpiring in the system.

See *cpxInstallEventFunction()* for a list of classes returned by *cpxGetEvent()*.

DIAGNOSTICS

`cpxERROR` is returned:

if this call is made and no notifications were sent.

if all events have been retrieved.

`cpxEOPEN` is returned if *hsc_Open()* has never been called.

`cpxOK` is returned when an event has been retrieved.

(EXTRACT EVENT INFORMATION)

cpxExtractEventReason

NAME

`cpxExtractEventReason` - retrieve event notification specifics

SYNOPSIS

```
#include <cpxapi.h>
int cpxExtractEventReason (arg1, arg2, arg3, arg4)
hsc_RESOURCE_ID long *arg1;
hsc_ATTRIBUTE *arg2;
hsc_ACTION *arg3;
unsigned long *arg4;
```

DESCRIPTION

This function is another companion to *cpxInstallEventFunction()* and is called immediately after the user's event notification function has retrieved an event using *cpxGetEvent()*.

Typically, for instance, an application that asked a functional board to power on would anticipate a 'board healthy' event. Likewise, a board that was pulled or suddenly lost power would cause 'board healthy' to be de-asserted. In another case, an application might alter the state of an LED. If *cpxInstallEventFunction()* also specified an `cpxACTION` call of events, this notification would also be received.

If this function is not called after each *cpxGetEvent()* call, then the reasons for the events will be lost.

ARGUMENTS

There are four arguments which provide the remaining information about the retrieved event. The values returned in these arguments vary based upon the type of class being examined. They are described below in a table.

For example; if hsc_SLOT_01's hsc_HEALTHY attribute went off, then the values returned would be:

arg1(resource) == `hsc_SLOT_01`

arg2(attribute) == `hsc_HEALTHY`

arg3(action) == `cpxOFF`

arg4 == *not used*

class == `cpxINTRPT` (from previous *cpxGetEvent()* call)

status == `cpxOK` (cpxINTRPT class events are *always* `cpxOK`!)

Note that in the following table, the “hsc” column. If this column is marked ‘y’, then arg1 and arg2 values returned will be of the “hsc_” variety. Otherwise, they will use the new “cpx” variety. All **status**’s are of the cpx-variety. All **actions** are of the cpx-variety. Actions can take on the same values as status’s also and cover a wider range of possibilities. Status and action values are in a separate table below.

Here are the various classes and the meanings of the values returned for each:

Table B-12: Extraction Event

Class of Event	hsc	Arguments returned from cpxExtractEventDetail()				Status
		arg1	arg2	arg3	arg4	
`cpxINTRPT`	**y**	**resource**	**attribute**	action	n/a	`cpxOK`
`cpxACTION`	**y**	**resource**	**attribute**	action	n/a	`cpxON`, `cpxOFF`, `cpxNOP`, `cpxROATT`, `cpxBADRES`, `cpxBADATT`, `cpxBADACT`
`cpxREGDIR`	**y**	**resource**	n/a	action	value	`cpxOK`, `cpxEOPEN`, `cpxBADRES`, `cpxBADACT`
`cpxREGBYT`	**y**	**resource**	n/a	action	value	`cpxOK`, `cpxEOPEN`, `cpxBADRES`, `cpxBADACT`
`cpxLEDTST`	**n**	n/a	n/a	n/a	duration	`cpxOK`, `cpxEOPEN`, `cpxEALRM`, `cpxELAMP`, `cpxEARG1`
`cpxALMTST`	n	n/a	n/a	n/a	duration	`cpxOK`, `cpxEOPEN`, `cpxEALRM`, `cpxELAMP`, `cpxEARG1`

Table B-12: Extraction Event (Continued)

Class of Event	hsc	Arguments returned from cpxExtractEventDetail()				Status
		arg1	arg2	arg3	arg4	
cpxLEDSET	n	resource group	attribute group	action	n/a	cpxOK cpxEOPEN cpxEARG1 cpxEARG2 cpxEARG3
cpxPRGINT	**y**	**resource**	n/a	n/a	n/a	cpxEOPEN, cpxBADRES, cpxBADATT, cpxOK
cpxSETMSK	n	n/a	n/a	n/a	classes	cpxEOPEN, cpxOK
(all classes can experience cpxSEGFLT)	n	n/a	n/a	n/a	n/a	cpxSEGFLT (memory access error with application supplied pointers or values)

DIAGNOSTICS

cpxERROR is returned if this call is made and no notifications were retrieved.

cpxOK is returned indefinitely if a valid event was earlier retrieved.

Table B-13: Status & Action values

Value	Meaning
cpxOK	function call was successful.
cpxEOPEN	*hsc_Open()* not yet called.
cpxARG1	function argument one is invalid.
cpxARG2	function argument two is invalid.
cpxARG3	function argument three is invalid.
cpxARG4	function argument four is invalid.
cpxBADRES	invalid resource (individual or domain) specified.
cpxBADATT	invalid attribute provided.

Table B-13: Status & Action values

Value	Meaning
`cpxBADACT`	invalid action requested.
`cpxROATT`	attempt to modify read-only attribute.
`cpxON`	status of attribute is in the on state. action specified is to change attribute state to on.
`cpxOFF`	as above for off state.
`cpxNOP`	as above for no operation state.
`cpxNCHG`	a status of no change to the attribute. already in requested state.
`cpxSTATUS`	action request for status of an attribute.
`cpxELAMP`	a lamp test is in progress. request denied.
`cpxEALRM`	an alarm test is in progress. request denied.

(PAUSING EVENT NOTIFICATION)

cpxBlockEventNotification()

NAME

`cpxBlockEventNotification` - prevent or allow event signals to an application

SYNOPSIS

```
#include <cpxapi.h>

int cpxBlockEventNotification(action)

int action;
```

DESCRIPTION

This function allows or disallows signaling an application signed up for event notifications when one occurs. This function must be used judiciously. The current event queue is 45 deep for any application requesting events. If they are not retrieved in a timely manner, it is possible for them to overflow the queue and be lost (and duly noted on the system console). In most system configurations, this will hardly prove to be an issue because event rates are relatively low in a correctly functioning system. This function is used when an application wants to temporarily halt notifications during a critical processing phase.

ARGUMENTS

Valid actions are `cpxON`, `cpxOFF` or `cpxSTATUS` only.

DIAGNOSTICS

`cpxOK` is returned for a successful `cpxON` or `cpxOFF` action.

