Pigeon Point Systems

Hot Swap Kit
User Guide

Version 20060310

PCI INDUSTRIAL COMPUTERS PICMG™

CompactPCI®

Hot Swap Kit

This manual, as well as the software described within, is furnished under license and may be used or copied only in accordance with the terms of such license. The contents of this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Pigeon Point Systems. Pigeon Point Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

Table of Contents

1. Introduction

Congratulations on your acquisition of the Pigeon Point Hot Swap Kit (HSK) for Windows 2000/XP/2003. HSK is system software that works with the Plug and Play (PnP) facility of Windows 2000/XP/2003 to support CompactPCI hot swap.

HSK provides support for hot swapping peripheral CompactPCI boards.

The support for hot swapping peripheral CompactPCI boards is generic and conforms to the PCI Industrial Computer Manufacturers Group (PICMG) CompactPCI Hot Swap Specification. The HSK is architected for wide applicability across CompactPCI systems.

This release of the HSK product and documentation supports the following CompactPCI systems:

- Force Centellis 8730
- Motorola CPX100x
- Motorola CPX200x
- Motorola CPX8216
- Intel ZT5082
- Intel ZT5083
- Intel ZT5084
- Intel ZT5085
- Intel ZT5087
- MIC-3038
- MIC-3041
- MIC-3081
- MPCHC5091
- Westek P5100

The CPX8216 is supported, however HSK does not support the High Availability features found on this platform. Please contact Pigeon Point Systems if you would like support of the High Availability features.

1.1 Hot Swapping Peripheral CompactPCI Boards

Windows 2000/XP/2003 provides extensive support for Plug and Play technology, allowing dynamic insertion and removal of devices in a computer system without stopping the system. However, there is no built-in support in the operating system for dynamic insertion and removal of CompactPCI devices. Additional software, provided within the HSK package, collaborates with the system to provide hot swap support for CompactPCI. The core of this software includes the following drivers:

- busfiltr.sys – the CompactPCI bus filter. This filter driver positions itself between the PCI bus driver and functional device drivers. Its purpose is to monitor CompactPCI topology and communicate with the PCI bus driver.

- hspci.sys – the hot swap system driver. This driver is responsible for handling the specifics of CompactPCI hot swapping as described in the specification: the HS_CSR register on the board, the ENUM# (enumeration) signal and the blue hot swap LED on the board.
- enumdrv.sys – the platform driver. This driver is responsible for handling the ENUM# signal for the target platform. (Since the specification does not define the implementation of the ENUM# signal for the platform, it is platform-dependent). If a platform does not support the ENUM# signal, the HSCK will do the periodic polling of the configuration space to detect a change.
- Alternate HS_CSR miniport drivers. These drivers handle non-standard implementations of the HS_CSR register for specific boards.

1.2 System Requirements

Windows 2000, 2003 (Professional, Server, or Advanced Server) or Windows XP (Home or Professional) with memory and processor as recommended for the specific Windows 2000/XP/2003 variants.

For the Intel NetStructure ZT5524 hardware platform under Windows XP, the HAL installed in the system should be “Standard PC” (by default, the MPS HAL is installed during normal installation of Windows XP). In case of Windows2000 the HAL must be MPS Multiprocessor HAL. Please, pay attention that these restrictions make sense for all ZT5524 SBC variants: uniprocessor (ZT5524A-1B), symmetrical multiprocessor (ZT5524A-1A), uniprocessor (MPCBL5524A1-D) with 4GB physical memory support and symmetrical multiprocessor (MPCBL5524A1-C) with 4GB physical memory support.

For the Intel NetStructure ZT5524 hardware the QLogic drivers need to be installed for the resource re-assignment on the PCI bus (that is performed by the HSK) to work properly. If these drivers are not installed, the system during boot is not able to relocate resources occupied by the QLogic devices, to different locations. The reason for that is that it assumes that these devices, being boot devices, are supported by the BIOS directly (instead of being supported by Windows drivers). For the Hot Swap to work, it is necessary to install these drivers. We would recommend to install QLogic drivers before installing HSK, HSK+RHOST.

This is the direct link to QLOGIC windows driver

ftp://download.qlogic.com/drivers/23451/q23w32ScsiV90110.exe.

The Intel NetStructure ZT5524 is officially supported under Windows 2000 SP4 only.

A CD-ROM drive is required for installation from the Pigeon Point Hot Swap Kit CD.

1.3 Supported Hardware

The HSK is a general purpose product that can support generic CompactPCI platforms. This release emphasizes the following set of platforms, which will be automatically configured upon installation:

Platform	System Controller
Force Centellis 8730	CPCI730
Motorola CPX100x	CPV5350, CPV5370
Motorola CPX200x	CPV5000, CPV5300, CPV5350, CPV5370
Motorola CPX8216	CPV5350, CPV5370
Ziatech 5082B	ZT5531
Ziatech 5082C	ZT5531
Ziatech 5083 / 5084	ZT5550
Intel NetStructure 5085C	ZT5524, MPCBL5525
Ziatech 5087	ZT5503
MPCHC5091	ZT5524, MPCBL5525
MIC-3038	MIC3358
MIC-3041	MIC3389
MIC-3081	MIC3369
Westek P5100	MIC3358
Westek P5100	Westek

The HSK is capable of supporting hot swap operations on any hot-swap-specification-compliant peripheral CompactPCI board whose functional device driver fully supports the Windows 2000/XP Plug and Play device driver model. Some boards may use an alternate implementation of the Hot Swap Control/Status Register (HS_CSR) and require a special Alternate HS_CSR driver to interface to that hardware. High Availability features of the CPX8216 are not supported with HSK. Pigeon Point Systems offers the product “Hot Swap Controller Kit” to support these features.

1.4 Redundant Host Support Software

Redundant Host (RHOST) support is the next level of the High Availability. It’s purpose to provide users with more precise control over the chassis and it’s components: slot’s power control, domain’s switchover and etc. Current version of Windows HSK may work with the Windows Redundant Host Support level which may be installed over Window HSK.

A particular Redundant Hot Support distribution is designed for the following platforms:

Platform	**System Controller**
Intel NetStructure ZT5085	ZT5524
Intel NetStructure ZT5084	ZT5550

Redundant Host Support Software works under the Windows 2000 SP4 and its variations.

The Redundant Host Systems are defined in the PICMG 2.12 R2.0 standard. A special RH API for controlling RH aspects of the system operation is also defined in that standard. This section provides a brief overview of the Redundant Host concepts provided in that document and describes the RH API.

By definition, a Redundant Host System is a system consisting of one or more hosts and one or more domains. An ownership relationship is defined between hosts and domains in this system: hosts own domains. At any given moment of time, one domain can be owned by no more than one host. If a host owns a domain, software on the host has access to PCI devices in (or behind) the PCI slots of the domain.

A domain is a collection of peripheral PCI slots that is the unit of ownership by the hosts. These slots can be populated by PCI-PCI bridges, so the domain is generally a collection of PCI trees (a forest). For some system architectures using the RH term “domain” may be confusing, since these architectures also define their own domains, which not necessarily coincide with the domains used in RH API. Thus, in the rest of this document, we will use the term “RH-domain”, when talking about domains in the RH API meaning.

Both hosts and RH-domains are identified by their unique, not necessarily sequential, numbers. In the RH API, these numbers are represented by 32-bit values. The number of hosts and RH-domains and their numbers are assumed static and not changing at run-time.

The process of transferring ownership of a particular RH-domain from one host to another is called a domain switchover. Switchovers can be divided into aggressive and non-aggressive. Non-aggressive switchovers involve shutting down the devices controlled by the current host and then restarting each device from the beginning after the switchover on the newly active host, while aggressive switchovers assume preserving to some extent the state of the devices constituting the RH-domain and their drivers.

There is an alternative classification of switchovers, which names them as “cold” (non-aggressive), “warm”, and “hot”. We use the terms “non-aggressive” and “aggressive” switchover because there is no universally agreed definition that characterizes “warm” domain switchovers nor of the additional characteristics that must be present to accurately claim “hot” switchover support. It’s all a matter of the degree to which device and driver state is preserved and down time minimized across the switchover.

2. Installation

2.1 Proper Sequence of the Pigeon Point Hot Swap Kit and the Redundant Host Software Installation / Deinstallation

The Redundant Host Software can be installed on ZT5524 on ZT5085 and ZT5550 on ZT5084 hardware configurations. To successfully install / uninstall both packages the proper installation and deinstallation sequence must be kept.

The proper installation sequence is the following:

install the HSK package
reboot the system
install the RH package
reboot the system
repeat the same actions on the peer system

The proper deinstallation sequence is the following:

switch both domains to the one host - so it becomes the Active Host
start deinstallation procedures on the Backup Host
uninstall the RH package
reboot the system
uninstall the HSK package
reboot the system
repeat the same actions on the Active Host

2.2 Installing the Pigeon Point Hot Swap Kit

Insert the Pigeon Point Hot Swap Kit CD into your system. Your system should be running Microsoft Windows 2000 or Windows XP. To begin installation, run the setup.exe program off of the CD. The HSK setup screen below will be shown.

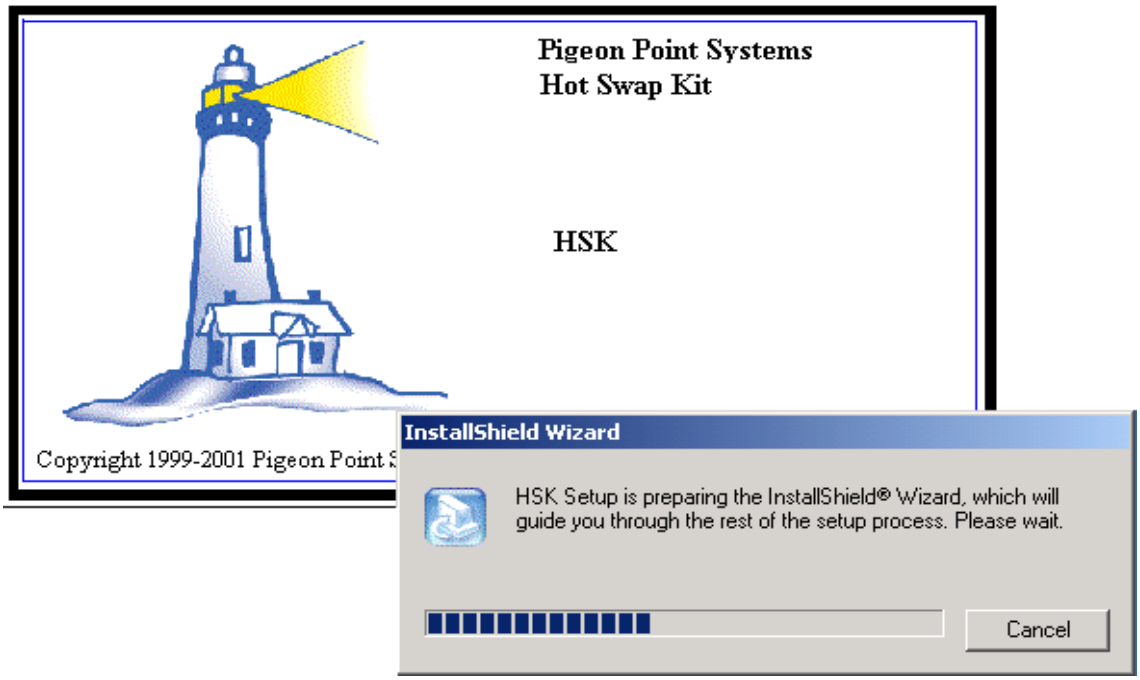

Fig 2.1: Initial Setup preparation screen

After the setup has loaded all of the necessary files, the Installation welcome screen will be shown.

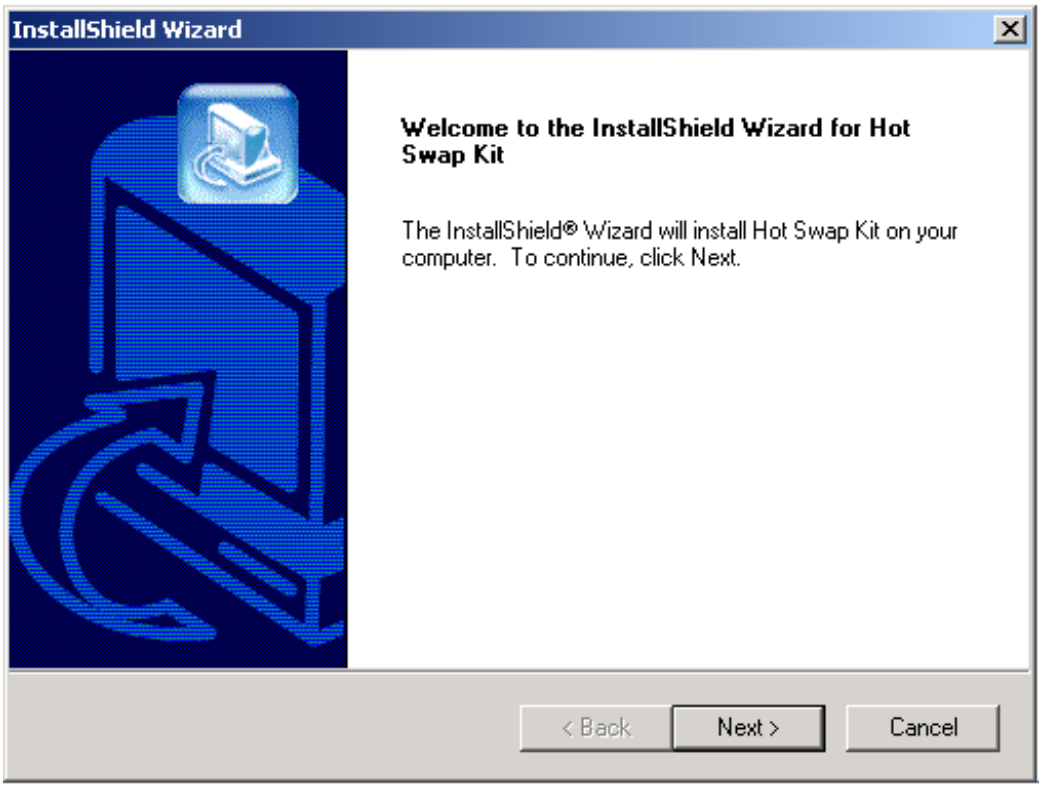

Fig 2.2: Installation Welcome Screen

Click Next to proceed with the installation.

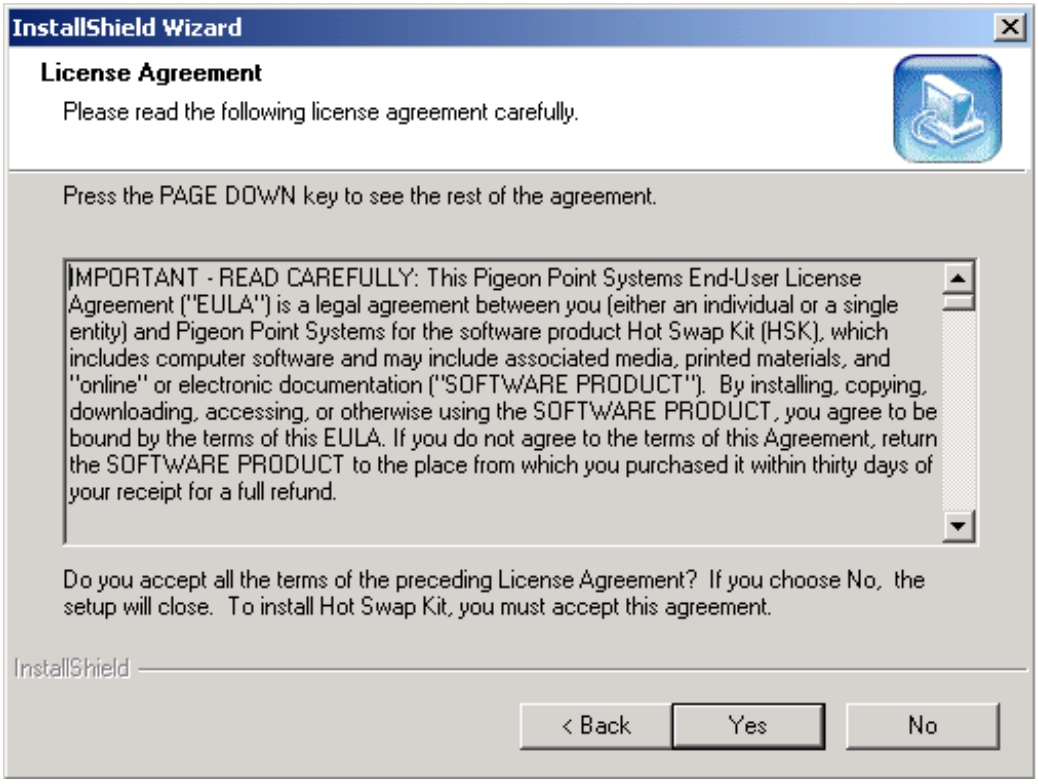

Fig 2.3: License Agreement

Please read the License Agreement fully before giving your consent. If you do not agree with the license agreement, please return all Pigeon Point HSK product packaging, without installing the software, back to the place of purchase. If you agree with the license agreement, select the “Yes” button.

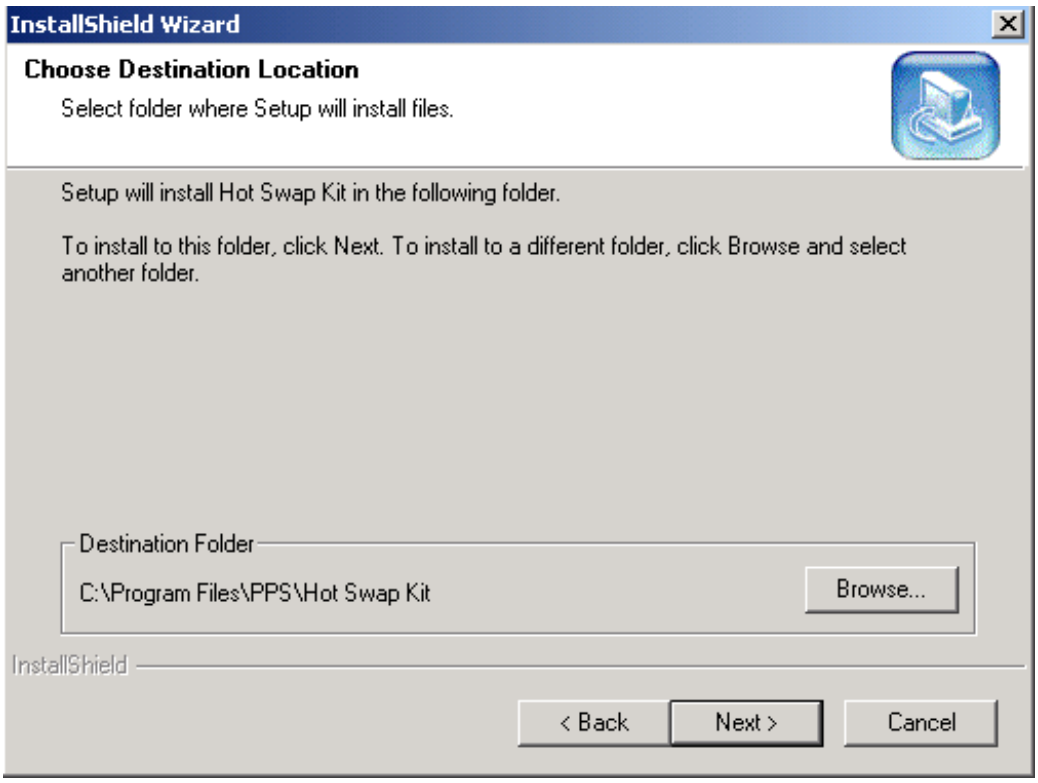

Fig 2.4: Choosing the Destination Folder

The default target directory for the installation is "C:\Program Files\PPS\Hot Swap Kit". If you would prefer to install the HSK in another location, you may browse through your system to select another location.