`cpxON` (blocked) or `cpxOFF`(unblocked) is returned if the action specified is `cpxSTATUS`.

`cpxNOFUNC` is returned if the calling application is not currently registered to receive event notifications.

`cpxBADACT` is returned for an invalid action.

`cpxEOPEN` is returned if an *hsc_Open()* has not been successfully performed.

SETTING/GETTING BRIDGE STATES

hsc_SetBridgeStates()

NAME

`hsc_SetBridgeStates` - clrs the special removal & insertion bridge states

SYNOPSIS

```
#include <cpxapi.h>
int hsc_SetBridgeStates(bridge, removal, insertion)
hsc_RESOURCE_ID bridge;
hsc_ACTION * removal;
hsc_ACTION * insertion;
```

DESCRIPTION

This function allows the clearing of special R/W bridge states which also are capable of generating interrupts. These states are used in conjunction with the *ejector_state* and *installed* status information to understand and control the comings and goings of the HSC/B card.

ARGUMENTS

Valid processor arguments are `hsc_BRIDGE_A` and `hsc_BRIDGE_B`.

The 2nd and 3rd arguments are `hsc_ACTION` variable pointers. If either of them is NULL(0), then it is ignored--status values are taken only from those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_BAD_OBJECT` is returned for if either `hsc_ACTION` argument is not `hsc_SET` or `hsc_CLEAR`.

`hsc_BAD_REGISTER` is returned for an invalid bridge argument.

hsc_GetBridgeStateSettings()

NAME

`hsc_GetBridgeStateSettings` - return removal/insertion state settings

SYNOPSIS

```
#include <cpxapi.h>
```

```
int hsc_GetBridgeStateSettings(bridge, removal,
insertion);
```

```
hsc_RESOURCE_ID bridge;
```

```
hsc_ACTION * removal;
```

```
hsc_ACTION * insertion;
```

DESCRIPTION

This function returns the insertion and removal states for a processor's bridge slot.

ARGUMENTS

Valid processor arguments are `hsc_BRIDGE_A` and `hsc_BRIDGE_B`.

The 2nd and 3rd arguments are `hsc_ACTION` variable pointers. If one or more of them is NULL(0), then it(they) are ignored--status values are written only to those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_TRUE` or `hsc_FALSE` is returned for each `hsc_ACTION` variable.

`hsc_BAD_REGISTER` is returned for an invalid bridge argument.

hsc_SetEBridgeStates()

NAME

`hsc_SetEBridgeStates` - sets extension bridge states

SYNOPSIS

```
#include <cpxapi.h>

int hsc_SetEBridgeStates(ebridge, removal, insertion,
installed)

hsc_RESOURCE_ID ebridge;

hsc_ACTION * removal;

hsc_ACTION * insertion;

hsc_ACTION * installed;
```

DESCRIPTION

This function allows the setting of special R/W bridge states, the first two of which also are capable of generating interrupts.

ARGUMENTS

Valid processor arguments are `hsc_EXT_BRIDGE_1` and `hsc_EXT_BRIDGE_2`.

The 2nd, 3rd and 4th arguments are `hsc_ACTION` variable pointers. If any of them is NULL(0), then it is ignored--status values are taken only from those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_BAD_OBJECT` is returned for if either `hsc_ACTION` argument is not `hsc_SET` or hsc_CLEAR.

`hsc_BAD_REGISTER` is returned for an invalid ebridge argument.

hsc_GetEBridgeStateSettings()

NAME

`hsc_GetEBridgeStateSettings` - gets the special removal & insertion bridge states

SYNOPSIS

```
#include <cpxapi.h>

int hsc_GetEBridgeStateSettings(ebridge, removal,
insertion)

hsc_RESOURCE_ID ebridge;

hsc_ACTION * removal;

hsc_ACTION * insertion;
```

DESCRIPTION

This function allows the acquisition of special R/W bridge states which also are capable of generating interrupts. These states are used in conjunction with the *ejector_state* and *installed* status information to understand and control the comings and goings of the HSC/B card.

ARGUMENTS

Valid processor arguments are `hsc_EXT_BRIDGE_A` and `hsc_EXT_BRIDGE_B`.

The 2nd and 3rd arguments are `hsc_ACTION` variable pointers. If either of them is NULL(0), then it is ignored--status values are taken only from those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_SET` or `hsc_CLR` is returned for each `hsc_ACTION` variable.

`hsc_BAD_REGISTER` is returned for an invalid extension bridge argument.

RETRIEVING STATUS ONLY

hsc_GetSlotStatus()

NAME

`hsc_GetSlotStatus` - return status conditions for a payload slot

SYNOPSIS

```
#include <cpxapi.h>

int hsc_GetSlotStatus(slot, present, healthy, connected,
reset_state, active, power, connect, force_healthy)

hsc_RESOURCE_ID slot;

hsc_ACTION * present;

hsc_ACTION * healthy;

hsc_ACTION * connected;

hsc_ACTION * reset_state;

hsc_ACTION * active;

hsc_ACTION * power;

hsc_ACTION * connect;

hsc_ACTION * force_healthy;
```

DESCRIPTION

This function returns the general status conditions for a particular payload slot.

It is important to note that the only status guaranteed to be valid regardless of domain control is the `hsc_PRESENT` status. The `hsc_RESET_STATE` is valid only if the slot's domain/bus is under control. All other status indicators are valid only if the *current* processor controls the domain/bus the slot is in.

ARGUMENTS

Valid slot arguments include all payload slots(`hsc_SLOT_01` through `hsc_SLOT_16`(CPX8216) or `hsc_SLOT_21`(CPX8221).

The 2nd through 6th arguments are `hsc_ACTION` variable pointers. If one or more of them is NULL(0), then it(they) are ignored. Status values are written only to those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_TRUE` or `hsc_FALSE` is returned for each `hsc_ACTION` variable.

`hsc_BAD_REGISTER` is returned for an invalid slot argument.

hsc_GetProcessorStatus()

NAME

`hsc_GetProcessorStatus` - return status conditions for a processor.

SYNOPSIS

```
#include <cpxapi.h>
int hsc_GetProcessorStatus(proc, healthy, connected,
active, power)
hsc_RESOURCE_ID proc;
hsc_ACTION * healthy;
hsc_ACTION * connected;
hsc_ACTION * active;
hsc_ACTION * power;
```

DESCRIPTION

This function returns the general status conditions for a particular processor.

ARGUMENTS

Valid processor arguments are `hsc_PROC_A` and `hsc_PROC_B`.

The 2nd through 5th arguments are `hsc_ACTION` variable pointers. If one or more of them is NULL(0), then it(they) are ignored--status values are written only to those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_TRUE` or `hsc_FALSE` is returned for each `hsc_ACTION` variable.

`hsc_BAD_REGISTER` is returned for an invalid processor argument.

hsc_GetPBayStatus()

NAME

`hsc_GetPBayStatus` - return status conditions for a peripheral bay.

SYNOPSIS

```
#include <cpxapi.h>
int hsc_GetPBayStatus(pbay, present, own_device, active,
power)
hsc_RESOURCE_ID pbay;
hsc_ACTION * present;
hsc_ACTION * own_device;
hsc_ACTION * active;
hsc_ACTION * power;
```

DESCRIPTION

This function returns the general status conditions for a particular peripheral bay in a CPX8216 chassis only.

ARGUMENTS

Valid peripheral bay arguments are `hsc_PBAY_1` .. `hsc_PBAY_4` inclusive.

The 2nd through 5th arguments are `hsc_ACTION` variable pointers. If one or more of them is NULL(0), then it(they) are ignored--status values are written only to those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_TRUE` or `hsc_FALSE` is returned for each `hsc_ACTION` variable.

`hsc_BAD_REGISTER` is returned for an invalid peripheral bay argument.

hsc_GetPSStatus()

NAME

`hsc_GetPSStatus` - return status conditions for a power supply

SYNOPSIS

```
#include <cpxapi.h>
int hsc_GetPSStatus(ps, ps_present, power_good,
cooling_alarm, cooling_fault, fan_present, fan_fault)
hsc_RESOURCE_ID ps;
```

```
hsc_ACTION *ps_present;
hsc_ACTION *power_good;
hsc_ACTION *cooling_alarm;
hsc_ACTION *cooling_fault;
hsc_ACTION *fan_present;
hsc_ACTION *fan_fault;
```

DESCRIPTION

This function returns the general status conditions for a power supply. The statuses returned are valid *only* if either one of the processors owns domain A. Moreover, if `ps_present` is *not* `hsc_TRUE`, then the other information retrieved is invalid.

ARGUMENTS

Valid power supply arguments are `hsc_PS_1`, `hsc_PS_2` and `hsc_PS_3`.

The 2nd through 7th arguments are `hsc_ACTION` variable pointers. If one or more of them is NULL(0), then it(they) are ignored--status values are written only to those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_TRUE` or `hsc_FALSE` is returned for each `hsc_ACTION` variable.

`hsc_BAD_REGISTER` is returned for an invalid power supply argument.

hsc_GetBridgeStatus()

NAME

`hsc_GetBridgeStatus` - return status conditions for an processor’s bridge

SYNOPSIS

```
#include <cpxapi.h>

int hsc_GetBridgeStatus(bridge, present, installed,
connected, ejector_state, active)

hsc_RESOURCE_ID bridge;
hsc_ACTION *present;
hsc_ACTION *installed;
hsc_ACTION *connected;
hsc_ACTION *ejector_state;
hsc_ACTION *active;
```

DESCRIPTION

This function returns the general status conditions for a processor's bridge.

ARGUMENTS

Valid extension bridge arguments are `hsc_BRIDGE_A` and `hsc_BRIDGE_B`.

The 2nd through 6th arguments are `hsc_ACTION` variable pointers. If one or more of them is NULL(0), then it(they) are ignored--status values are written only to those variables whose addresses are supplied.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_TRUE` or `hsc_FALSE` is returned for each `hsc_ACTION` variable.

`hsc_BAD_REGISTER` is returned for an invalid bridge argument.

hsc_GetEBridgeStatus()

NAME

`hsc_GetEBridgeStatus` - return status conditions for an extension bridge

SYNOPSIS

```
#include <cpxapi.h>
int hsc_GetEBridgeStatus(ebridge, present, installed,
connected, active, state)
hsc_RESOURCE_ID ebridge;
hsc_ACTION *present;
hsc_ACTION *installed;
hsc_ACTION *connected;
hsc_ACTION *active;
int * state;
```

DESCRIPTION

This function returns the general status conditions for an extension bridge.

ARGUMENTS

Valid extension bridge arguments are `hsc_EXT_BRIDGE_1` and `hsc_EXT_BRIDGE_2`.

The 2nd through 5th arguments are `hsc_ACTION` variable pointers. If one or more of them is NULL(0), then it(they) are ignored--status values are written only to those variables whose addresses are supplied.

The 6th argument is a pointer to an integer to receive the current state of the extension bridge. AT THIS TIME!, 0_{10} is for bus idle and 12_{10} for bus active. If the state pointer is zero (0), then no bus state can be returned.

DIAGNOSTICS

`hsc_OK` is returned for success.

`hsc_TRUE` or `hsc_FALSE` is returned for each `hsc_ACTION` variable.

An integer value is returned for state.

`hsc_BAD_REGISTER` is returned for an invalid extension bridge argument.

hsc_GetHSCRevision()

NAME

`hsc_GetHSCRevision` - get a Hot Swap Controller's revision number

SYNOPSIS

```
#include <cpxapi.h>
int hsc_GetHSCRevision(void)
```

DESCRIPTION

This function returns the HSC/B's hardware revision number for the domain the calling application is running in. This revision number is retrieved from the HSC/B board's configuration space during the loading of the HSC Driver.

ARGUMENTS

None

DIAGNOSTICS

An integer revision number is returned.

hsc_GetPSRevision()

NAME

`hsc_GetPSRevision` - get a power supply's revision number

SYNOPSIS

```
#include <cpxapi.h>
int hsc_GetPSRevision(ps)
hsc_RESOURCE_ID ps;
```

DESCRIPTION

This function returns the revision number for the power supply indicated. The revision number is valid only if domain A is under control by either processor and the power supply's hsc_PS_PRESENT bit is `hsc_TRUE`.

ARGUMENTS

The argument must specify a power supply in the range of `hsc_PS_1` .. `hsc_PS_3` inclusive.

DIAGNOSTICS

A positive integer revision number is returned.

`hsc_BAD_REGISTER` is returned if the power supply argument is invalid.