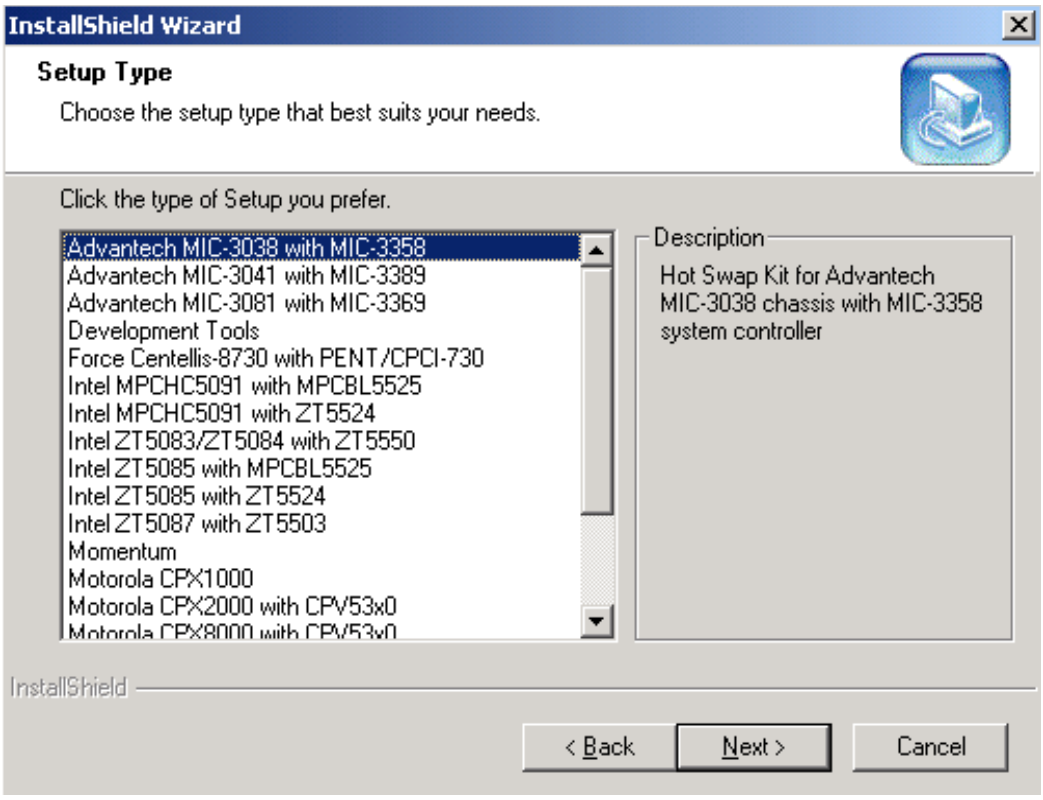

Fig 2.5: Choosing the system that you want supported.

Please select the system that you want supported. If you are developing applications on top of the HSK API and do not have a system connected, select “Development Tools”. In the event you do not find the specific platform you are using, choose “Other Platform”. Once you have selected, press the “Next>” button.

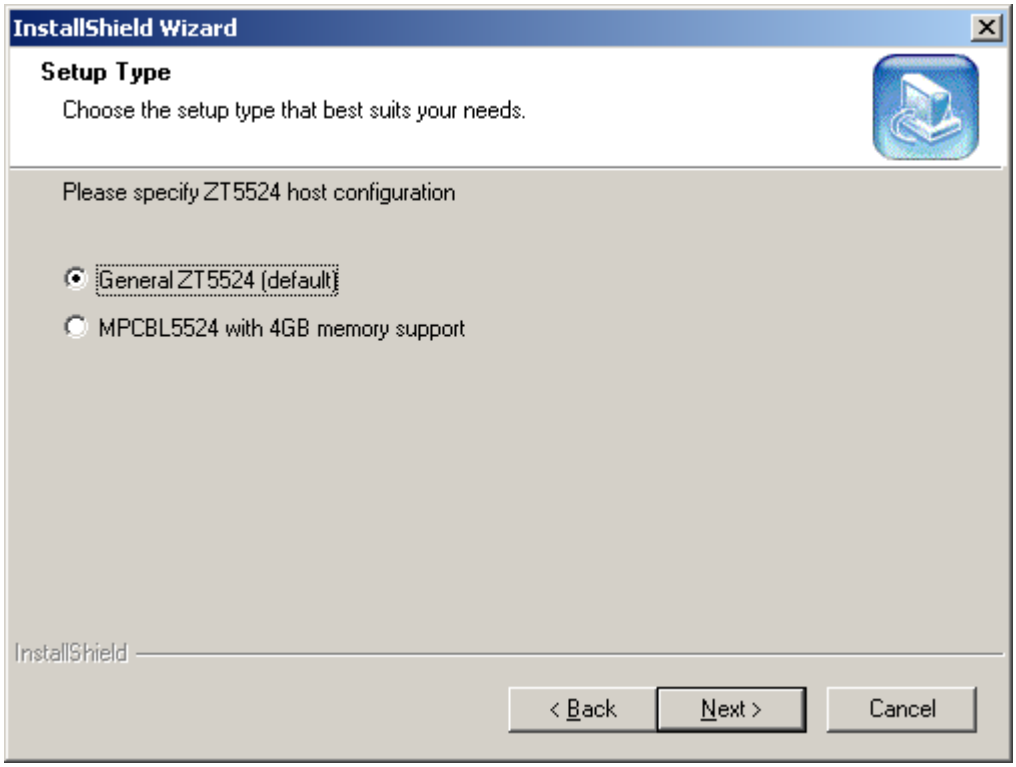

Fig 2.6: Specifing ZT5524 host controller configuration.

If you are installing on a ZT5085 or MPCHC5091 systems with ZT5524 host controller, you have to specify ZT5524 host configuration as well. There are two configurations for ZT5524 host controller:

- the first - "General ZT5524" configuration does not support 4GB physical memory;
- while the second, it is called "MPCBL5524 with 4G memmory support" does.

Choose proper configuration and press "Next>".

Note: If you have chosen "MPCBL5524 with 4G memory support" configuration the installer will update 'boot.ini' file automatically. It will set '/maxmem' parameter in accordance with chassis setup type. Correct value of '/maxmem' parameter for "Intel ZT5085 with ZT5524" configuration is 3584 and "Intel MPCHC5091 with ZT5524" uses 3840. Please don't change this value or system will be unusable.

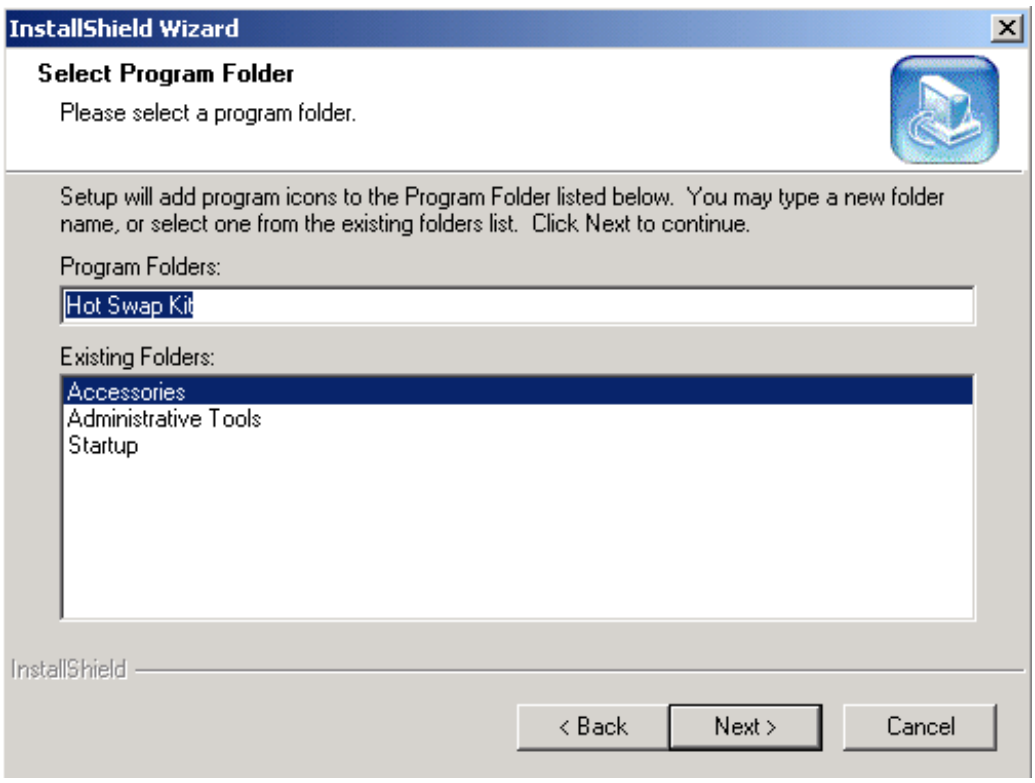

Fig 2.7: Choosing Program Folder for HSK to be installed.

Choose the Start Menu program folder to install the HSK utilities and press “Next>”.

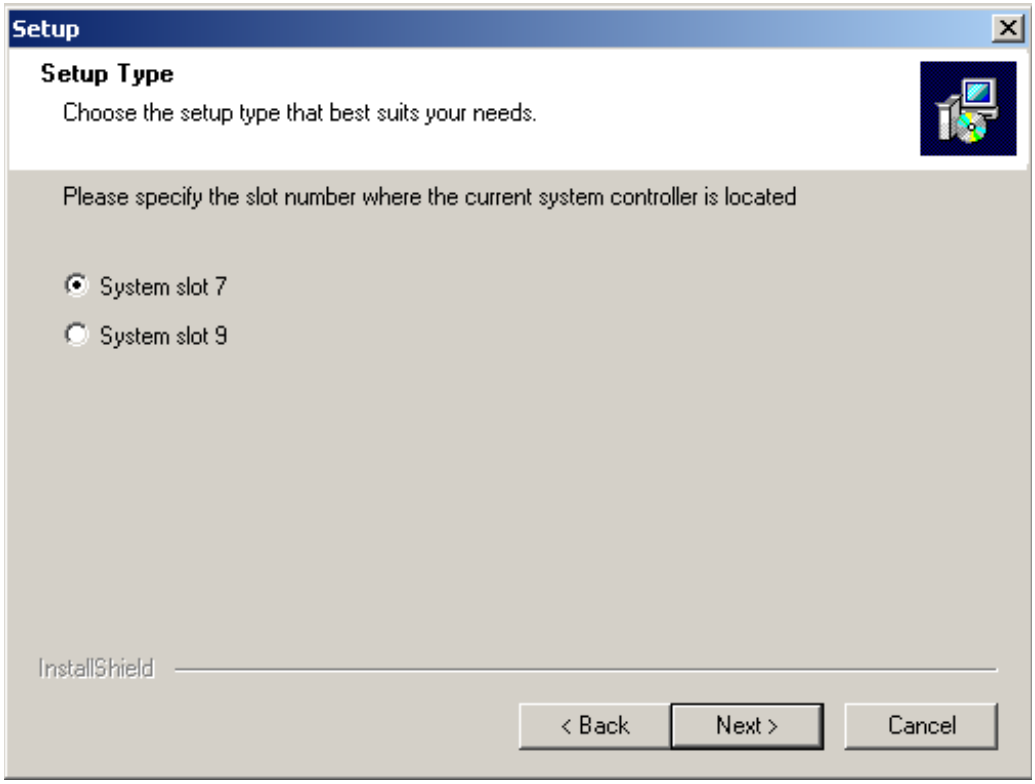

Fig 2.8: Installing the files to their proper location

If you are installing on a CPX8216 system, you will also need to enter in the system slot that you are installing on as pictured above. The system slot on the left has a physical slot of 7 and the system slot on the right has a physical slot of 9.

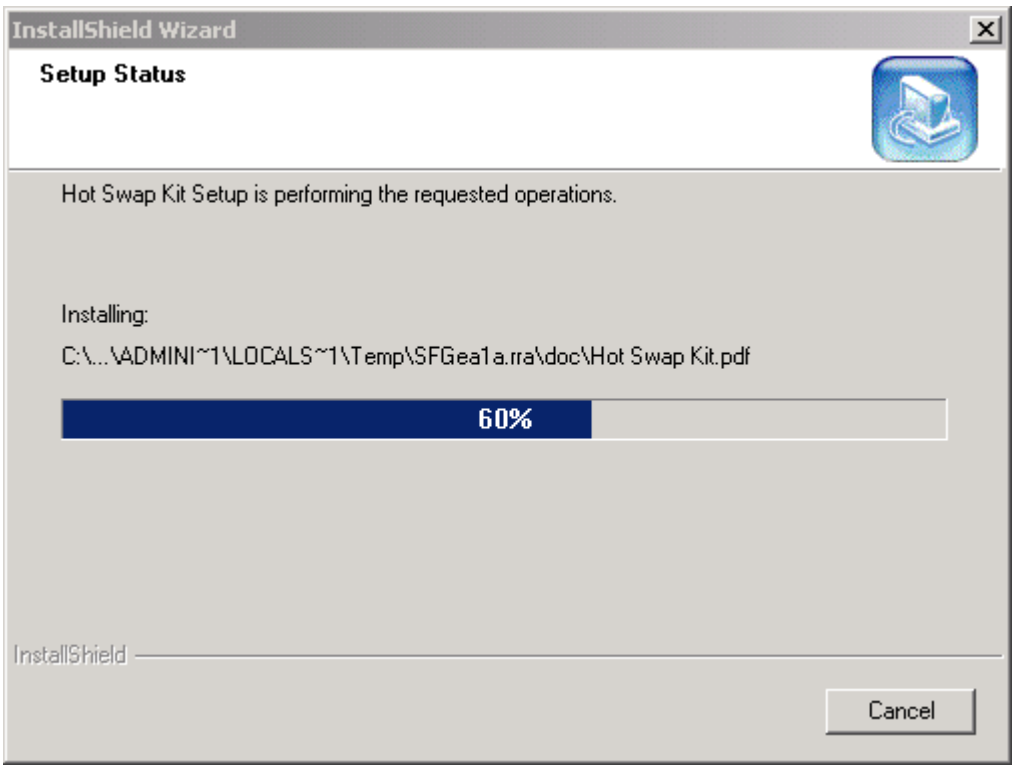

Fig 2.9: Installing the files to their proper location

Files will be copied to their proper location.

If you selected “Other Platform” as the chassis you want supported, you will be asked whether you would like to launch the HSK Wizard or not. The HSK Wizard is described in detail in the next section of the document.

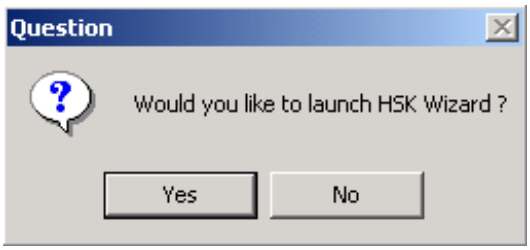

Fig 2.10:HSK Wizard Configuration

Congratulations, you have now successfully installed the Pigeon Point Hot Swap Kit.

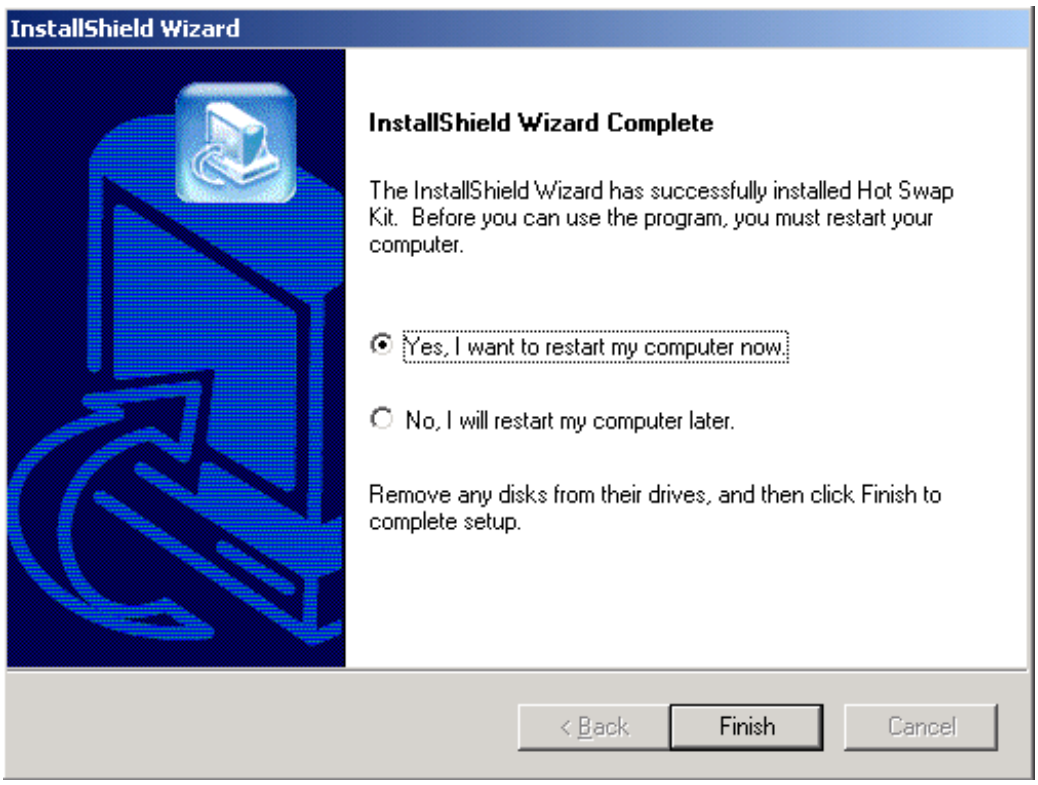

Fig 2.11: Setup Completion dialog box

Press the Finish button to complete the installation. You will need to reboot your system for the HSK to properly function.

2.3 Hot Swap Kit Wizard Utility

The Hot Swap Kit Wizard will step through a process to configure an unknown chassis. You should only need to run this utility if you selected “Other Platform” during the Hot Swap Kit installation process.

The Wizard can be found in the directory “Start Menu\Program Files\Hot Swap Kit”. To configure a chassis, the chassis must be empty except for the system slot processor board(s). At this point begin running the wizard utility.

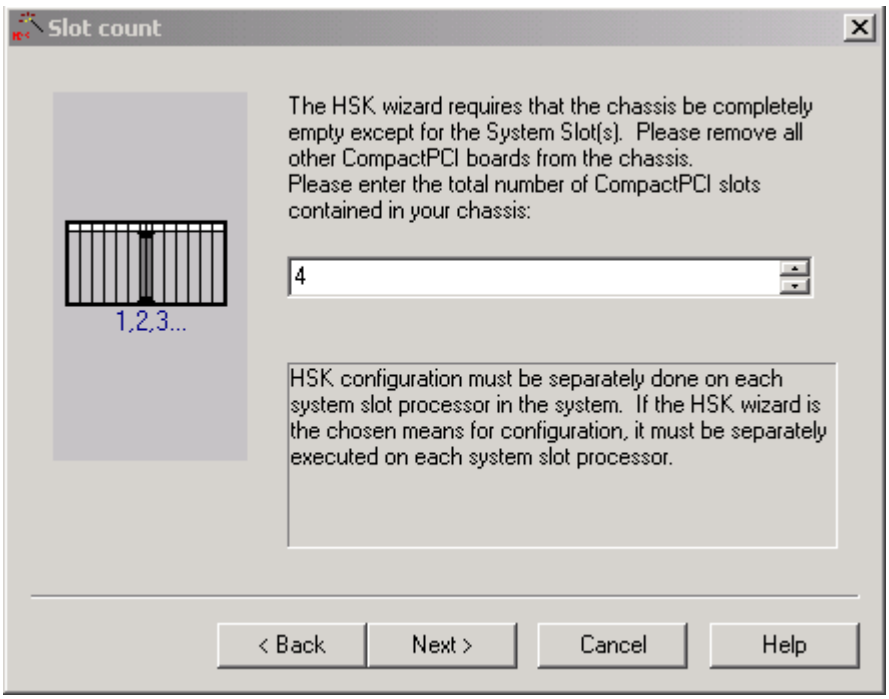

Fig 2.12:Slot Count dialog box

Enter the number of slots contained in the chassis and press “Next >”. You will now be requested to insert and extract a board into each peripheral slot.