`hsc_ACTION_DENIED` is returned if domain A is not active.

hsc_GetAlarmPLDRevision()

NAME

`hsc_GetAlarmPLDRevision` - get the alarm panel's PLD revision number

SYNOPSIS

```
#include <cpxapi.h>
```

```
int hsc_GetALarmPLDRevision(void)
```

DESCRIPTION

This function returns the revision number of the system's alarm panel PLD. The revision level is valid only if domain A is owned by either processor and `hsc_PRESENT` is `hsc_TRUE` for the hsc_SYS_LED register [alarm board is installed].

ARGUMENTS

None.

DIAGNOSTICS

A positive integer revision number is returned.

`hsc_ACTION_DENIED` is returned if domain A is not active.

hsc_GetAlarmPanelRevision()

NAME

`hsc_GetAlarmPanelRevision` - get the alarm panel's assembly revision number

SYNOPSIS

```
#include <cpxapi.h>
```

```
int hsc_GetALarmPanelRevision(void)
```

DESCRIPTION

This function returns the revision number of the system's alarm panel assembly. The revision level is valid only if domain A is owned by either processor and `hsc_PRESENT` is `hsc_TRUE` for the `hsc_SYS_LED` register [alarm board is installed].

ARGUMENTS

None.

DIAGNOSTICS

A positive integer revision number is returned.

`hsc_ACTION_DENIED` is returned if domain A is not active.

hsc_Action

NAME

`hsc_Action` - sets, clears or status the HSC/B's register bits

SYNOPSIS

```
#include <cpxapi.h>
hsc_ACTION hsc_Action (register, object, action)
hsc_RESOURCE_ID register;
hsc_ATTRIBUTE object;
hsc_ACTION action;
```

EXAMPLE

```
if(hsc_ON == hsc_Action(hsc_SLOT_02, hsc_CONNECTED,
hsc_STATUS)
  hsc_Action(hsc_SLOT_02, hsc_LED2, hsc_ON);
```

DESCRIPTION

This function permits the application program to modify and/or status all R/W bits in the registers controlled by the HSC/B board, but to only status their R/O bits.

CAVEAT: It does *not allow* the interrupt bits(27 & 28) or operation bits (29 & 30) to be modified, however. For those registers which are capable of generating one, the function *hsc_Interrupt()* performs the programming and/or deprogramming of interrupts. Alternatively, h*sc_ProgramInterrupts()* and *hsc_DeProgramInterrupts()* may be used. The invoking of *hsc_Action()*--among other functions--with the action specified, causes the proper programming of the operation bits.

Enumerations have been defined for all slots/registers, their bit objects and the actions to perform on those bit objects. These enumerations have been set forth in “Required Enumerations” on page 90.

For `hsc_SET`(_ON), `hsc_CLR`(OFF), and `hsc_NOP` actions the return value should equal the requested action unless anomalies were detected. For statusing--`hsc_STATUS`--an `hsc_SET` or `hsc_CLR` action value (or their equivalents) should be returned. If a totally problematic request is made, then `hsc_ERROR` is returned.

ARGUMENTS

register - `hsc_RESOURCE_ID` is the slot or register upon which the action is to be performed. See "hsc_RESOURCE_ID Enumerations" on page 90 for an inclusive list of values.

object - `hsc_ATTRIBUTE` - this is the object identifier for the bit to be set, cleared, or status'd. See "hsc_ATTRIBUTE Enumeration" on page 96 for an inclusive list of values.

action - `hsc_ACTION` is the action to be performed upon the bit object specified. See "hsc_ACTION Enumerations" on page 95 for the inclusive list of valid operations.

DIAGNOSTICS

`hsc_ON` should be returned for a valid `hsc_ON` operation or if any bit object status'd is currently on(set).

`hsc_OFF` should be returned for a valid `hsc_OFF` operation or if any bit object status'd is currently off(clr).

`hsc_NOP` will always be returned for an action of `hsc_NOP`. No registers are modified or status'd for a no-operation request.

`hsc_NCHG` will always be returned for an `hsc_ON` or `hsc_OFF` operation upon a register's R/W bit object **already** in that state. However, *the requested action still will be performed.*

`hsc_BAD_REGISTER` will be returned for any invalid resource enumeration passed to *hsc_Action()*.

`hsc_BAD_OBJECT` will be returned for any invalid attribute enumeration passed to *hsc_Action()*.

`hsc_BAD_ACTION` will be returned for any invalid action enumeration passed to *hsc_Action()*.

`hsc_RO_OBJECT` will be returned for `hsc_ON` or `hsc_OFF` operations on a read only object (for example, the `hsc_CONNECTED` object and opposed to the `hsc_CONNECT` object).

`hsc_ACTION_DENIED` will be returned for `hsc_ON` or `hsc_OFF` operations on a read only register (for example, the `hsc_INT_STAT_A` register cannot be modified).

hsc_RegDirect

NAME

`hsc_RegDirect` - write, set, clear, or get a register directly

SYNOPSIS

```
#include <cpxapi.h>

int hsc_RegDirect (register, bitmask, action)

hsc_RESOURCE_ID register;

unsigned long *bitmask; /* an address must be passed */

hsc_ACTION action;
```

DESCRIPTION

This function permits a program to directly modify or retrieve the contents of an entire register without regard to specific bit objects and actions like those provided in *hsc_Action()*.

The bit mask should not contain the 2-bit sequence specifying the SET/WRITE/CLEAR action because the driver installs those bits based upon the action specified in the call. Whatever is there will be overwritten.

Only `hsc_READ` can be used with a R/O register.

The function returns the `hsc_ACTION` specified if the operation succeeds.

All bits in the mask are applied against the target register. No check is made to ensure that R/O bits aren't being written.

ARGUMENTS

register - The register name is as enumerated in `hsc_RESOURCE_ID` Enumerations, above.

bitmask - The unsigned long word pointed to by bitmask contains the bits which are to be set (`hsc_SET`), cleared (`hsc_CLR`), or written (`hsc_WRITE`). Or, it points to the unsigned long word which is to receive the register's current contents (`hsc_READ`).

action - The action is as enumerated in `hsc_ACTION` Enumerations, above.

DIAGNOSTICS

An `hsc_BAD_REGISTER` is returned if the register specified is not within the range of valid register/slot enumerations.

An `hsc_ACTION_DENIED` is returned for any modifying action attempted against a R/O register.

An `hsc_BAD_ACTION` is returned for an invalid operation. An `hsc_STATUS` is considered an invalid action.