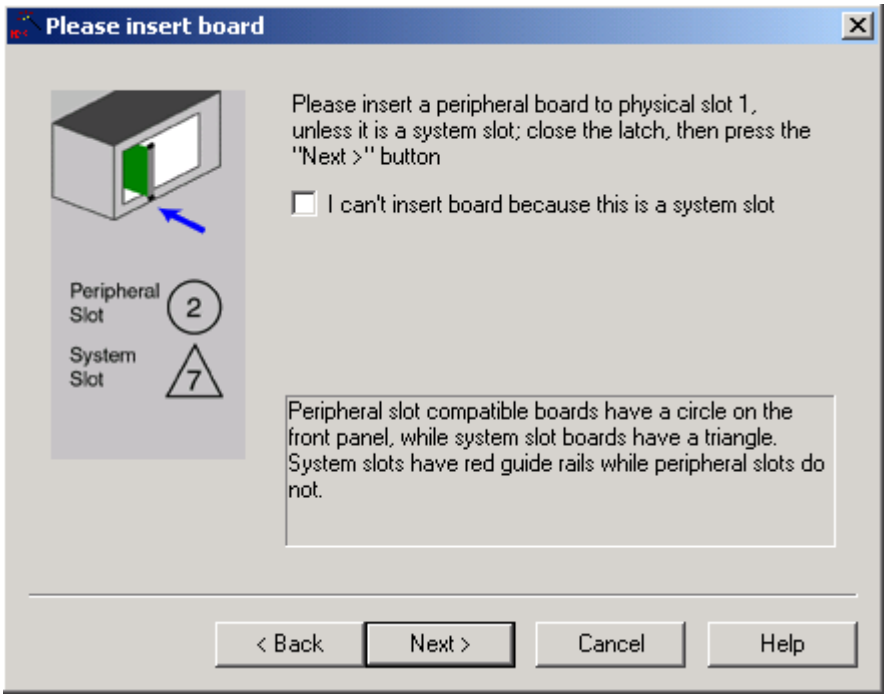

Fig 2.13: Inserting a board

At this point you insert a board into the specified slot. In the event that the designated slot contains a system slot processor board, check the “I can’t insert board because this is a system slot” checkbox. Press “Next >”.

If you inserted a peripheral board, you will then be asked to extract this board.

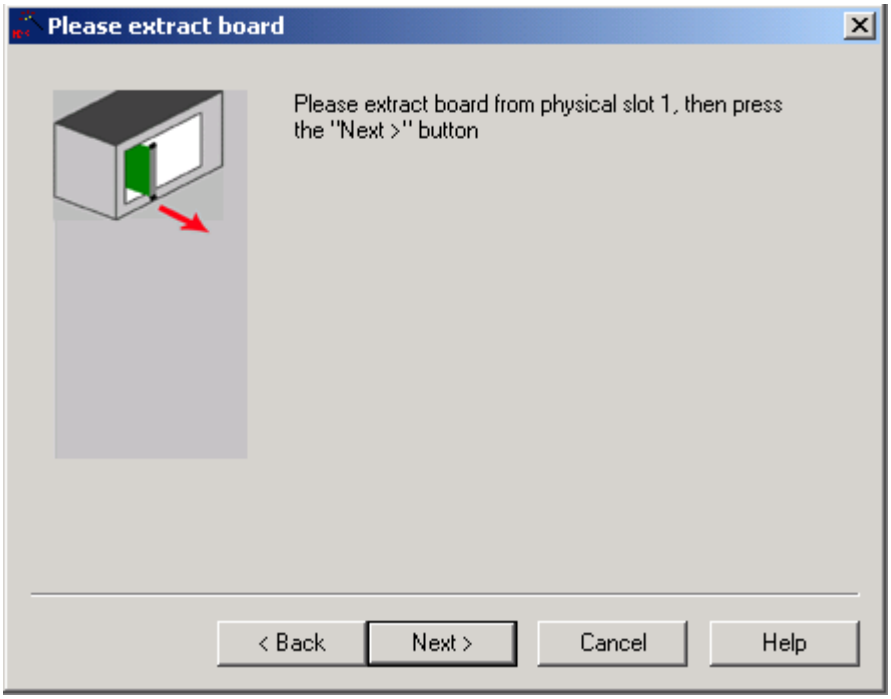

Fig 2.14:Extract board dialog

At this point, you should extract the board normally and press “Next >”. Once all slots have been configured, you will be able to specify resource requirements for parent CompactPCI bridges.
The wizard automatically detects parent bridges and shows the CompactPCI Bridge dialog for each of them. In each dialog, the wizard proposes you the resource window sizes (I/O window, conventional memory window and prefetchable memory window) that HSK will require from the system for that bridge. Also, if the parent bridge is located in a physical slot, the physical slot number can be specified in this dialog as well.

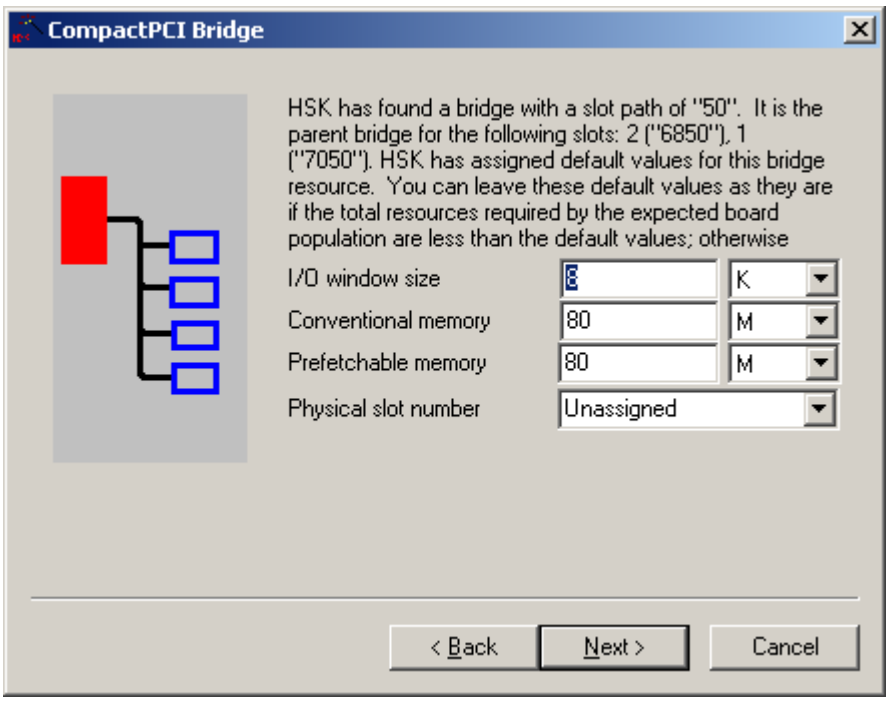

Fig 2.15:CompactPCI bridge dialog.

After setting the bridge parameters, press the “Next>” button. The dialog for the next CompactPCI parent bridge in the system will appear. After you finish setting parameters for the last CompactPCI parent bridge, the HSK Wizard will show you the summary of the configuration.

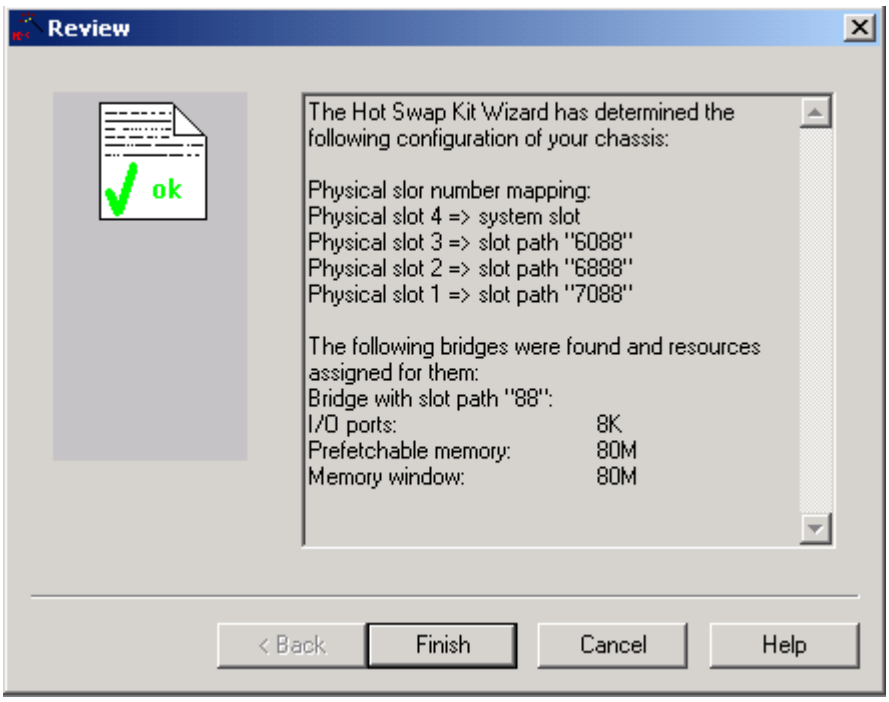

Fig 2.16:Summary Report of a CPX1204 chassis

Assuming that the information presented is correct, press Finish. You will then be asked to import these changes into the registry.

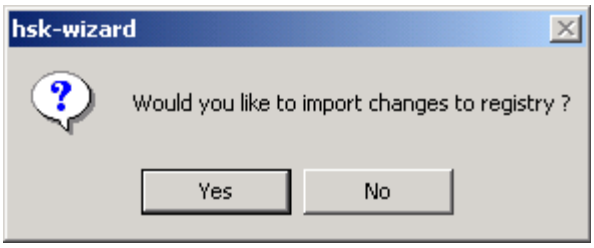

Fig 2.17:Importing information into the registry.

This is the final step. For the chassis information to be fully configured, the information gathered must be put into the registry. Press “Yes” if you are satisfied with the configuration.

2.4 Removing the Pigeon Point Hot Swap Kit

In the event that you would like to uninstall the HSK, you will need to open the Add/ Delete Programs control panel. Find and select the “Pigeon Point Hot Swap Kit” program and press the “Change/Remove” button. Another approach to uninstalling the HSK is to run the original setup.exe installer from the Pigeon Point Hot Swap Kit CD.

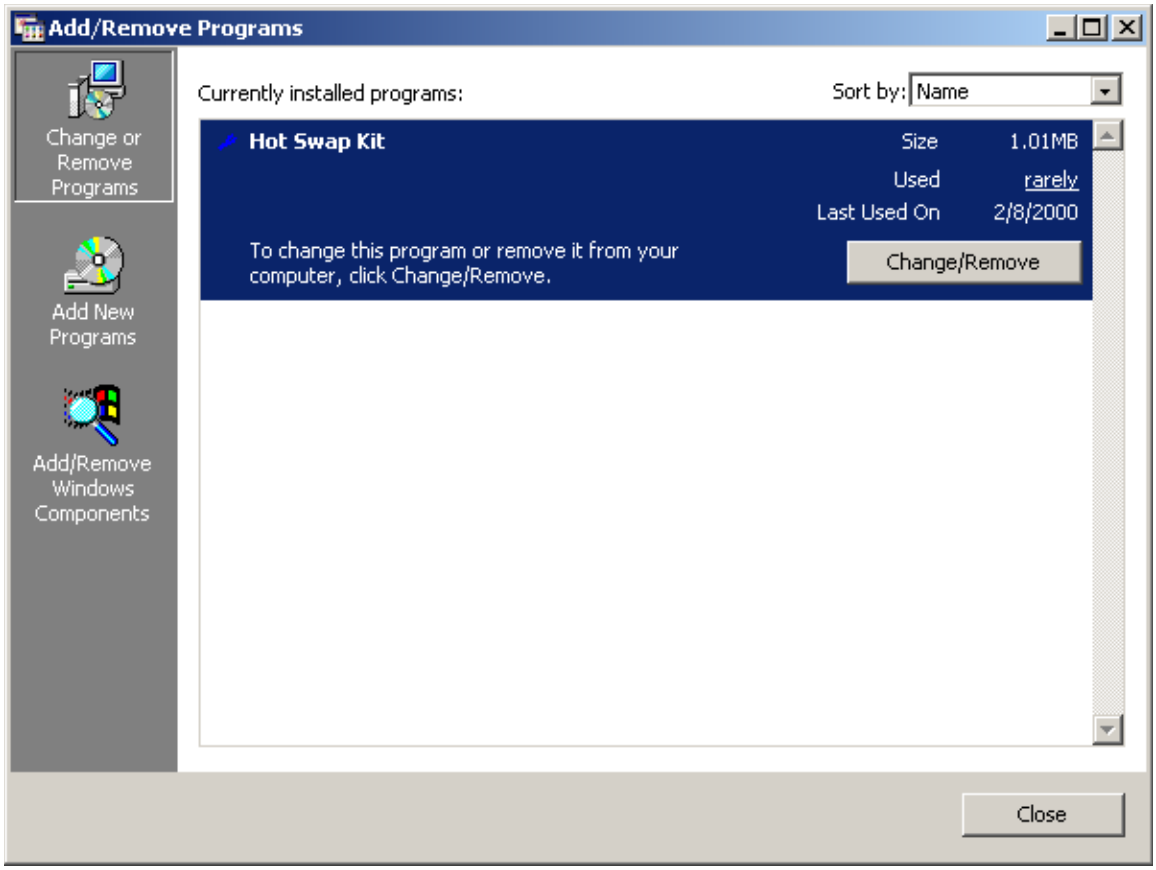

Fig 2.18:Add/Delete Control Panel

The HSK un-installer will begin operation.

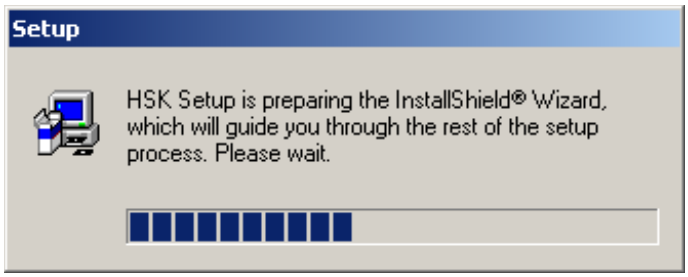

Fig 2.19:Preparing for HSK de-installation

Make sure that the following dialog box appears. If you want to continue to uninstall the HSK, press OK.

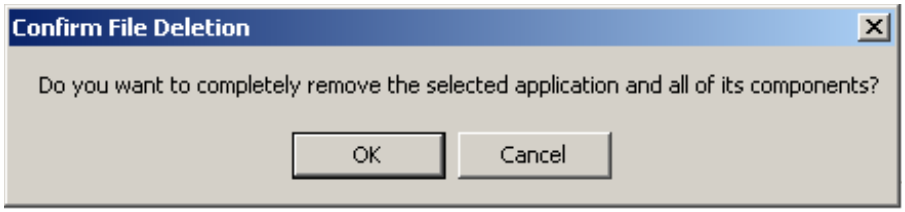

Fig 2.20:Final check for un-installing

If NetStructure HA SDK software is installed yet then the following message appears. If you want to remove NetStructure HA SDK software immediately, press YES.

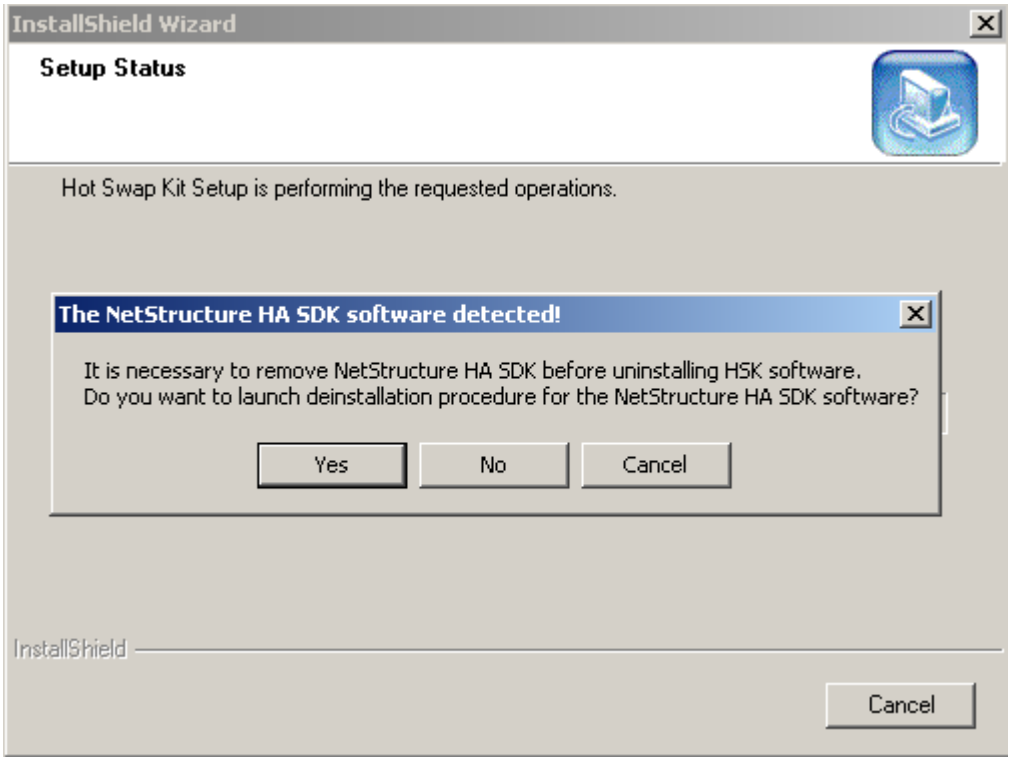

Fig 2.21:Setup detected installed NetStructure HA SDK software

The HSK files will begin the process of being un-installed.