An `hsc_NOP` is returned for an `hsc_NOP` action. No register contents are returned. Support for this action will soon be removed, so its use is discouraged.

An `hsc_READ`, `hsc_WRITE`, `hsc_ON` or `hsc_OFF` is returned if one of these actions requested was performed.

hsc_RegByte

NAME

`hsc_RegByte` - write or retrieve a register's SW scratch byte

SYNOPSIS

```
#include <cpxapi.h>

hsc_Action hsc_RegByte (register, byteval, action)

hsc_RESOURCE_ID register;

unsigned char *byteval; /* note that an address must be
passed */

hsc_ACTION action;
```

DESCRIPTION

This function allows access to the software scratch byte made available in many of the HSC/B board's registers. The HSCD does not use this byte, so it is available to the applications. This byte is read (`hsc_READ`) or write (`hsc_WRITE`) only. The concepts of set or clear do not apply. The concept of status does not apply. However, you can perform `hsc_NOP` operations till the cows come home.

If successful, an `hsc_READ` or `hsc_WRITE` is returned.

ARGUMENTS

register - The register name is as enumerated in hsc_RESOURCE_ID Enumerations, above.

byteval - The character pointed to by byteval contains the value to be placed in the designated register's scratch byte, or is the byte which is to receive the current value of the register's scratch byte.

action - The action is as enumerated in `hsc_ACTION` Enumerations, above.

DIAGNOSTICS

An `hsc_BAD_REGISTER` is returned if the register specified is not within the range of valid register/slot enumerations.

An `hsc_BAD_REGISTER` is returned if the register specified does not support a scratch byte.

An `hsc_ACTION_DENIED` is returned for any *modifying* action attempted against a R/O register.

An `hsc_BAD_ACTION` is returned for an invalid operation. An `hsc_STATUS` is considered an invalid action. The actions `hsc_SET` and `hsc_CLR` are also considered invalid actions.

An `hsc_NOP` is returned for an `hsc_NOP` action. No register contents are returned.

An `hsc_READ` is returned for for a successful `hsc_READ` action request.

An `hsc_WRITE` is returned for for a successful `hsc_WRITE` action request.

hsc_Interrupts

NAME

`hsc_Interrupts` - program/deprogram slot/domain register interrupts

SYNOPSIS

`#include <cpxapi.h>`

`int hsc_Interrupts`(domain, action)

`hsc_DOMAIN_ID` domain;

`hsc_ACTION` action;

`int hsc_Interrupts`(register,action)

`hsc_RESOURCE_ID` register;

`hsc_ACTION` action;

DESCRIPTION

This function programs or de-programs a register's (or a set of domain registers') capability to generate interrupts on a particular interrupt vector when interrupts are enabled. Programming is permitted only for those registers actually capable of generating interrupts.

The interrupt vectors used are as defined in the HSCD's device file (`hscdev`). At this time, only `PCI_A` interrupts are used.

ARGUMENTS

domain - `hsc_RESOURCE_ID's` are as specified in the `hsc_RESOURCE_ID` Enumeration list. It specifies the register for which interrupts are to be enabled or disabled.

Alternatively, `hsc_DOMAIN_ID`s are as specified in `hsc_DOMAIN_ID` Enumerations. The domain is that set of registers for which interrupts are to be enabled or disabled

action - `hsc_ACTION`s are as enumerated in `hsc_ACTION` Enumerations, above. The action is either to program (`hsc_ON`) interrupts or to deprogram (`hsc_OFF`) them. An `hsc_NOP` request performs no operation.

DIAGNOSTICS

If an `hsc_STATUS` is the action to perform, then either `hsc_ON` or `hsc_OFF` is returned to provide the current interrupts enabled state.

If `hsc_ON` or `hsc_OFF` is the action, then `hsc_ON` or `hsc_OFF` should be returned, respectively.

An `hsc_NOP` is returned for an action of `hsc_NOP` and no changes are made to the registers.

And `hsc_NCHG` is returned if the interrupts are already in the requested on or off state.

An `hsc_BAD_DOMAIN` is returned if the domain specified is not a single valid domain. The domain cannot be specified as `hsc_DOMAIN_A_B`.

An `hsc_ACTION_DENIED` is returned if the register specified does not support interrupts. Read only and non-interrupting registers are not considered valid.

An `hsc_BAD_REGISTER` is returned if the register specified is not within the range of valid register/slot enumerations. Read only and non-interrupting registers are not considered valid.

Appendix1: Changes in Version 2

This document describes version 2 of the API interface to the CPX8000 series Hot Swap controller. It is the beginning of a change from the old hsc_, bit oriented interface, towards the new cpx system resource interface. A major change is the introduction of resource aggregation, which permits a function to logically access more than a single resource at a time. *Depending upon which pre-release version 1 of the API you received and used earlier, some of this information may not apply to your development efforts. In such cases, it is offered as background material which may be of interest.*

What's New?

1. *cpxLED()* - This functions controls the state of all system LEDs. LED states now include `cpxON`, `cpxOFF`, cpxFASTBLINK (~.25 sec.), cpxMEDBLINK (~.5 sec.) and cpxSLOWBLINK (~1 sec.). Slots or other system functions who's LEDs are to be affected are specified as aggregates--one or more of them arithmetically OR'd together. The LEDs are also specified as aggregates of cpxLED*x* id's arithmetically OR'd together. Convenient enumerations have been defined as follows; cpxALL_SLOTS for all *payload* slots and `cpxALL_LEDS` for all LEDs available. When acquiring status though, only a single resource and single LED id may be specified, for obvious reasons. The *hsc_Action()* function should *not* be used if any blinking of LEDs will be used. Not only can blink not be specified in *hsc_Action()*, but the status of an LED can only take on the values of `hsc_ON` or `hsc_OFF`. If an LED is blinking, then *hsc_Action()* status for a particular LED will be indeterminate. New cpx-style enumerations are used in the *cpxLED()* function call. Special note: Alarms for the `cpxALARM` resource can also be blinked, even though the fourth alarm(`cpxRACK`) does not have an associated LED. Those enumerations are: `cpxCRIT`, `cpxMAJOR`, cpxMINOR and `cpxRACK`. The whole alarm aggregate may be designated using the `cpxALL_ALARMS` enumeration.
2. *cpxLampTest()* - allows all system LEDs to be lit for a given amount of time. This does *not* include `cpxALARM` LEDs. New cpx-style enumerations are used.
3. *cpxAlarmTest()* - allows all alarms for the `cpxALARM` resource to be activated for the requested amount of time. Three of these alarms also have LEDs tied to them. New cpx-style enumerations are used.
4. *cpxInstallEventFunction()* replaces *hsc_InstallEventFunction()*. It now accepts an aggregate of classes which specify the type of events being signed up for. Events comprise interrupt and non-interrupt entities. If all classes are wanted, then the enumeration for all of them is `cpxALLCLS`. New cpx-style enumerations are used.
5. *cpxSetEventList()* permits the list of classes specified in the *cpxInstallEventFunction()* to be replaced by a new set. New cpx-style enumerations are used.
6. *cpxRemoveEventFunction()* replaces *hsc_RemoveEventFunction()*. New cpx-style enumerations are used.