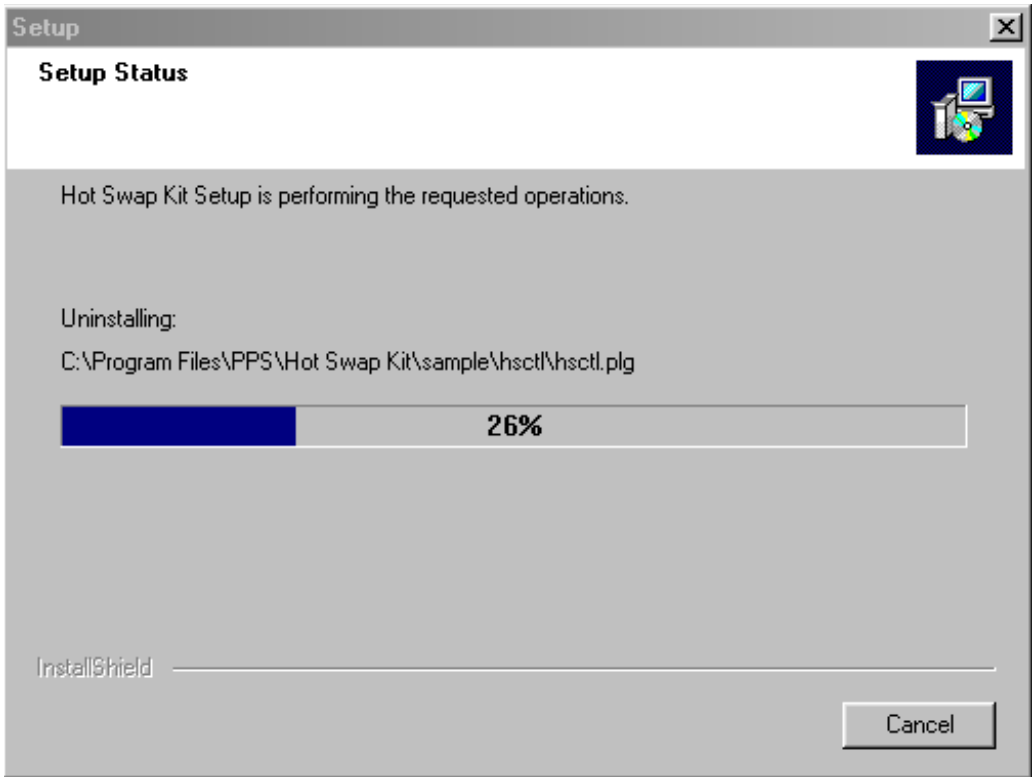

Fig 2.22:Setup status for removing the HSK files

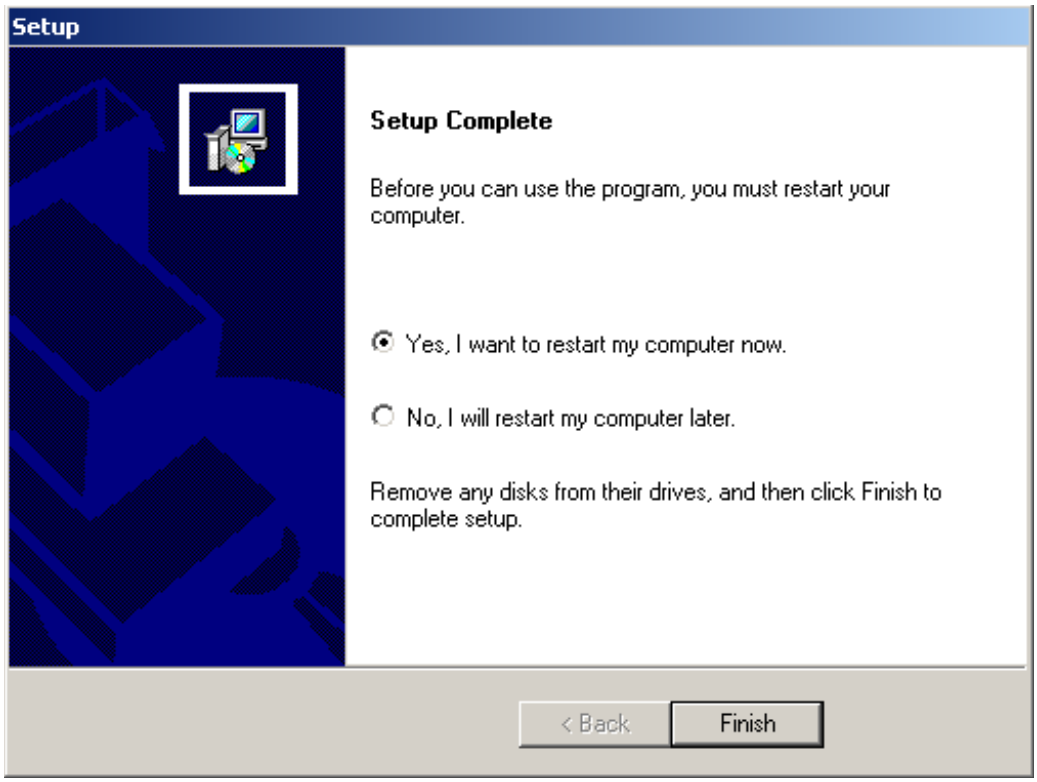

Fig 2.23:Final completion dialog

The Pigeon Point Hot Swap Kit has now been successfully removed. We suggest that you reboot the system at this point or risk potential future instability.

2.5 Validating Successful Installation

Once you have properly installed the HSK, you can validate that everything is working properly. Use the Slot Information utility from the Start Menu. If the HSK is properly installed, this program will show all of the boards found in the system. In the event that the HSK is not properly installed, no information will be provided and a dialog (as pictured below) will be shown. Please refer to the Chapter entitled “Hot Swap Kit Support Utilities” for additional information on the use of this utility and other utilities provided with the HSK.

Fig 2.24:Failure in SlotInfo as a result of HSK not being properly installed.

2.6 Installing the Pigeon Point Redundant Host Software

Note! ZT5524 on ZT5085 and ZT5550 on ZT5084 hardware configurations are supported Insert the Pigeon Point Redundant Host software CD into your system. Note! Redundant Host support is the part of the NetStructure HA SDK that brings the RHOST support. Your system should be running Microsoft Windows 2000 SP4. In case of ZT5524 OS should have the MPS Multiprocessor HAL. To begin installation, run the rhswsetup.exe program off of the CD. The NetStructure HA SDK setup screen below will be shown.

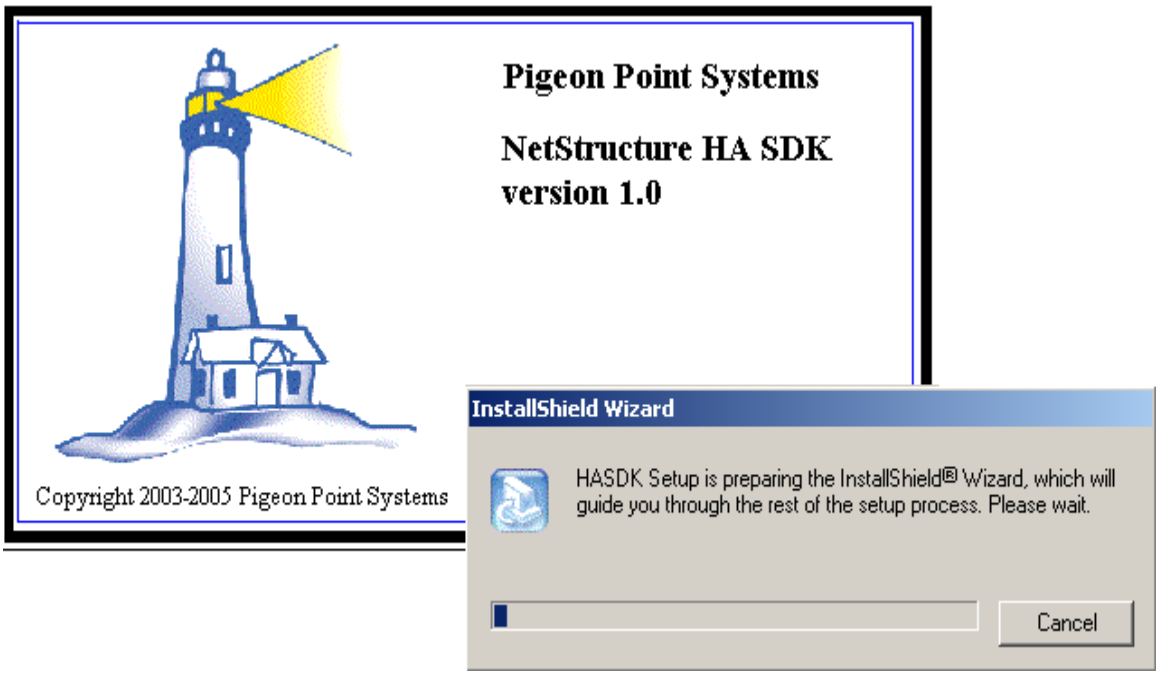

Fig 2.25:Initial Setup preparation screen

After the setup has loaded all of the necessary files, the Installation welcome screen will be shown.

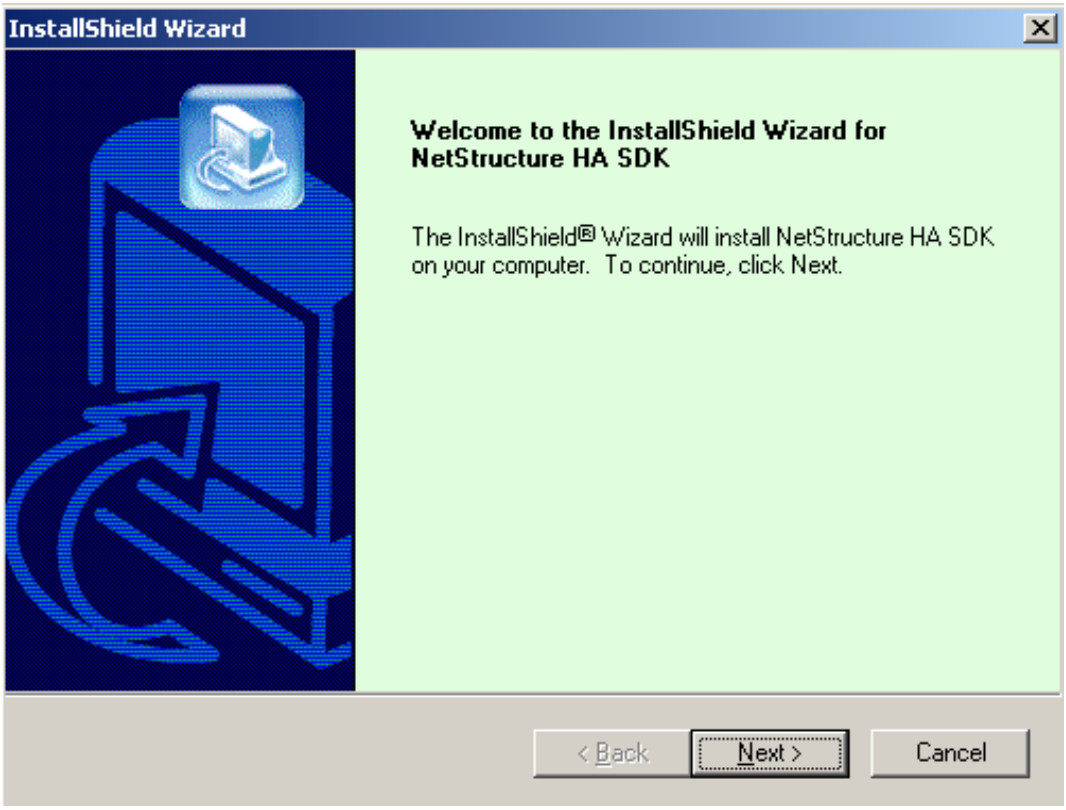

Fig 2.26:Installation Welcome Screen

Click Next to proceed with the installation.

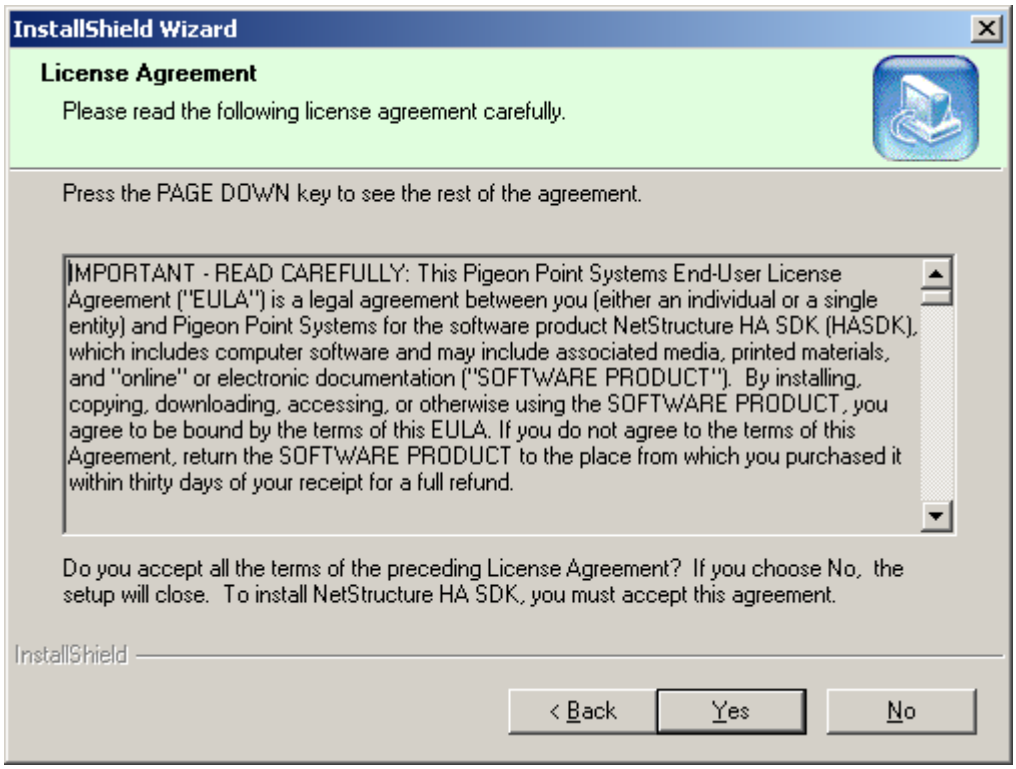

Fig 2.27:License Agreement

Please read the License Agreement fully before giving your consent. If you do not agree with the license agreement, please return all Pigeon Point NetStructure HA SDK product packaging, without installing the software, back to the place of purchase. If you agree with the license agreement, select the “Yes” button.

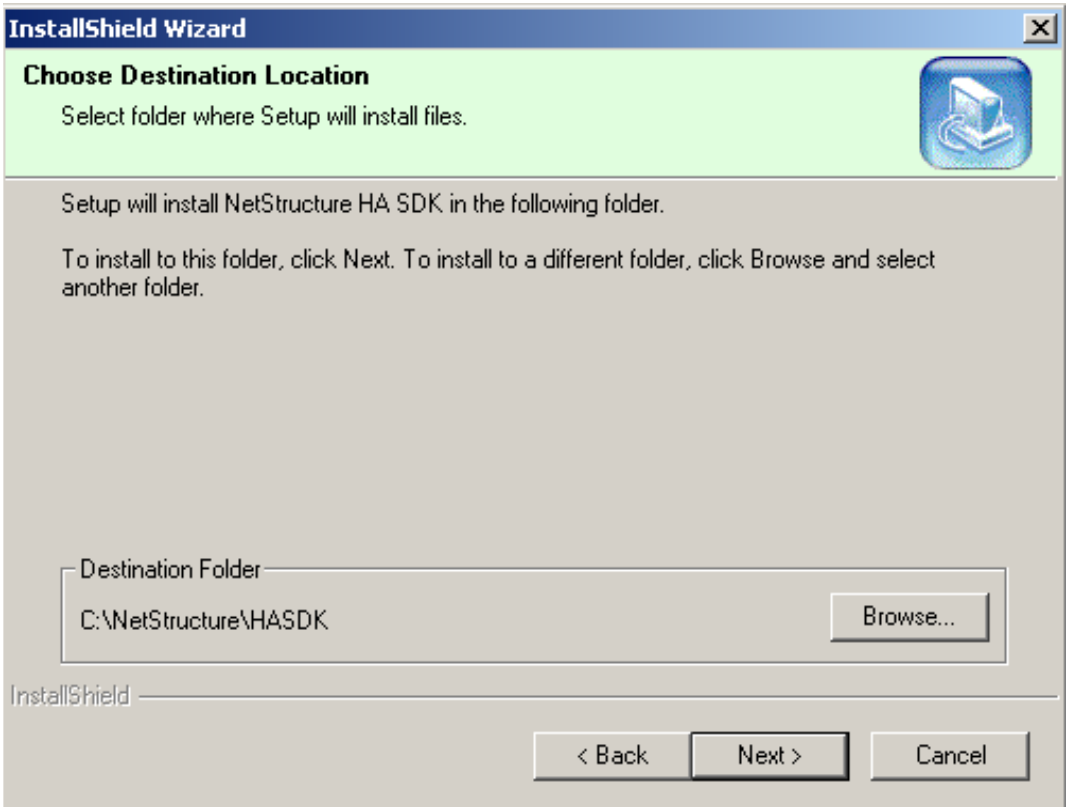

Fig 2.28:Choosing the Destination Folder

The default target directory for the installation is “C:\NetStructure\HASDK”. If you would prefer to install the NetStructure HA SDK in another location, you may browse through your system to select another location.

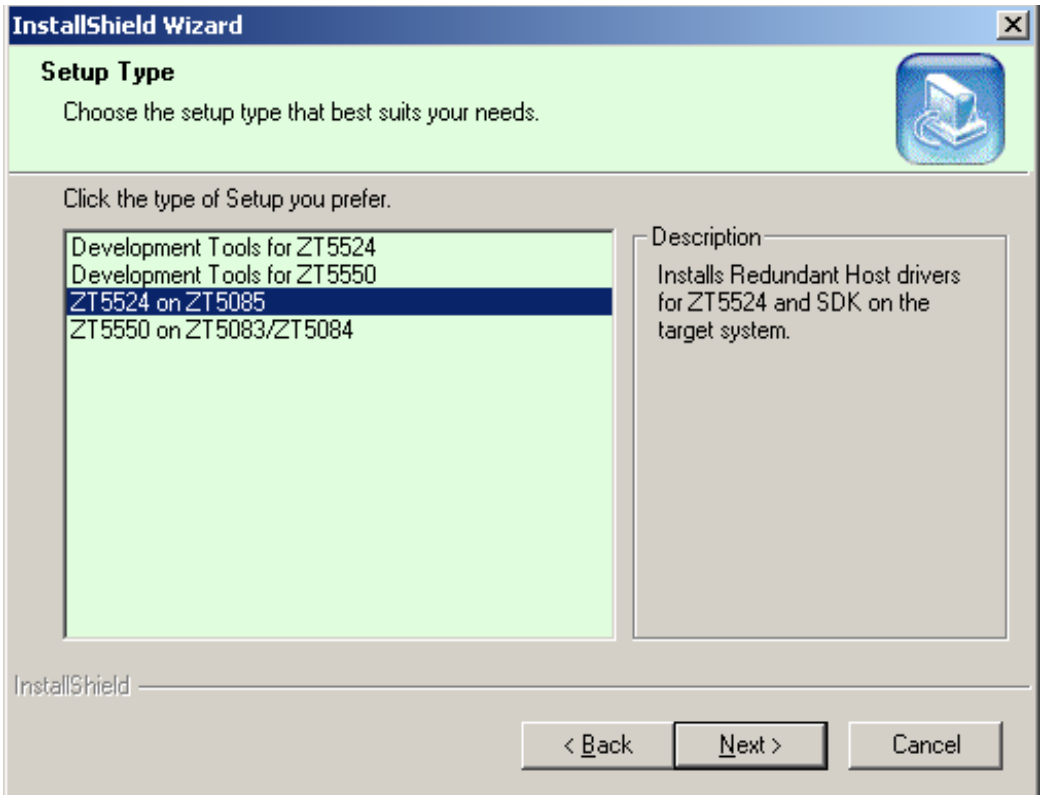

Fig 2.29:Choosing the system that you want supported.

Please select the system that you want supported. If you are developing applications on top of the NetStructure HA SDK API and do not have a system connected, select “Development Tools” that you want to supported. Once you have selected, press the “Next>” button.