7. *cpxGetEvent()* replaces *hsc_GetEvent()*. It now returns the status and class of an event for which the function was signaled. New cpx-style enumerations are used for the *class* and *status* parameter.
8. *cpxExtractEventReason()* replacing *hsc_GetEventReason()*, acquires detail about the event which caused it to occur. The *action* parameter is valued as a new cpx-style parameter which loosely translates to the old `hsc_`-style enumeration in *meaning*, but not in value.
9. *cpxSetEventFunction() replaces hsc_SetEventFunction()*. New cpx-style enumerations are used.
10. *cpxBlockEventNotification()* prevents signals from the driver to the function signed up to handle events when they occur until they are unblocked. Upon unblocking, if any events have been queued for the function, then a signal will be sent immediately to the receiving function.
11. The substring '`ForceHealthy`' replaces '`IgnoreHealthy`' in function names.
12. *cpxPower() replaces hsc_PowerOff()*, *hsc_PowerOn()* and *hsc_GetPowerSetting()*. This new call permits the use of resource aggregates.
13. *cpxConnect() replaces hsc_Connect()*, *hsc_Disconnect()* and *hsc_GetConnectSetting()*. This new call permits the use of resource aggregates.
14. *cpxForceHealthy() replaces hsc_ForceHealthyOn()*, *hsc_ForceHealthyOff()* and *hsc_GetForceHealthySetting()*. This new call permits the use of resource aggregates.
15. *cpxInfoByte() replaces hsc_WriteInfoByte()* and *hsc_ReadInfoByte()*. This new call permits the use of resource aggregates.
16. *cpxProgramInterrupts() replaces hsc_ProgramInterrupts()*, *hsc_DeProgramInterrupts()* and *hsc_GetProgramInterruptsSetting()*. This new call permits the use of resource aggregates.
17. *cpxEnableInterrupts() replaces hsc_EnableInterrupts()*, *hsc_DisableInterrupts()* and *hsc_GetInterruptsSetting()*.
18. *cpxProgramEnums() replaces hsc_ProgramEnums()*, *hsc_DeProgramEnums()* and *hsc_GetProgramEnumsSetting()*.

19. *cpxFanHigh() replaces hsc_FanHighOn()*, *hsc_FanHighOff()* and *hsc_GetFanHighSetting()*. This new call permits the use of attribute aggregates.

20. **cpxtool** *replaces* hsctool. It permits access to the new functionality.

    - use 'L' to access *cpxLampTest()*; simply enter **L** for details.
    - use 'A' to access *cpxAlarmTest()*; simply enter **A** for details.
    - use 'l' to access *cpxLED()*; simply enter **l** for details.

21. **cpxapi.h** *replaces* hscd.h for the application writers. hscd.h will *not* be distributed and contains info for use only in building our distribution.

22. **libcpxapi.a** *replaces* libhscd.a.

23. **.cpxtoolrc** *replaces* .hsctoolrc

24. cpxtool users!

    - You can now use c1 and c2 in lieu of / addition to xfr_ctl1 and xfr_ctl2. Also, s1 and s2 in addition to xfr_stat1 and xfr_stat2. Should reduce typing fatigue for those of you grabbing domains from the busa and busb registers. Of course, you could have used macros....but I know a lot of you never bothered! ;-)
    - Execute only "menus" have been introduced to simplify access to frequently executed script fragments or as block of processing to be used with the ....

        -New "if..then..else" capability.

        -Ability to get user input into variables and well as to set variables.

what's gone?

1. *hsc_GetGeneralStatus()* - this capability is exclusively used by cpxtool.

2. enumerations for `hsc_DOMAIN_OTHER` and `hsc_DOMAIN_NONE` removed.

3. The following tracing capability has been *replaced* by the *event* mechanisms discussed in What's New above:

    - *hsc_SetTraceLevel()* - nominally **replaced** by *cpxSetEventList()*

- *hsc_SetTraceFunction()* - **replaced** by *cpxSetEventFunction()*
- *hsc_GetTraceCode()* - nominally **replaced** by *cpxGetEvent()*
- *hsc_InstallTraceFunction()* - **see** *hsc_SetTraceFunction()* above
- *hsc_RemoveTraceFunction()* - **see** *hsc_SetTraceFunction()* above
- *hsc_BuildTraceMsg()* - no equivalent

What's Changed?

1. `hsc_ACTION_REG_ID` is now `hsc_RESOURCE_ID`; adjust your sources!
2. `hsc_BIT_OBJECT` is now `hsc_ATTRIBUTE`; adjust your sources!
3. *hsc_GetSlotStatus()* has three(3) additional arguments. See "hsc_GetSlotStatus()" on page 147 for details.
4. *hsc_GetPBayStatus()* has one(1) additional argument. See "hsc_GetPBayStatus()" on page 148 for details.
5. `cpxProgramInterrupts` *replaces* *hsc_ProgramInterrupts()*, *hsc_DeProgramIntrrupts()* and *hsc_GetProgramInterruptsSetting().*
6. *hsc_GetProcessorStatus()* has one(1) additional argument. See "hsc_GetProcessorStatus()" on page 148 for details.
7. Most `hsc`_Style API return codes have been changed to either `hsc_OK` (for success), or a *negative* value indicating the reason for failure--such as `hsc_BAD_REGISTER`, etc. Adjust your sources to accept `hsc_OK`.
8. The use of many old API calls is now discouraged in light of several replacement calls. The old calls are shown under "CPX8000 System HSC Driver Control Functional Interface" on page 101 in a lightened and italicized font. Their descriptions are in a lightened font. At some future date, they will be removed from the API, so timely conversion to their replacements is encouraged.
9. *hsc_Open() no longer supports* an argument to specify a different device node name. It expects `/dev/hsc0` to be present.
10. To maintain access to all enumerations used for test code, API's and tools written at MCG, two header files must be included in the following order:

```
#include cpxapi.h /* for all users of the API */
```

`#include hscd.h`/* for MCG use only - not distributed */

11. *End users* developing code which uses the API must include **cpxapi.h** *only*; upon successful compilation, the object module(s) must be linked with the archive **cpxapi.a** (or object **cpxapi.o**) to be able to access the API.