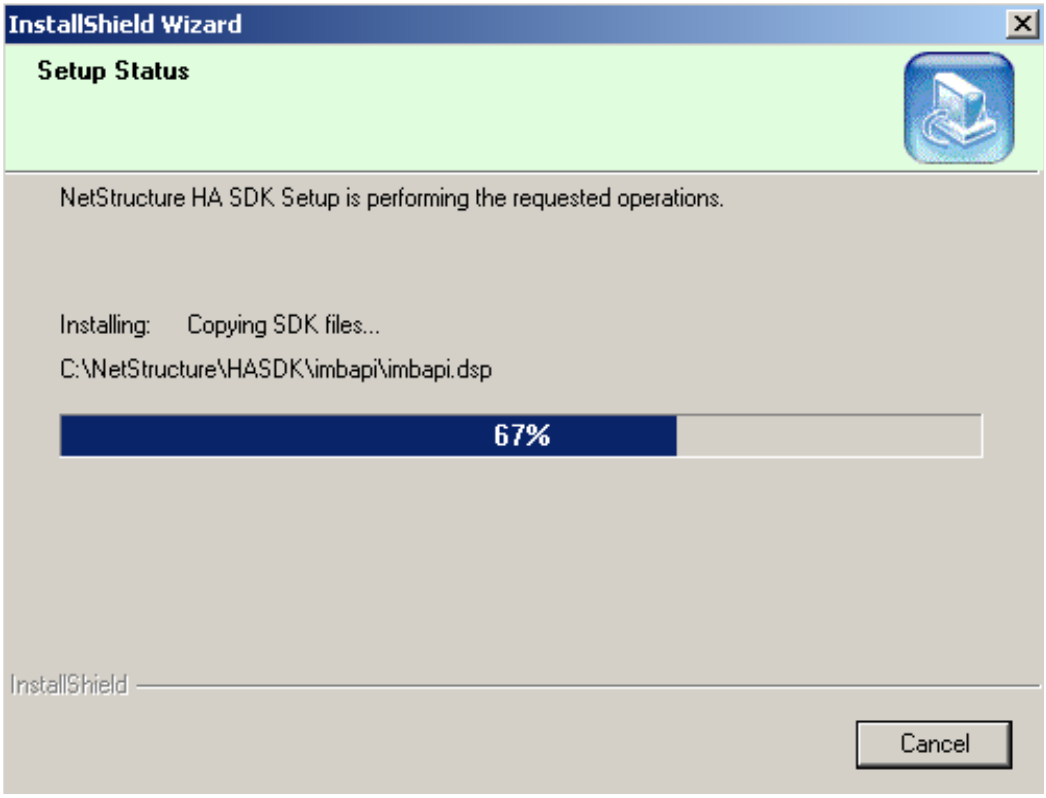

Fig 2.30: Installing the files to their proper location

Files will be copied to their proper location.
Congratulations, you have now successfully installed the Pigeon Point NetStructure HA SDK software.

If HSK software was not installed yet then the following message appears. You should install HSK first.

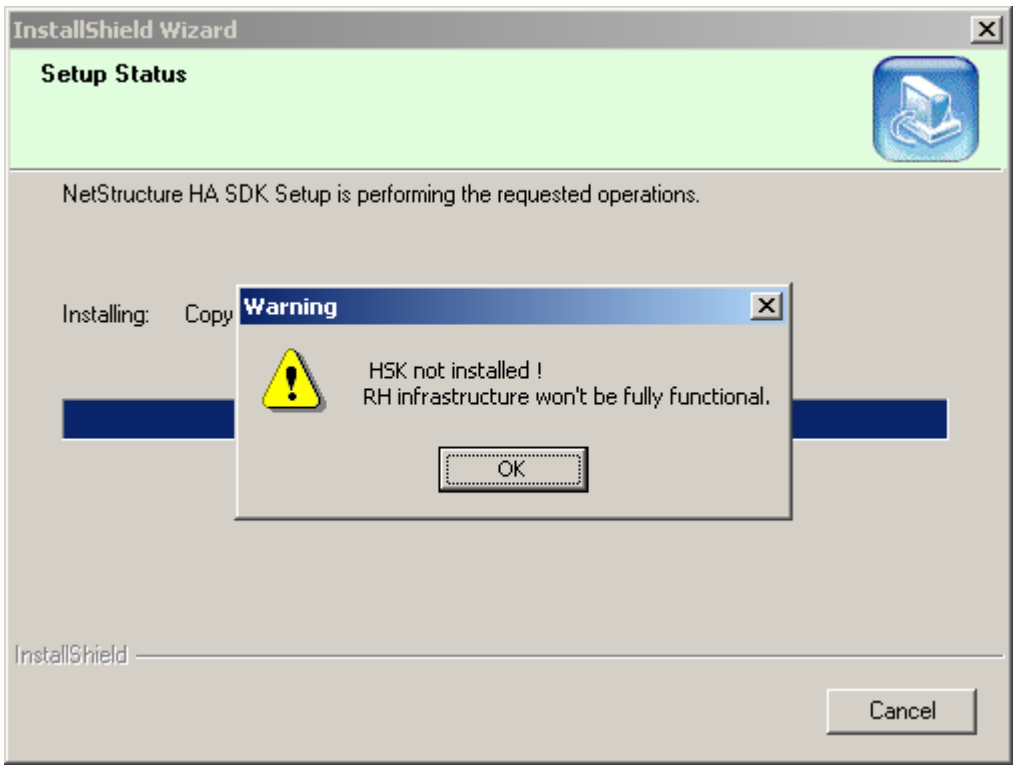

Fig 2.31: Setup detected that the HSK was not installed yet

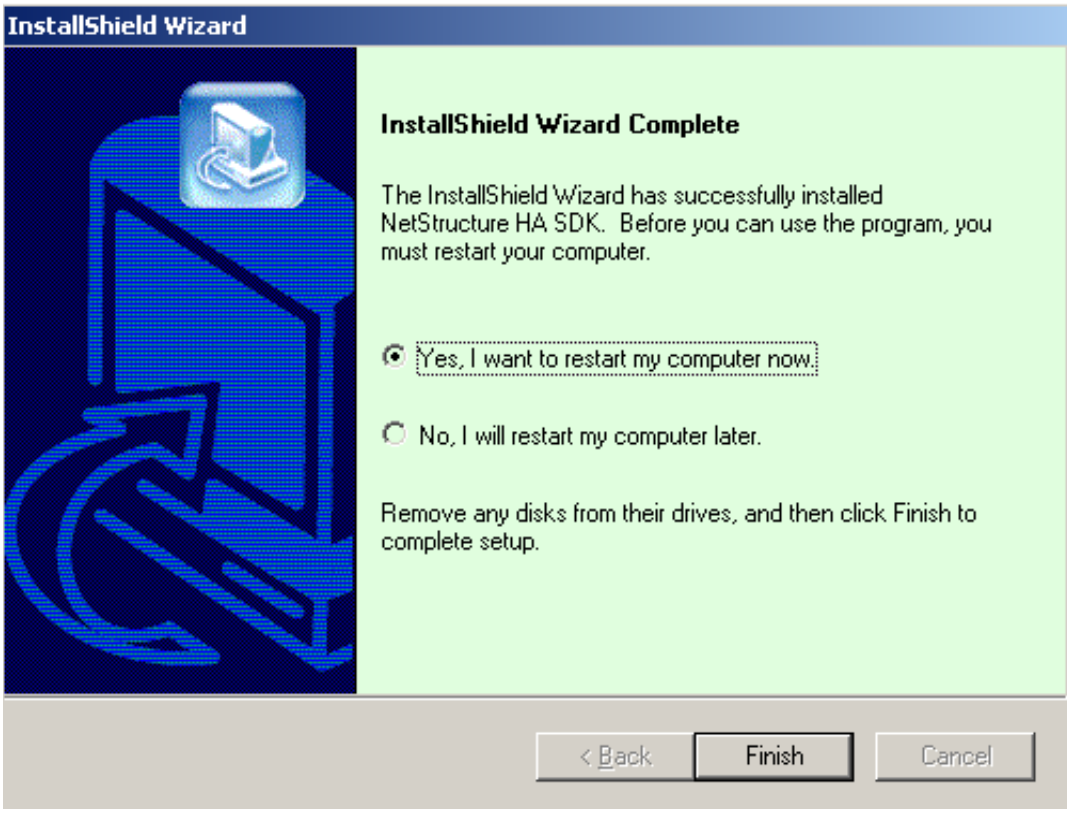

Fig 2.32: Setup Completion dialog box

Press the Finish button to complete the installation. You will need to reboot your system for the NetStructure HA SDK to properly function

2.7 Removing the Pigeon Point Redundant Host Software

In the event that you would like to uninstall the NetStructure HA SDK, you will need to open the Add/Delete Programs control panel. Find and select the “NetStructure HA SDK” program and press the “Change/Remove” button. Another approach to uninstalling the NetStructure HA SDK is to run the original rhswsetup.exe installer from the Pigeon Point NetStructure HA SDK CD.

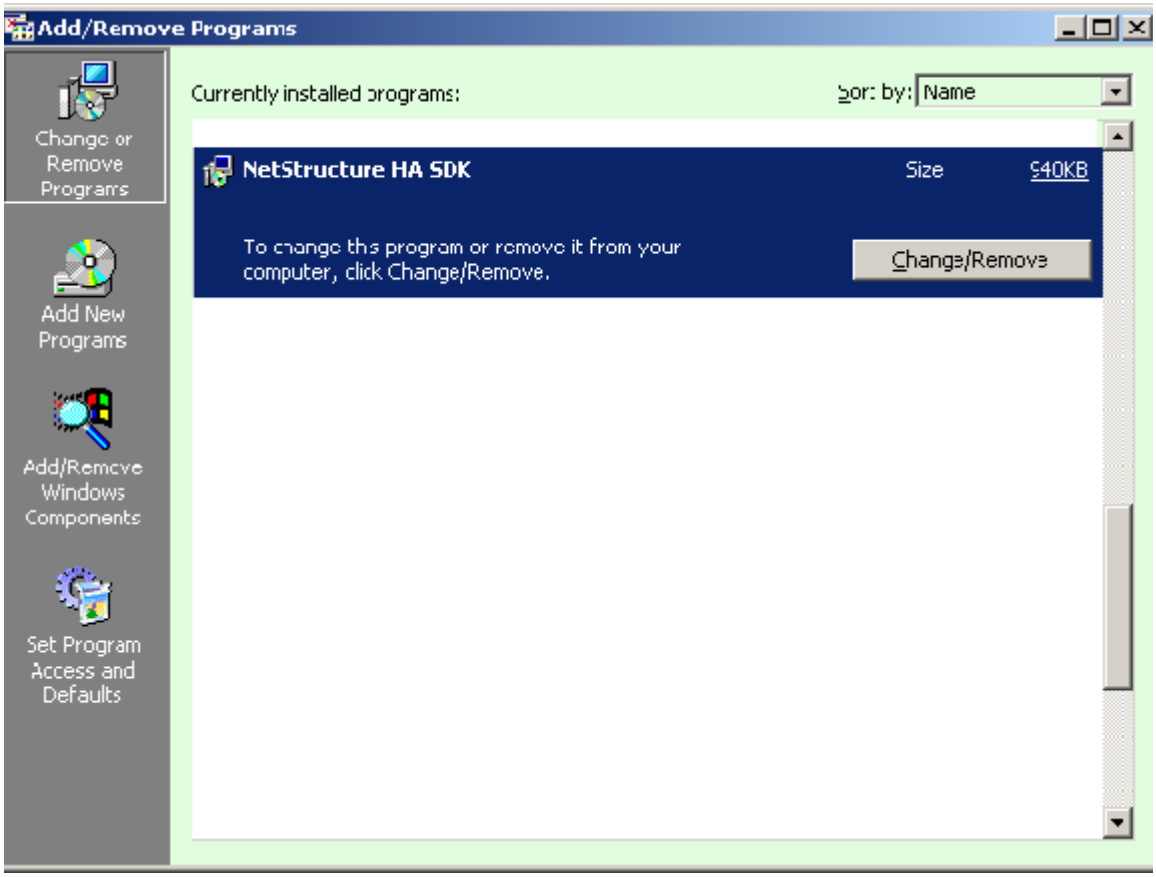

Fig 2.33:Add/Delete Control Panel

The NetStructure HA SDK un-installer will begin operation.

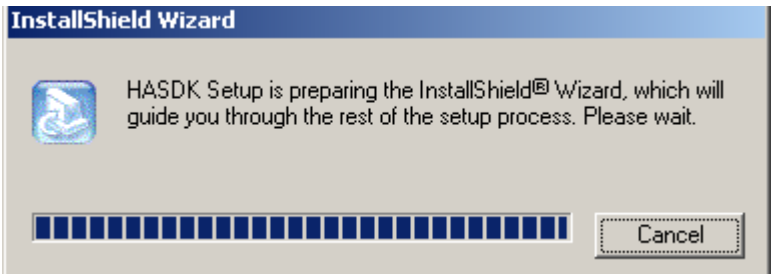

Fig 2.34:Preparing for NetStructure HA SDK de-installation

Make sure that the following dialog box appears. If you want to continue to uninstall the NetStructure HA SDK, press OK.

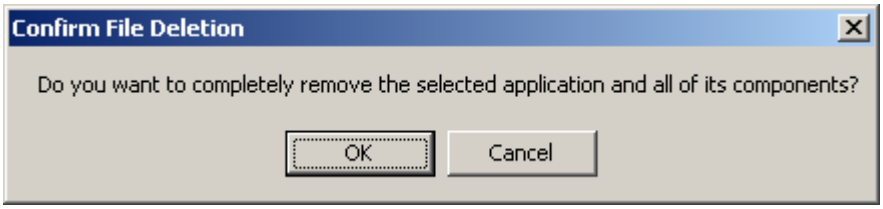

Fig 2.35:Final check for un-installing

The NetStructure HA SDK files will begin the process of being un-installed.

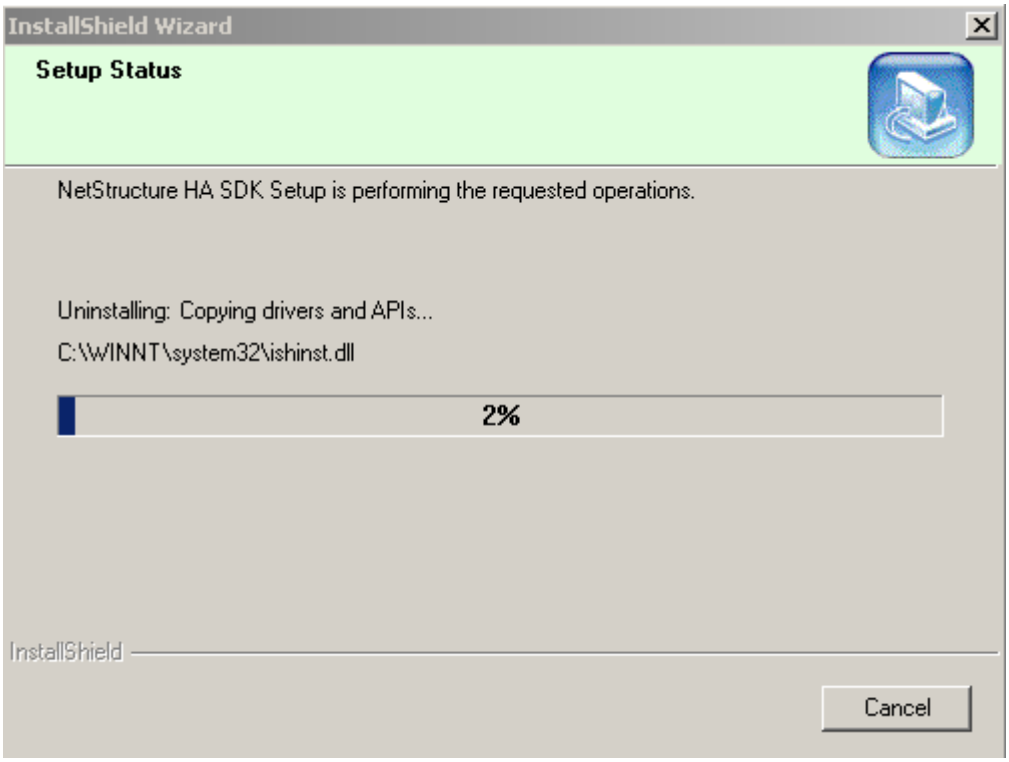

Fig 2.36:Setup status for removing the NetStructure HA SDK files

Fig 2.37:Final completion dialog

The Pigeon Point NetStructure HA SDK has now been successfully removed. We suggest that you reboot the system at this point or risk potential future instability.

3. Working with the Pigeon Point Hot Swap Kit

3.1 Overview

The Pigeon Point Hot Swap Kit was designed to allow hot insertion and extraction of CompactPCI boards while the host Windows 2000/XP operating system continues normal operation. CompactPCI boards fall into two categories: (1) System slot processor (or "host") boards; and (2) peripheral I/O boards for *non*-system slots. We highly recommend that you review the relevant hardware installation and user guide for instruction on how to properly insert and extract your CompactPCI boards.

On some specially designed systems, it is possible to hot swap the system slot processor boards. *Never attempt to hot swap those boards on other systems*. HSK does not support swapping of system slot processor boards. Pigeon Point Hot Swap Controller Kit provides system slot processor swapping for CPX8216T. Please contact Pigeon Point Systems for more information.

3.2 Inserting a New Board

Insert a board firmly into an empty CompactPCI slot and close both handles[1]. After the insertion, the blue hot swap LED on the board should be off. The hot swap infrastructure will automatically recognize the new board and initiate the software connection process for it.

[1]In this release of the HSK documentation, we assume 6U-sized CompactPCI boards. 3U-sized boards have only one handle.

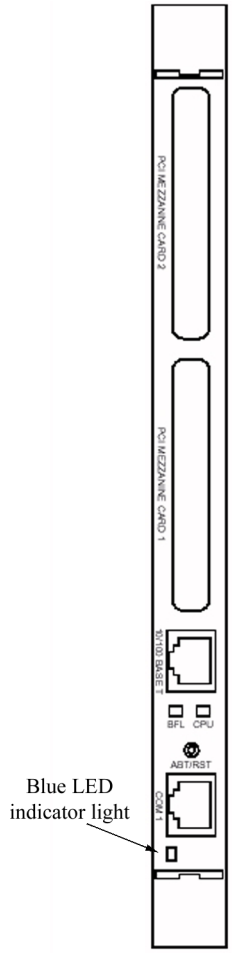

Fig 3.1: Front Panel for Motorola MCPN750 non system slot processor module.

Once Windows 2000/XP has determined that a new board has been inserted, it will attempt to find the drivers.

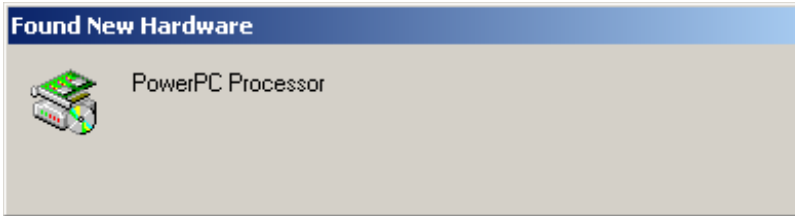

Fig 3.2: Found new hardware dialog after inserting MCPN750

Upon inserting the board into a slot, the operating system will look for drivers that can be installed for this new device. If drivers have been previously installed, no user interaction will be required. If the operating system cannot find any drivers for this device, the Hardware Installation Wizard will be started.

Fig 3.3: Hardware Wizard installation screen

You will then be prompted to find the appropriate drivers. You should consult the user manual for the inserted device for any driver installation instructions.

3.3 Extracting a Board

To extract a board from a slot, unlock the board handle. Windows 2000/XP will recognize that you have begun the process of extracting a board and will begin to prepare the operating system for the board's removal. Once it is safe to remove the board from the system, the blue LED will be illuminated (see Figure 3.1).
Note: for some boards, the light may blink during the software disconnection process; only when the light is steadily on for a couple of seconds is it safe to remove the board. If the board is currently being accessed or used, the software disconnection request may be denied by the board's device driver, other device drivers, or by a user-mode application. In this case, the blue LED light will not be illuminated. The hot swap infrastructure may repeat software disconnection attempts periodically while the board handle is down. Refer to the Chapter entitled "Hot Swap Kit Support Utilities" section "Modifying Software Disconnection Timeout" to query or change this setting. If one of subsequent disconnection attempts finally succeeds, the blue LED will be illuminated.