Programmer's Notes

This version of the API is significantly different from all previous versions. In all likelihood, developers using this API will need to recompile and relink against this new version. The good news is that newer versions of the API, when available, will remain backward compatible with this version. To determine if you need to rebuild your application, note the following uses of the old API which would require a new compile and build against the new one.

1. If you have used ANY interrupt or tracing mechanisms, you will need to change your code and rebuild. Both interrupts and driver access are treated as events and reported via the same mechanism.
2. If in using higher level functions, such as *hsc_PowerOn()* or *hsc_PowerOff()*, and you paid attention to any non-negative values, you will probably have to change your code and rebuild. Most of these functions have been altered to return a negative error enumeration or a value of `hsc_OK`. If your code checked for `hsc_ON`, `hsc_OFF` or `hsc_NCHG` specifically, then you may have a problem. By the way, *hsc_PowerOn()* and *hsc_PowerOff()* have been replaced by *cpxPower()*. You should change your code anyway as the old calls are considered obsolete and will disappear at some later date.
3. 3. If you used any status gathering calls such as *hsc_GetSlotStatus()*, you will need to alter your argument lists as additional arguments are now returned.

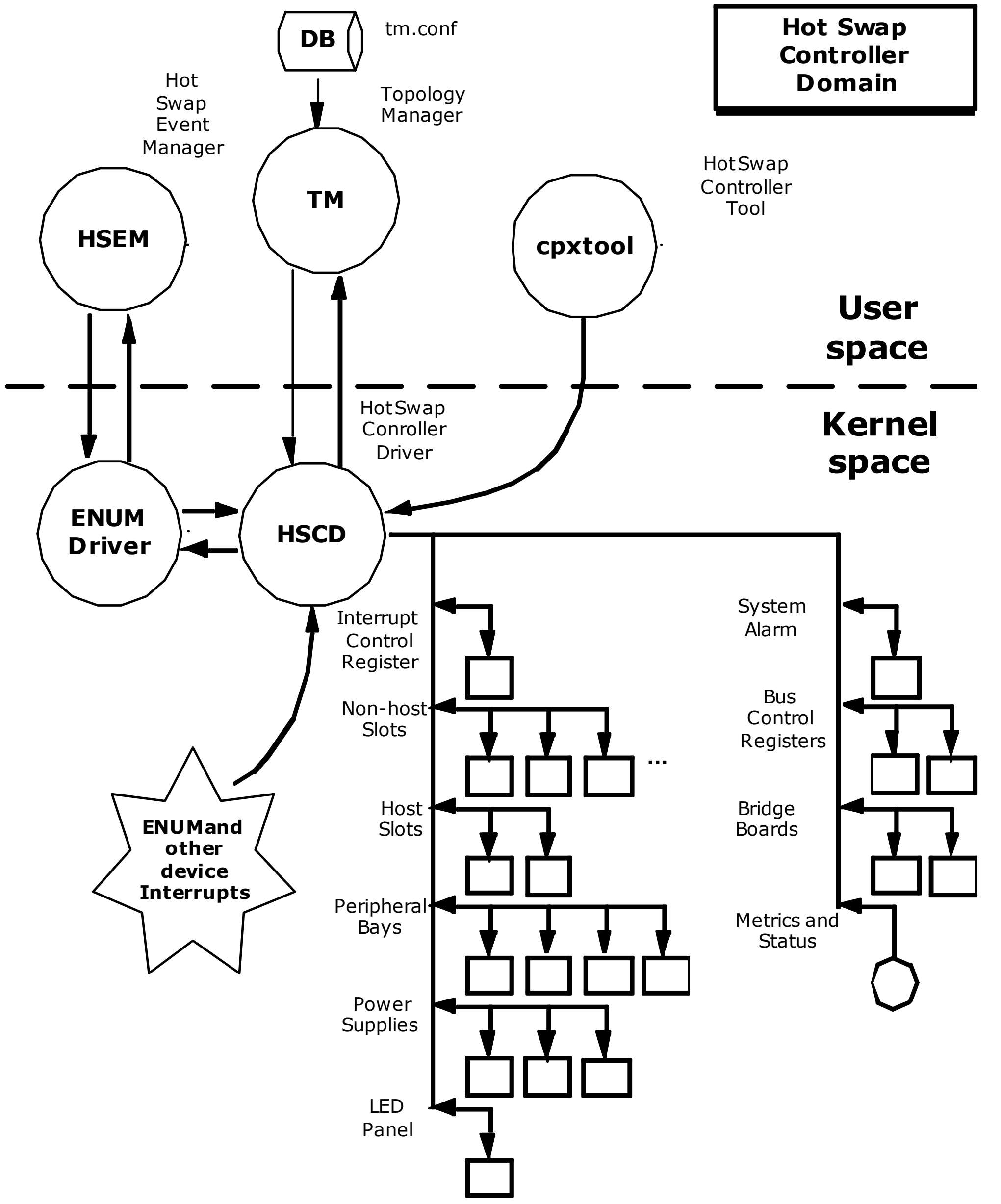
Appendix 2: Programming Information & Considerations

The following notes may be of use to the development engineer in understanding how the programming of the HSC/B affects the access to, and operation of, the CPX8xxx chassis.

- In a single chassis, there are one or two processors with hsc controller/bridges and one or two extension bridges (CPX8221 only) residing in two domains called A and B. Although each domain's processor can run stand alone, it may, when appropriate, take control of the other processor's resources--its domain. Each processor may have control over no domains, one domain or both domains. Having control of a domain means having control over the 6 primary, non-host slots in that domain. In the case of Domain A, additional system resources (alarms, system LEDs, etc.) are also controlled. On the CPX8221, having control of Domain B also means having control over the 5 additional payload slots on Bus C, for a total of 11 payload slots in that domain.

- The buses in Domain B may be taken one at a time. In this scenario, bus B must be taken before control of bus C can be realized. Control of bus C *by itself* is not possible (CPX8221 only).
- A payload slot's board *present* status is valid only when power is *not* applied to the slot. When power is applied, the *healthy* status would indicate a board being present and operational. If power is applied, but healthy is not asserted, then the board is either absent, non-cPCI Hot Swap compliant or defective. If it is non-compliant, then the `cpxForceHealthy()` function may be used to cause the *healthy* status to reflect the *power on* state.
- For purposes of this document, a *status* change typically results from a *setting* being changed or some asynchronous event occurring. For example, when *power* is applied to a board (a setting of power to on), an event usually takes place in that its *healthy* status is asserted. Therefore, the functions described herein make the distinction between setting, acquiring the value of a setting and acquiring a status value. Settings are deterministic, driven by software. Statuses track state transitions, a read-only condition which reflects a resource's change of state.
- Having control of domain A also allows the processor to control system functions such as alarms, system LEDs, power supplies and fans. Taking control of domain A by an application executing on the processor installed in domain B is accomplished by taking control of bus A. Again, taking control of buses is better accomplished by specifying a domain as opposed to a bus-by-bus method. Once a domain takeover bid is made, it cannot be rescinded except under very specific conditions.
- A domain (hence its associated bus) may be taken by the other domain's processor either cooperatively or 'by force'. A processor's own domain (and bus) may be taken back from the other processor either cooperatively or 'by force'.
- Just as a domain may be taken by a processor, it may also be given up *without* the other processor's intervention.

- When a domain is owned, its slots and system functions are called '*active*,' Otherwise, they are '*inactive.'* In most instances, the API calls may be made against inactive slots and system functions, but they will be *pending* the takeover of the bus by the programming processor and will be effected only *after* the domain is taken. Domains are taken by taking control of the *primary* bus in that domain.
- Both processors in the CPX8216 can have simultaneous control over the four peripheral bays based upon a hardware strapping option. The CPX8221 has no peripheral bays under hot swap control, which makes this point moot.
- A processor may *not* cut its *own* power. It can cut the power to the other domain's processor though.
- A processor removing/restoring power from/to a power supply must also change the LEDs to reflect its new operational status before the operation can actually occur. This is done automatically by the functional interface.
- The *last* powered on power supply in a system *cannot* be powered off under software control. Upon "powering off" the last supply, it does so only momentarily, then comes back on, causing both processors to begin their reboot sequence.
- For peripheral bays owned by both processors (a drive jumper setting), either processor can turn the LEDs on. However, both must turn the same LEDs off to actually make it happen. This action is specific to the CPX8216, which has controller managed peripheral bays.
- Performing non-host slot actions regarding power, connections, LEDs, etc., in a domain *not* currently owned by the processor can be done in preparation for taking the domain. As soon as this takeover is accomplished, the last *pending* actions performed upon them take effect.
- All functions and slots may be programmed (or deprogrammed) to interrupt with a single command, or they may be targeted singly. The application may also declare a function to receive their interrupt information when exceptions arise. This function must, in turn, make one or more calls to retrieve all the interrupt reasons.



Hscd Context Diagram

APPENDIX C *stmd.conf file example*

This appendix provides a sample `stmd.conf` file for generic use. For additional information on configuring `stmd.conf`, please see the man page `stmd.conf(5)`

```
$Id: stmd.conf,v 1.4 2000/12/13 02:43:15 carlb Exp $
;*************************************************************
; (C) Copyright 2000
; LynuxWorks, Inc.
; San Jose, CA
; All rights reserved.
;
; $Date: 2000/12/13 02:43:15 $
; $Revision: 1.4 $
; $Source: /home/Lynx/src/HAP2.0/stmd/RCS/stmd.conf,v
; $
;*************************************************************
;-------------------------------------------------------------
;$State: Exp $ by $Locker:  $
;-------------------------------------------------------------

; static char sccsid[] = "@(#) hsem/hsem.conf, hsem, Phase0   1.8     99/02/02";
;
; COMPONENT_NAME: (HSEM) Hot Swap Event Manager
;
; Copyright (c) 1998,1999 MOTOROLA
; All Rights Reserved
;
; THIS IS UNPUBLISHED SOURCE CODE OF MOTOROLA.
; The copyright notice above does not evidence any actual or
; intended publication of such source code.

; The "slots" entry defines the slot table for the specific chassis.
; Field 1 = Physical Slot number
; Field 2 = Bus Number
; Field 3 = Device Number
; Field 4 = Function Number
; The slot table below is for the domain A side of an 8216 Chassis.
; It should not be modified.
slots
;slot#,bus#,dev#,func#
slot 1,1,14,0
slot 2,1,13,0
slot 3,1,12,0
slot 4,1,11,0
slot 5,1,10,0
slot 6,1,9,0
slot 7,0,0,0
slot 8,0,0,0
slot 9,0,0,0
slot 10,0,0,0
slot 11,42,14,0
slot 12,42,13,0
slot 13,42,12,0
slot 14,42,11,0
slot 15,42,10,0
slot 16,42,9,0
end_slots

; This is the drivers section of the config file.  In it are
; sections for each driver.

drivers


driver
device_name_prefix /dev/wan

; This signature is used during device parsing.
; It corresponds to the PCI device specific vendor ID.

signature
vendor 0x00091011,0,255
end_signature

; The script invoked during device insertion transition.

device_install
system /etc/hasw/install.scr
end_device_install

; The script invoked during device removal transition.

device_uninstall
system /etc/hasw/uninstall.scr
end_device_uninstall

; NOTE these scripts MUST exist or these lines should be omitted
;      placing these lines but omitting the scripts will cause
;      all standby and active trasitions to return error
;      the net effect of this would be that the driver would
;      remain in Inactive state
; The script invoked during domain transition to active.

active_command /etc/hasw/active.scr

; The script invoked during domain transition to standby.

standby_command /etc/hasw/standby.scr

end_driver

end_drivers.
```

NOTE: The `active_command` and `standby_command` scripts must exist if they are declared in the file. If these scripts are not in the declared location, all transitions to Active or Standby fail and the driver will be stuck in the Inactive state. If the `active_command` and `standby_command` declarations are left out, transitions to Active and Standby are still allowed.

Index