The board can be successfully extracted after the blue LED comes on. The hot swap infrastructure will detect that the board has disappeared and will notify the operating system.

To cancel extraction, lock the board handle without removing the board. The blue LED will turn off and the software connection process will begin.

In Windows 2000, requests to the operating system to eject a device (posted via IoRequestDeviceEject) are handled by the system asynchronously, after the call to IoRequestDeviceEject returns. Hence, the following sequence of events is possible:

1) The extraction request is issued to the board (handle is unlocked) and the request to eject the device is posted to the system via IoRequestDeviceEject;
2) The extraction request is cancelled (handle is locked) *before* the system processes the pending eject request;
3) The ejection request (EJECT IRP) arrives for the device though the handle is already locked.

At stage 3, HSK cannot normally distinguish this stale ejection request from an ejection request issued to the system by non-HSK means (e.g. via the configuration management API). To complicate matters, the system may just drop the IoRequestDeviceEject request and not issue the EJECT IRP for the device without any indications (if, for example, the ejection request is vetoed by a driver or an application). Hence, HSK will illuminate the blue light and pend the ejection request awaiting physical removal of the board; the board won't be functional and will be awaiting physical removal though the board handle is back up. To make the board functional again in this state, the operator should unlock the handle down and then lock the handle, or remove the board and reinsert it.

To prevent this undesirable behavior from happening, a special interval timer has been added to the Hot Swap System driver, and the timer duration in milliseconds is configurable via the Hot Swap Configuration Utility (see Chapter 4) and also directly as a registry value.

Path: HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Hspci\Parameters\InsertBeforeEjectInterval
Type: REG_DWORD
Synopsis: This value specifies an interval timer that is activated on a per-device basis when an insertion request takes place for the device and a pending extraction request exists for that device (that is, IoRequestDeviceEject has been issued before). If the Eject IRP arrives for this device when this timer is active, it is considered to be a stale Eject IRP from the previous ejection request, and is rejected. Thus, if the time difference between the insertion request (stage 2) and the EJECT IRP arrival (stage 3 above) is no more than InsertBeforeEjectInterval milliseconds, the stale EJECT IRP will be discarded, the device will be re-activated and its state will be consistent with the position of the handle.
Default Value: 5000 (5 seconds)

3.4 "Surprise" Extraction

A "surprise" extraction can occur if you extract a board without waiting for the blue light. If this occurs, the operating system will be notified by the hot swap infrastructure about the sudden disappearance of the device. The device driver will immediately be notified about the surprise removal of the device.

Surprise extraction is unsafe and may bring the system down if the device driver tries to access the hardware before getting the surprise removal notification. The operating system warns you about this by issuing a message to the system console in the case of surprise removal.

3.5 System Slot Processor Extraction

HSK does not support system slot processor extraction. Although the CPX8216 has this capability, Pigeon Point Systems Hot Swap Controller Kit is necessary for this functionality. If you attempt to unlock the system board on a CPX8216, the system will continue to operate without any change. If you attempt to remove a system slot processor board on any platform, this may have disastrous affects on your system.

3.6 Redundant Host operations

Slot Processor extraction / insertion are the general operations if Redundant Host software (which is part of the NetStructure HA SDK) was installed. Intel® High Availability (HA) systems feature built-in redundancy for active system components such as power supplies, system master processor boards, and system alarms. Redundant Host (RH) systems are HA systems that feature an architecture allowing the Active Host system master processor board to hand over control of its bus segment to a Standby Host system master processor board and vice versa. The following "High-Availability CPU Architecture" figure shows how the basic elements in an High Availability system are related.

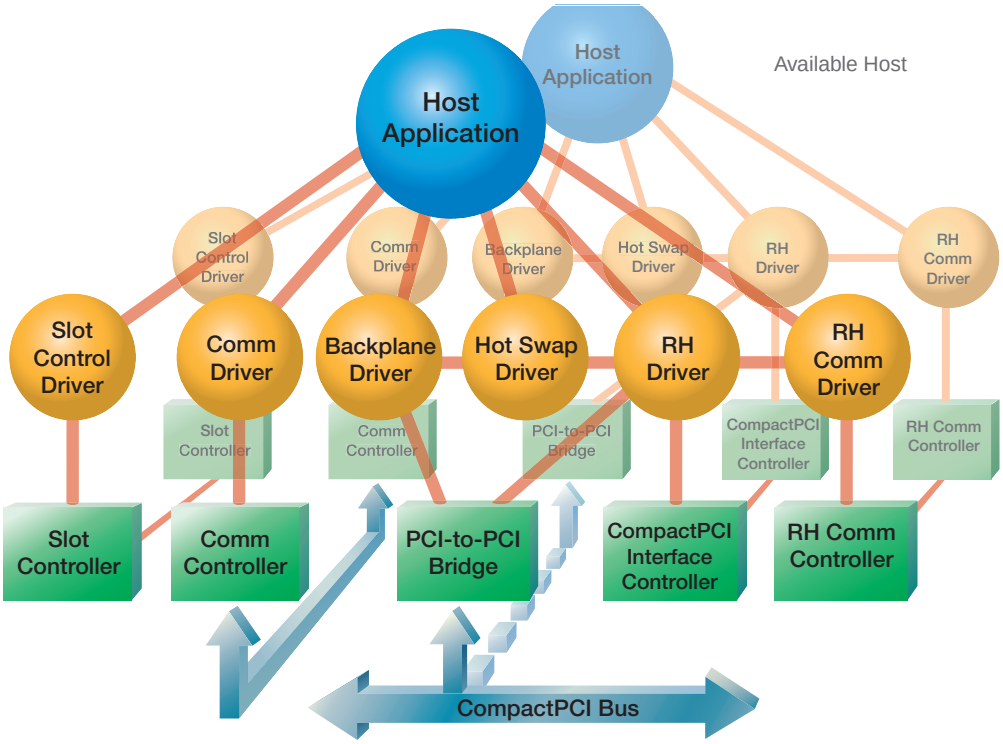

Fig 3.4: High-Availability CPU Architecture

3.6.1 High Availability Hardware Approach

In an RH system is spread across several building blocks. These include:

- Processor boards (such as the Intel® NetStructure™ ZT 5524 or ZT5550 System Master Processor Board)
- Bridge mezzanine (such as the Intel® NetStructure™ ZT 4901 Mezzanine Expansion Card)
- Backplane (such as the Intel® NetStructure™ ZT 4103 Redundant Host Backplane)

Other building blocks and subsystems may be required to support the RSS subsystem. These include:

- System management
- Storage
- Power distribution
- Cooling
- Media
- Packet switching

Intel's RH software runs on system master processor boards with bridge mezzanine cards in a PICMG 2.13 compliant RSS backplane to provide redundant system master functionality. This allows the failover of control of redundant PCI buses. It provides faster hardware that is PICMG 2.9 and 2.16 compliant. The system makes use of the IPMI infrastructure for fault detection and correction.

Processor Boards

The Host processor board is a CompactPCI system master processor board, such as the ZT 5524, ZT5550, that can operate in Owner Mode or Drone Mode, and may operate in Peripheral Mode. Additionally, it must be able to gracefully transition between modes by coordinating with a Redundant Host (RH). The processor board must also support hot swap when it is in Drone Mode.

Bridge Mezzanine

The HA driver set works in single and dual bus segment configurations. In order for the dual bus configuration to be supported a bridge mezzanine must be mounted on the processor board.

The bridge mezzanine is a board that is physically attached to the base processor board. The processor board and bridge mezzanine are stacked such that they occupy two adjacent CompactPCI slots.

Like the base processor board, the bridge mezzanine has a CompactPCI bus segment interface that can operate in Owner Mode or Drone Mode. The bus interface mode of the bridge mezzanine is independent of the processor board's mode.

The bridge mezzanine contains elements that are identical to the base processor board in order to create a second CompactPCI interface for connection to a different bus segment, as shown below.

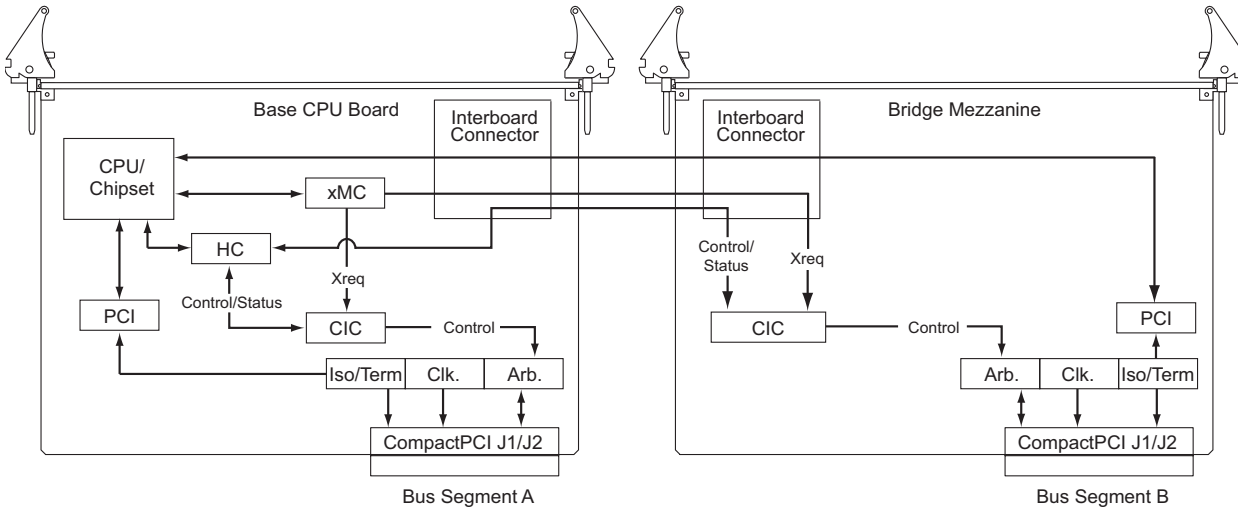

Fig 3.5: RSS Host with Bridge Mezzanine Block Diagram

Backplane

The RH system backplane supports two CompactPCI buses accessible by both Redundant Hosts. In Active-Standby mode, the active processor board controls the buses (Active Host) and the standby processor board is isolated from the backplane (Drone mode). The backplane has separate buses for active-to-standby processor board communication (COMM) and Host Controller functions. See the "High-Availability System Backplane Architecture" figure below for an example of a typical High-Availability backplane.

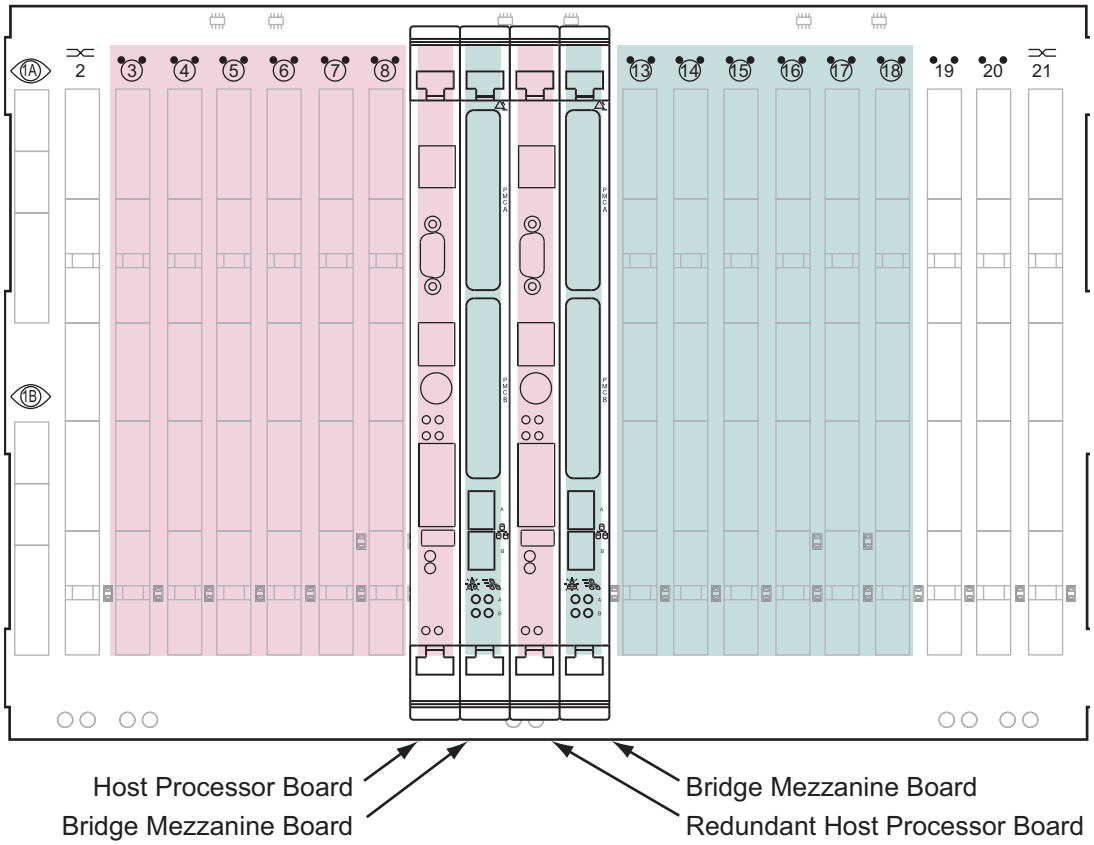

Fig 3.6: High-Availability System Backplane Architecture

3.6.2 High-Availability Software Approach

As shown in the "High-Availability CPU Architecture" figure, there are three High-Availability software components are intended:

- Host application
- System Management
- Backplane Device Drivers

Host Application

The host application serves as the central control mechanism for the platform. For a host application to function in an RH environment it must be able to relinquish or receive control of the system in a controlled manner. Dynamically transitioning of bus segment ownership between active and backup requires the application to maintain data synchronization between the applications on the redundant Hosts.

The design of the application should be made as portable as possible. This requires that the design be implemented in a modular approach that isolates the system management requirements from the host application. This division of responsibilities can be achieved through a layered implementation.

In addition to taking a modular approach, the application designer should recognize the importance of producing a hardened application. A hardened application must at least provide a capable logging mechanism that allows for application faults to be reconstructed and corrected. It should also adhere to good coding practices such as validating all input parameters and return statuses. A more proactive approach is to implement fault recovery mechanisms. This could include the capturing of faults and the isolation of faulted application components.

Host application is not he component of the Redundant Host software. It is the additional product that is build atop of the Redundant Host and Hot Swap layers.

System Management

System management is the mechanism by which system configuration and fault characteristics are established, insuring system health is maintained. In the Redundant Host architecture there are extensive sets of APIs that provide the developer with a fine level of control of the platform.

In order to manage such a configuration, a number of function calls are required so that predetermined default actions can be prescribed depending on the desired switchover strategy. The required functions are based on the *Hot Swap Infrastructure Interface Specification, PICMG 2.12*, specifically in the Redundant Host API chapter. The supplied APIs provide the following abilities:

- Enumerate the hosts, domains, and slots in the system
- Get information about devices in slots
- Initiate domain switchovers among hosts
- Enable and disable notifications regarding switchover operations
- Specify actions that result from hardware-initiated alarms and control notifications about alarms.

ZT5524 management is achieved using the IPMI infrastructure. The IPMI interface exposes the embedded monitoring devices such as temperature and voltage sensors. There currently is no industry standard API for managing IPMI devices, primarily because the devices that are used may vary significantly between chassis configurations. Since the drivers supplied for use in the Redundant Host architecture are operating system dependant, the interfaces used to access the IPMI devices are not necessarily portable between the supported operating systems.

The supplied Hot Swap API provides a mechanism to identify the topology and Hot Swap state within a specified chassis. By using this API the system management application is able to identify which slots are populated and the power states of the occupying boards. There are additional APIs that allow for simulated backplane peripheral insertion and extraction. In addition, this API provides for notification of Hot Swap events.

The Slot Control Interface is independent of the Redundant Host driver. This separation of functionality is designed to allow for slot control functionality in a chassis without full hot swap or redundant host capabilities. The Slot Control API is based on the PICMG 2.12 High Availability Slot Control Interface functions. It interacts with the Slot Control Driver to create IPMI messages through which a finer granularity of board control can be

achieved then was found in previous generations of High Availability systems. Using the Slot Control API the application can retrieve information regarding “Board Present Detection”, “Board Healthy”, and “Board Reset” capability.

Backplane Device Drivers

Backplane device drivers are a critical component of High Availability system. The drivers need to be robust in their operations as well as to be dynamic given the “Stated” nature of a Hot Swap architecture.

The ability of a driver to remain loaded and initialized even though the Host may not have visibility to the device is critical when Host ownership transfer can occur almost instantaneously. This means that the driver must be able to be started and stopped asynchronously.

4. Using Hot Swap Kit Support Utilities

Included with the HSK are several utilities. These utilities provide additional functionality so that you can better use your CompactPCI system. Before providing details on the HSK utilities, we must first introduce some general concepts. For more details about these general concepts, please refer to Chapter 5.

4.1 General Concepts

4.1.1 Slot Paths

Slot path is the means of slot identification by Windows 2000/XP that does not depend on possible bus number changes. Each slot is identified by the sequence of PCI device/function numbers, starting from the device itself, and going through the chain of bridges up the PCI tree to bus 0. Each number in the sequence represents the PCI device number, multiplied by 8, plus the PCI function number. PCI function number is usually 0.

4.1.2 Physical Slot Numbers

The HS infrastructure supports physical slot numbers, treated as arbitrary non-zero 32-bit unsigned numbers (but recommended to correspond to the physical slot numbers defined in the CompactPCI specification and established by the platform and intended for communication with the user). Mapping between physical slot numbers and slot paths is assumed to be known a priori to the infrastructure and can be modified using the Hot Swap Kit Configuration Utility.

4.2 Hot Swap Kit Configuration Utility

The Hot Swap Kit Configuration Utility allows you to modify the default settings for your system. This utility can be found in the HSK program group under the Start Menu.

4.2.1 Physical Slot Numbering

The Physical Slot Numbering tab allows you to change the default mapping between the physical slot number which might be displayed on the external chassis of the system and the logical slot number which is known to the host system. Physical slot numbers are integer numbers, starting from 1; they need not be consecutive.
In the absence of a standard method to define the physical numbering of slots in CompactPCI systems, the Hot Swap Kit establishes the physical slot numbering for known platforms at the time of installation. To reconfigure the default or to enter the slot mapping for other platforms, this utility should be used.
The Physical Slot Numbering information is stored in the registry and has the following format:

\System\CurrentControlSet\Services\busfiltr\Parameters\...
\PhysicalSlots

Key	Value
SlotNumber	*SlotPath*(ex. 7, "7840")

There should be an entry for each non system slot in the chassis.
\SystemSlots

Key	Value
SystemSlotNumber	""(ex. 8, "")

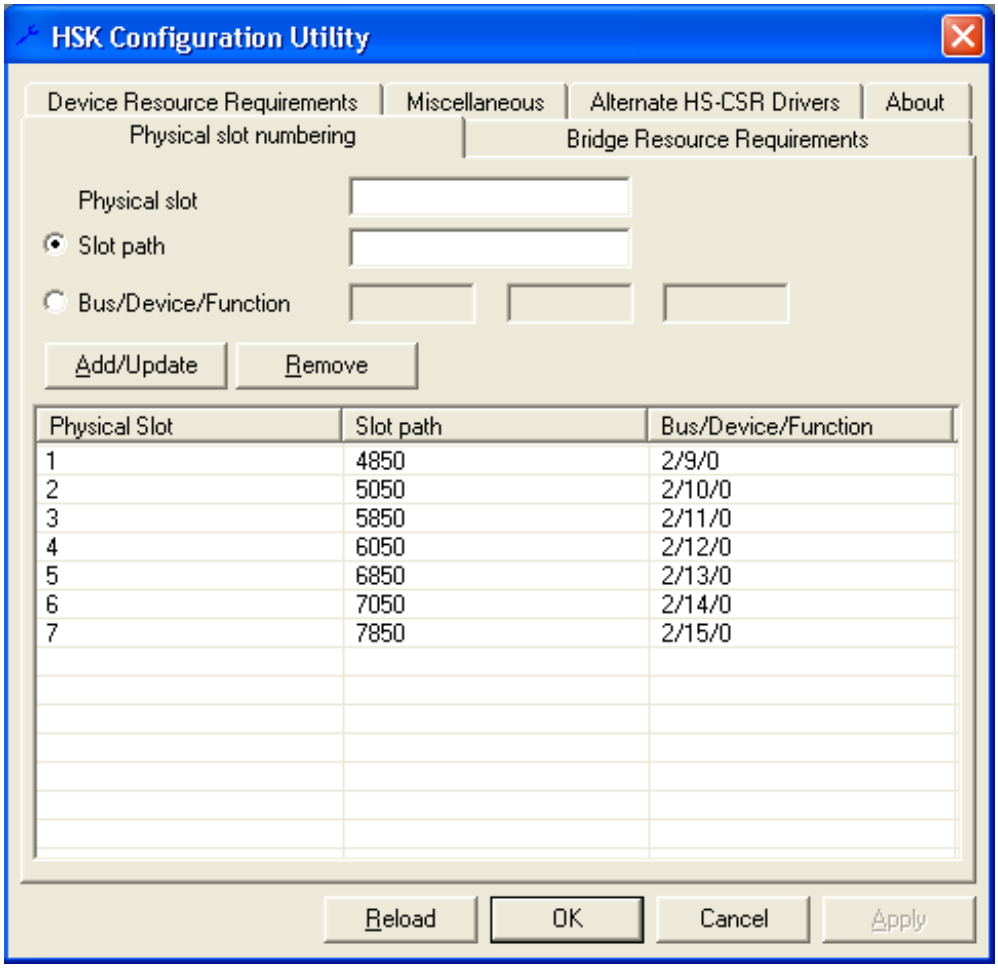

Fig 4.1: Physical Slot Numbering Window

4.2.2 Bridge Resource Requirements

Typically, CompactPCI bus segments are bridged to the root PCI segment. During system initialization, the BIOS usually assigns just enough resources to these bridges to accommodate existing devices. Unfortunately, with a hot swap environment, not all devices will

necessarily be available at system initialization. In this case, the windows assigned by the BIOS may be too small to allow later hot insertion.

To address this problem, the HSK allows you to specify the initial resource requirements for the PCI-PCI bridges leading to CompactPCI segments. The HSK will override the BIOS memory assignments with the specified allocations from the resource assignment settings.

Bridge Resource Requirements can be set for the bridge I/O window, the memory window, and the prefetchable memory window. Each resource requirement can be adjusted by the size of the window as well as the window boundaries that the allocation can be made. This will allow you to guarantee that the address allocations to devices behind the bridge will fall between the "From" and the "To" settings. All values specified are hexadecimal.
In Figure 4.2, the PCI-PCI bridge identified by the slot path "88" (bus 0, device 17) has the following requirements: 8 Kb of I/O space, 80 Mb of memory space and 80 Mb of prefetchable memory space.

For each window, alignment can be specified. By default, bridge I/O windows are aligned to 4Kb boundary, and memory windows are aligned to 1Mb boundary. Larger alignment may be required if devices with large resource requirements are placed behind this bridge (alignment for device resources is always power of 2 and cannot be changed). For example, a 64Mb memory device resource must be aligned to 64Mb boundary. Even if the parent bridge window is larger but is not aligned properly, the device resource may not fit into the window.

For systems with built-in support in this release of the HSK, a default set of memory allocations are automatically created during the installation. Each bridge, leading to the CompactPCI segment is preconfigured to 8Kb of the I/O window and 80 Mb of both regular and prefetchable memory windows. The preconfigured memory allocations can be adjusted by modifying the size field for the respective resource assignments and saving the new values.

Normally, the BIOS will assign the bus numbers to be used. BIOS assignment of bus numbers is usually done sequentially, leaving no room for a new bus to be inserted. A user can choose to allow dynamic bus numbering to occur by checking the dynamic bus number assignment checkbox.

The bridge resource requirements are stored in the registry and have the following format:

\System\CurrentControlSet\Services\busfiltr\Parameters\...
\ResourceAssignments

Key	Value
BridgeSlotPath	*Resources Assignments*

The format of Resource Assignments will contain the hexadecimal assignments for I/O space, Memory Space, Prefetchable Memory, and the secondary bus number for the bridge

that should be assigned dynamically. Value 0 means that the system will choose the bus number; absence of the last parameter inhibits dynamic bus number assignment. Example:

40, “4000,5000000,5000000,0”

means that the bridge at “40” will get 16Kb of I/O space and 80Mb of regular and prefetchable memory each, and the secondary bus number for the bridge will be assigned dynamically.

The following example:

40, “4000,5000000,5000000”

means the same resource assignments but the secondary bus number assignment made by BIOS will not be changed.

Non-default alignment can be specified by placing it after the size of the corresponding resource, separated by the colon character. The following example requests that both memory windows be aligned to the 64Mb boundary:

40, “4000,5000000:4000000,5000000:4000000”

To specify non-default boundaries for the bridge window allocation, the corresponding resource group is placed in parentheses and starting and ending boundaries are listed after the size and alignment, separated by commas.The following example is similar to the previous example, but requires that the prefetchable memory region be placed above the bus address 0xC0000000:

40, “4000,5000000:4000000,(5000000:4000000,C0000000,FFFFFFFF)”

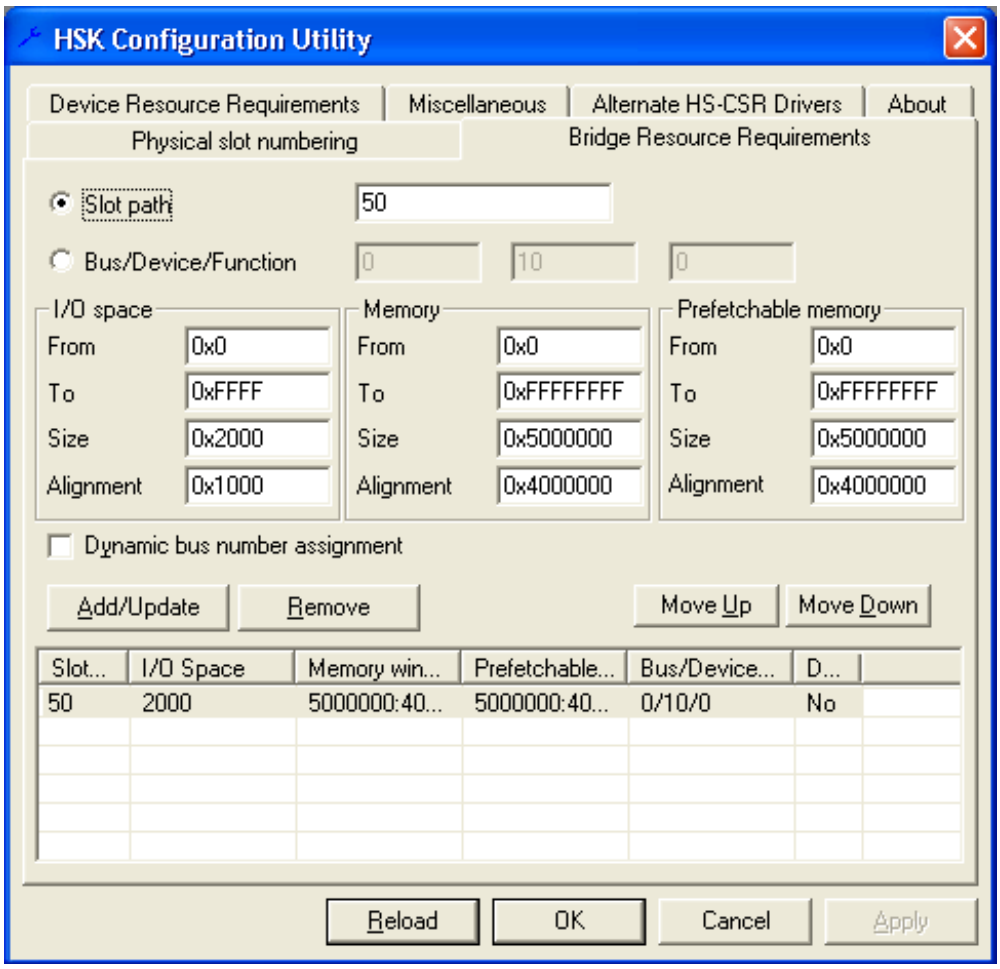

Fig 4.2: Bridge Resource Requirements Window

4.2.3 Device Resource Requirements.

When a PCI-PCI bridge device (for example, a PMC carrier) is hot inserted, and no explicit bridge resource requirements are specified for it, HSK scans the bus below it to find out how many resources this bridge needs. To do that, HSK calculates recursively the resource requirements of all devices below the bridge. HSK determines device resource requirements from the device Base Address Registers (BARs). The calculated totals are specified by HSK to the system as resource requirements for the bridge windows.

However, BARs for some devices can be programmed by device drivers so that they will actually have resource requirements different from what they originally reported. To accommodate this possibility, the configuration utility allows the user to specify explicit resource requirements for specific BARs of specific PCI devices on the Device Resource Requirements page.

The PCI device subject to explicit resource requirements definition is specified by its PCI identification attributes: Vendor ID, Device ID, Revision ID, Subsystem Vendor ID and Subsystem ID. The last three attributes are optional.

The explicit resource requirements for this device are specified by the user on a per-BAR basis. Fields for those BARs that don't have their resource requirements changed by the driver software, should be left blank.

The device resource requirements are stored in the registry according to the following format:

\System\CurrentControlSet\Services\busfiltr\Parameters\...
\ExplicitResourceRequirements

Key	Value
PCI Device ID	*Multistring; each component specifying one BAR*

The registry key name is the PCI device identifier, its format is compatible with the format of Hardware IDs for PCI devices. The registry value is a multistring. Each component of the multistring defines requirements for one BAR and consists of the BAR number and resource requirement for that BAR, separated by colon. The strings that the represent registry key name and the value are shown in the bottom part of the page.

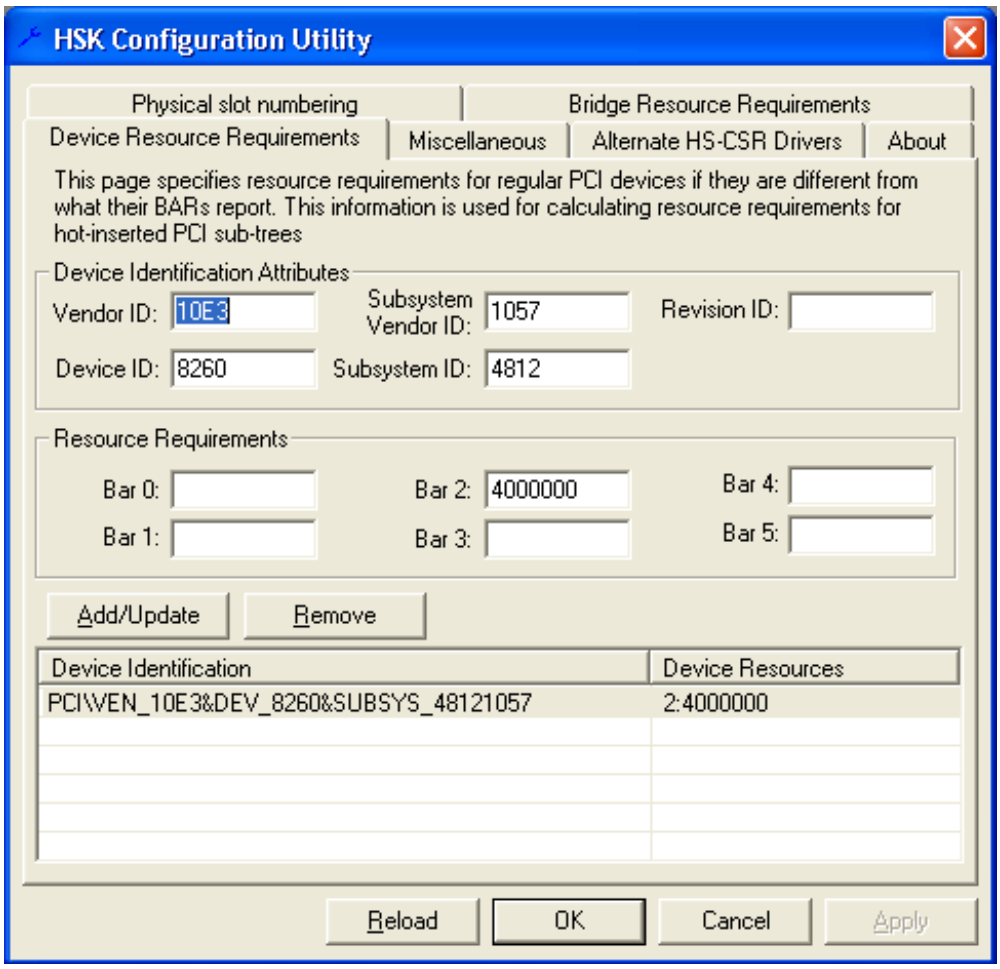

Fig 4.3: Device Resource Requirements Window

4.2.4 Miscellaneous

Modifying the PCI Configuration Space Polling Rate

The polling rate represents the time (in milliseconds) between subsequent invocations of the polling thread. During each invocation, the polling thread reads the PCI configuration space detecting arrival and departure of CompactPCI devices. Detection of the ENUM# signal causes the polling thread to wake up immediately rather than waiting for the expiration of the polling interval. The HSK defaults the polling rate to 500 milliseconds. Setting this value to 0 prevents the Hot Swap System Driver from all polling. If the Hot Swap Kit is unable to detect the ENUM# signal and the polling value is set to 0, the board will not be detected unless the APIs are invoked to recognize this board explicitly.

Modifying the “Scan for Surprise Removals” flag.

This flag controls whether HSK is able to detect surprise removals of devices even if polling rate is set to 0. If this flag is set, HSK still polls existing devices periodically and initiates surprise removal procedure if it detects that the device has disappeared from configuration space. HSK does not look for new devices in this mode unless ENUM# signal has been detected. If this flag is not set, HSK polls only devices awaiting orderly removal (with the blue LED illuminated).

Modifying Software Disconnection Timeout

The Software Disconnection Timeout represents the time (in milliseconds) between subsequent attempts of software disconnection in the case of the board handle being held down. If the initial software disconnection request is vetoed, the Hot Swap System Driver will repeat the request after the specified time interval expires.

The HSK defaults the software disconnection timeout to 20000 milliseconds (20 seconds). Setting this value to 0 prevents the Hot Swap System Driver from issuing additional software disconnection requests, if the initial software disconnection request has been vetoed.

Modifying “Insert Before Eject” timeout.

This value specifies an interval timer that is activated on a per-device basis when an insertion request takes place for the device and a pending extraction request exists for that device (that is, IoRequestDeviceEject has been issued before). If the Eject IRP arrives for this device when this timer is active, it is considered to be a stale Eject IRP from the previous ejection request, and is rejected. Thus, if the time difference between the insertion request and the EJECT IRP arrival is no more than ”Insert Before Eject” milliseconds, the stale EJECT IRP will be discarded, the device will be re-activated and its state will be consistent with the position of the handle. See section 3.3 for more information.
The HSK default value for this timeout is 5000 milliseconds (5 seconds).

The Polling Rate, Software Disconnection Timeout, “Insert Before Eject” Timeout and Scan for Surprise Removals flag are stored in the registry. These can be found in the following location:

\System\CurrentControlSet\Services\hspci\Parameters\...

Key	Value
DisconnectRetryInterval	*timeout*
StallInterval	*polling rate*
InsertBeforeEjectInterval	*timeout*

NoScanForSurpriseRemoval*the value opposite to the flag*

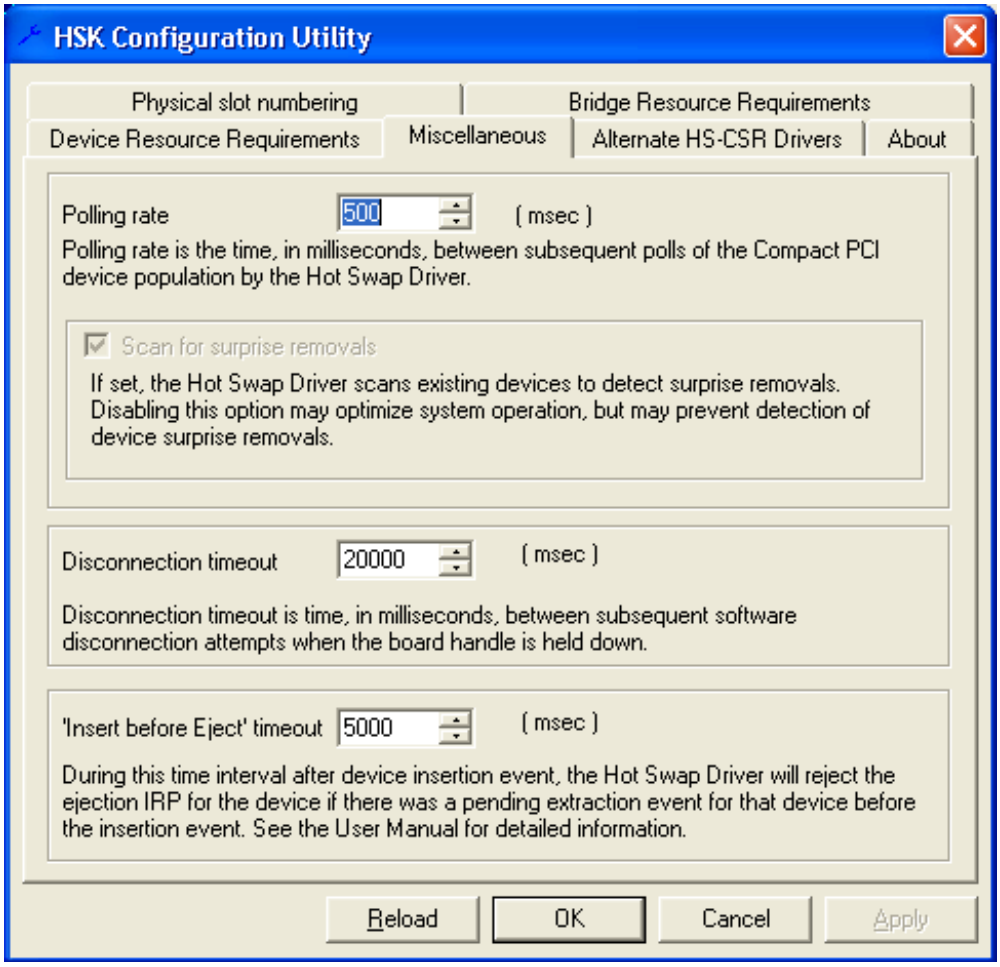

Fig 4.4: Miscellaneous Window

4.2.5 Miniports

Miniports are used when a board does not use the recommended approach for a Hot Swap Friendly Hot Swap Control and Status Register (HS_CSR) implementation, but rather uses an alternate HS_CSR implementation.

The HSK comes with a default set of miniports which are listed in Figure 4.4. In the event that you want to use a non-supported board and the board requires a miniport, you will need to add a new miniport entry. Simply click the Add button and browse to find the location of the miniport driver that you want to load. Once you have found the appropriate miniport, you will need to press the Save button.

The information relating to what miniports should be loaded and where they are located can be found in the registry and has the following format:

\System\CurrentControlSet\Services\hspci\Parameters\...

Key	Value
Miniports	*list of miniports*

The list of miniports is a MultiString and contains a list of each miniport to be loaded. For example: “cpv8540 i21554 force731 zt554x”.

The order that the Miniports are listed is very important as miniports will be loaded in order starting with the first one. The miniport list should start with the most generic miniports and then become more specific as the list is parsed. If there is a board that can be supported by multiple miniports, you should make sure that the miniport you want to load should be listed last. For example the i21554 miniport supports boards that contain an Intel 21554 bridge chip. The Ziatech ZT554X board contains an Intel 21554 bridge chip, but also requires additional support covered by the zt554x miniport driver. The i21554 miniport should be listed first, with zt554x miniport coming after. You can use the “Move Up” or “Move Down” buttons to adjust the order of miniports.

Each miniport should have an associated registry key under the Services subkey:

\System\CurrentControlSet\Services\...
miniportname

Key	Value
ImagePath	*“location of miniport”*
ERRORCONTROL	0x1
START	0x4
TYPE	0x1

Here is an example of the location for a miniport:

\System\CurrentControlSet\Services\cpv8540\...

Key	Value
ImagePath	“system32\drivers\cpv8540.sys”
ERRORCONTROL	0x1
START	0x4
TYPE	0x1

The restrictions field allows the user to specify to which slots the particular Alternate HSCSR driver may be applied. This can be very advantageous, when you want a miniport that can load for multiple boards to be restricted to only the boards that you want. The string is a catenation of slots and slot ranges separated by commas. For example, the restriction: "2,4-6" refers to a particular miniport allowed to be loaded only for slots 2, 4, 5, and 6. The restrictions field is stored in the following registry location:

\System\CurrentControlSet\Services\hspci\Parameters\RestrictMiniports\...

Key	Value
“*miniportname”*	*“list of slots miniport applies to”*

Example:
“cpv8540” “1-6,11-16”

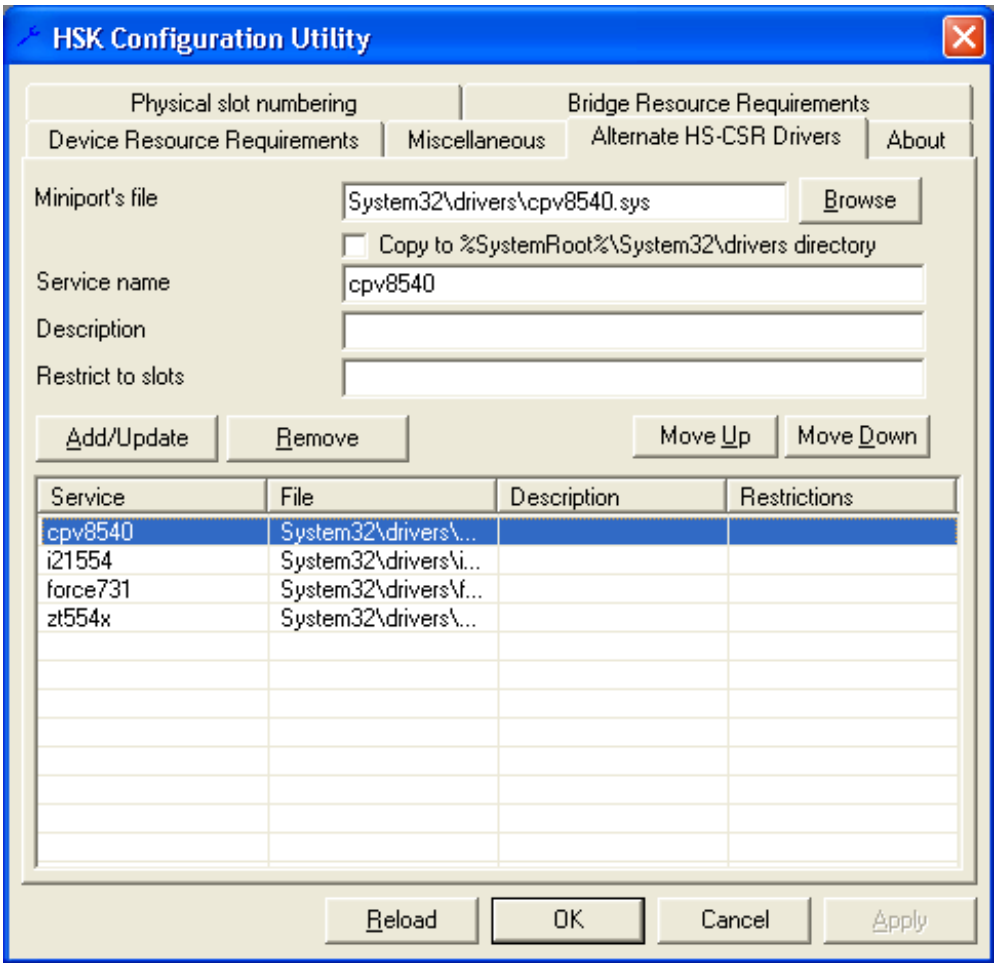

Fig 4.5: Miniports Window

4.2.6 About

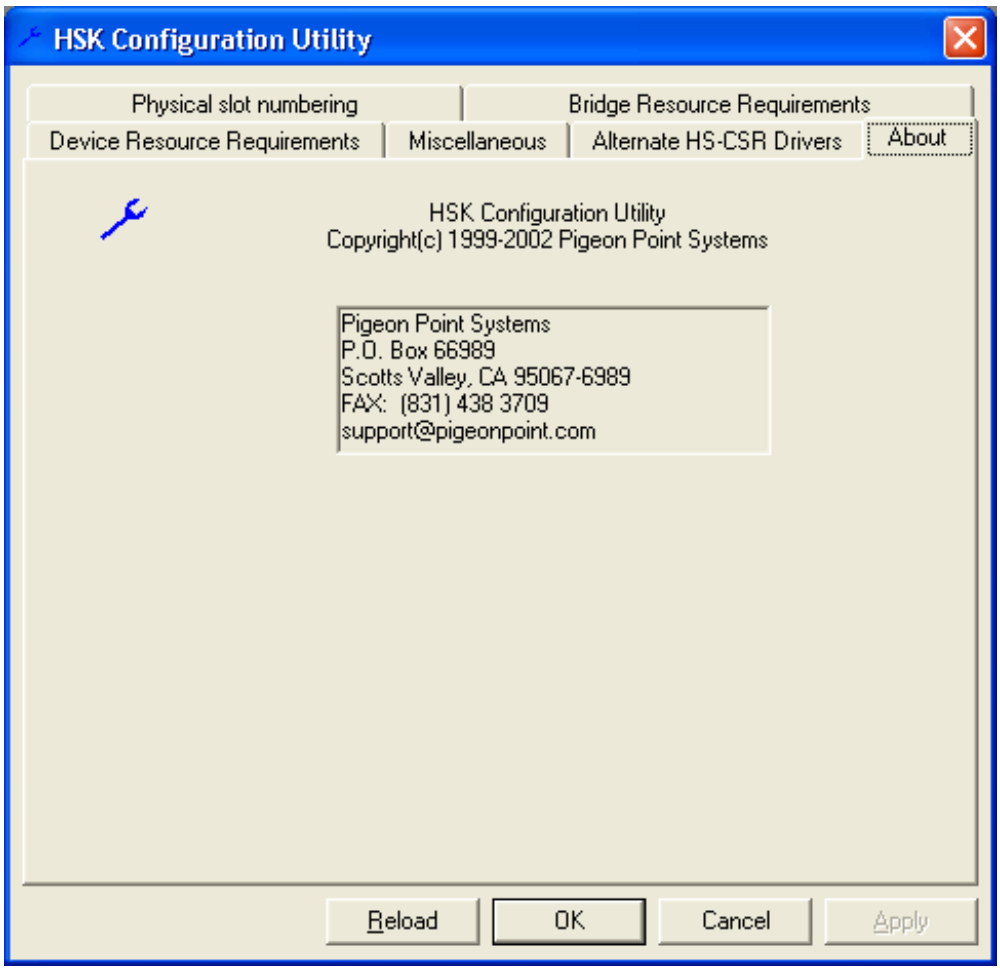

Fig 4.6: About Window

4.3 Slot Information Utility

The Slot Information Utility can be used to inventory the boards that are currently installed in the system. This utility can be found in the Hot Swap Kit program group under the Start Menu.
The Physical Slot number identified for each board is determined by the settings in the physical slot numbering tab of the HSK configuration utility. The board information includes the slot path, the type of board, the PCI configuration space header information. The slot information displayed has the following format:
[Phys #]:
[slot-path]:
[Description of board]:
[Class Code, Sub-class Code, Programming Interface]:
[ID=(Vendor ID,Device ID,Revision ID,

SubsystemVendorID, SubsystemDevice ID)]

A slot with no configuration space information does not have a board inserted.

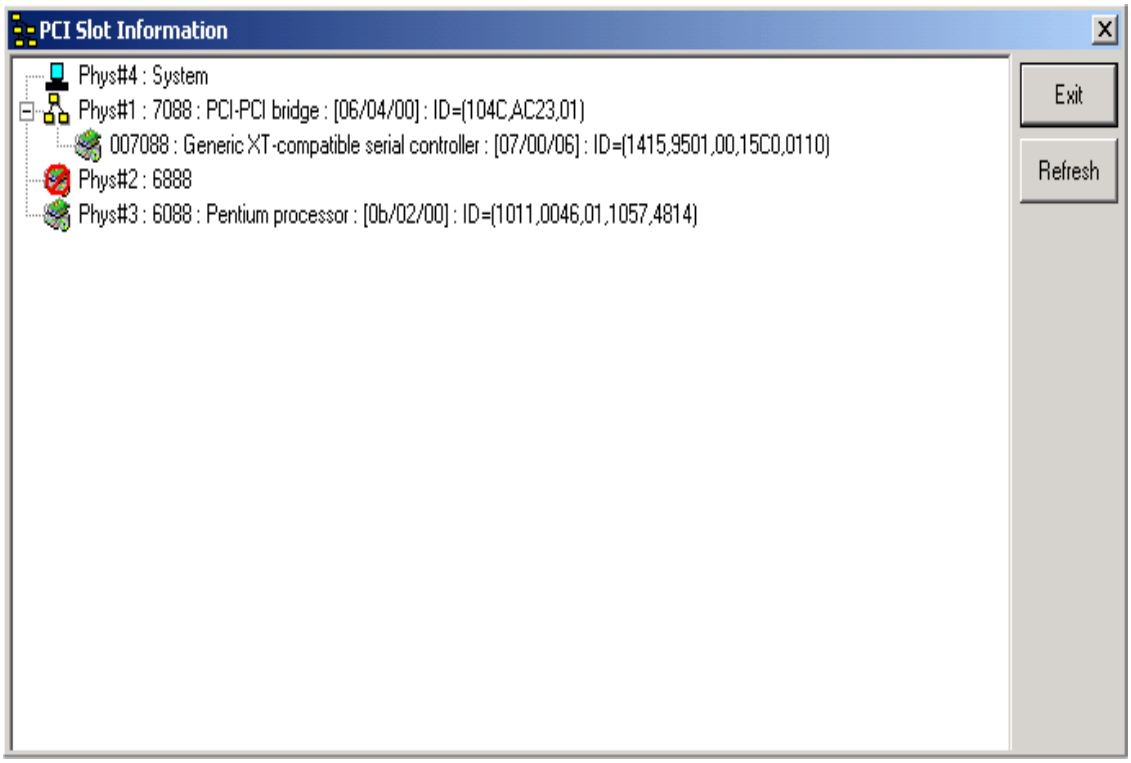

Fig 4.7: Slot Information Utility Window

4.4 Hot Swap Control Utility

This command line utility (hsctl.exe) is the user front end to the hot swap infrastructure. It gives access for the command line user to the hot swap functionality of the HSK.

The general invocation format is:
hsctl command [arguments]

4.4.1 Hot Swap Functionality

Extract

Format:

hsctl extract physical-slot-number

Synopsis:

This command simulates an extraction request for the board in the specified physical slot. Actually, this function simulates the board being unlocked. The hot swap infrastructure initiates software disconnection for the board. When the software disconnection is complete, the blue LED is illuminated.

The extraction will follow the Software Disconnection Timeout to determine if continued attempts will be issued in the event of initial disconnect failure. The Software Disconnection Timeout value specifies the time in milliseconds to wait before re-issuing the disconnection request. This will continue until the disconnection succeeds or you manually cancel the extraction. If this value is set to 0, no retries will occur. More information can be found in the Chapter 4 section entitled "Modifying Software Disconnection Timeout".

Example:

hsctl extract 6
Extract the board in slot 6.

Insert

Format:

hsctl insert physical-slot-number

Synopsis:

This command initiates cancellation of the extraction request for the board in the specified physical slot. This function simulates the board being locked. The hot swap infrastructure clears the blue LED on the board and initiates software connection for this board.

Example:

hsctl insert 6
Insert a board in slot 6.

4.5 RhDemo

The purpose of the demonstration utility is to demonstrate and expose the main functionality and features of the HSSDK driver set, the Application Programming Interface (API) and the Redundant Host (RH) capabilities of the ZT 5524 System Master CPU board. It also serves as a test tool for exercising the APIs while acting as a programming tutorial. The functional interfaces are listed below:

- RH API exercising
- Hot Swap API exercising
- Inter-host communications mechanism
- Fault Configuration
- Switchover Management
- Any extra exposed status and control that is not covered in the previously mentioned APIs

4.5.1 Functional Description

The architecture of the RhDemo application is represented by five major functional blocks:

- User interface

- RH interface
- IPMI interface (ZT5524 specific feature)
- Hot swap interface
- Slot Control interface (ZT5524 specific feature)

These are described in the following topics.

4.5.2 User Interface

The user interface is based on a command line interface and is menu driven. Enter a number and press Enter to make a selection. Press M to go to the main menu, press B to go back to the previous menu, and press Q to quit the demo.

4.5.3 RH Interface

The RhDemo exercises the Redundant Host functionality exposed via the RH interface. It supports the PICMG 2.12 RH API. This should also be sufficient to exercise the functionality in the ZT5524 / ZT5550 RHost System Master board. The ZT 5524 is dynamic enough to function in multiple mode host-domain ownership configurations. The multiple modes are:

- Active/Standby
- Active/Active
- Cluster

ZT5550 can own both domains only or be fully Backup.

A Standby Host is a host that does not control a bus domain. A Standby Host is referred to as being in Standby mode. An Active Host is a host that owns at least one bus segment. Functionality such as software initiated handovers, hardware initiated failovers, switchovers, event reporting and alarms are exercised.

4.5.4 Software Initiated Handovers

Software initiated handovers allow an active system master board to switch over to the backup host through application software intervention. This allows the user to perform preventative maintenance or software upgrades to one host without shutting down the entire system. During a handover, the device drivers are allowed to quiesce activity to the devices and synchronize state information to allow an orderly transition of a bus segment.

4.5.5 Hardware Initiated Failovers

Hardware initiated failovers occur when a catastrophic failure occurs on the active system master board. The active host can then failover to the backup host or the backup host can perform a takeover so that interruptions to system operation are minimized. Examples of catastrophic failures are a software watchdog timeout or a detected voltage spike that may render the CPU unstable. These events warrant a hardware-initiated failover.

4.5.6 Multiple Mode Capabilities

4.5.6.1 Active/Standby Mode

The Active/Standby mode is the standard Redundant Host configuration. This mode allows only one system master CPU board to have visibility to all backplane bus segments and all the connected PCI devices. This mode requires that the standby system master CPU board be electrically disconnected from the backplane at the PCI-to-PCI Bridge. A PCI spoofing mechanism is required for proper operation. The spoofing mechanism allows the standby host to access the PCI configuration space of backplane devices without having direct access to the devices. If a host fails and requires a takeover, one of the hosts initiates a handover or failover and upon completion the roles of active and standby hosts are reversed.

4.5.6.2 Active/Active Mode

Active/Active mode configuration allows each board segment to control a single bus segment. Each system master CPU board controls the clock and arbitration for its controlled or owned bus segment. It is through the PCI spoofing mechanism that each system master has visibility to the bus segment and the devices that are owned by the redundant host. In this mode if one host fails then the redundant host can take ownership of the relinquished bus segment.

4.5.6.3 Cluster Mode

Cluster mode is a variant on the Active/Active host mode. In Cluster mode if either host fails then the bus assigned to the failed host is unavailable for ownership transference. This is referred to as bus locking. While a system is dynamically capable of transitioning between Active/Standby and Active/Active modes, and even into a Cluster mode, it is only through a system power cycle that a system can transition out of Cluster mode. This is due to the fact that a locked bus segment may not have PCI spoofing information consistent across multiple host domains.

4.5.7 Switchover Functions

The RhDemo exposes the following functionality:

- Prepare for Switchover
- Cancel Prepare for Switchover
- Get Domain Software Connection Status
- Get Slot Software Connection Status
- Perform Switchover
- Set Hardware Concession Host
- Get Hardware Concession Host

4.5.8 Host Domain Enumeration and Association

The RhDemo enumerates hosts and domains, reports host-domain associations and returns useful data on the domains, hosts and slots. It covers the following functions:

- Get Domain Count
- Get Domain Numbers
- Get Domain Ownership
- Get Domain Slot Path
- Get Domain Slot Count
- Get Domain Slots
- Get Slot Domain
- Get Current Host Number
- Get Host Count
- Get Host Numbers
- Get Host Name
- Get Host Availability
- Get Domain Availability to Host

4.5.9 Slot Information

The RhDemo returns the following device information on the system slots:

- Physical slot information
- Slot Child Information

4.5.10 Notification, Reporting and Alarms

The RhDemo reports the following switchover notifications, alarms and other events:

- Enable Domain State Notification
- Enable Switchover Notification
- Enable Switchover Request Notification
- Enable Unsafe Switchover Notification
- Disable Notification

4.5.11 IPMI Interface (ZT5524 specific feature)

The IPMI interface is an important element of the ZT5524 RHost system architecture. It is used extensively for system management and inter-chassis communications:

- Access
- Configure
- System management
- Fault configuration and management
- Isolation strategies
- Inter-host communications

The IPMI interface exercises the following IPMI API, fault configuration and system management functions:

Get Temperature Sensor Status/Thresholds - Gets the temperature sensor status and readings.
Set Temperature Sensor Status/Thresholds - Sets the temperature sensor status and thresholds.
Get Voltage Sensor Status/Thresholds - Gets the voltage sensor status and readings.
Set Voltage Sensor Status/Thresholds - Sets the voltage sensor status and thresholds.

4.5.12 Fault Configuration

The RhDemo performs the following fault configuration activities:
- Upper/Lower non-critical threshold
- Upper/Lower critical threshold
- Upper/Lower non-recoverable threshold

4.5.13 Isolation Strategy

The RhDemo executes one of the following isolation strategies, depending on the occurring event:
- Alert
- Power Off
- Reset
- Power Cycle
- OEM Action
- Diagnostic Interrupt (NMI)

4.5.14 Hot Swap Interface

The basic purpose of the CompactPCI hot swap functionality is to allow orderly insertion or extraction of CompactPCI boards without affecting operation of the system involved. The hot swap interface in this demo operates under Linux* and VxWorks*. The current demo version does not support hot swap functionality. However, this new HS module does demonstrate manipulation of the Hot Swap API (HS API).

4.5.15 HS Functional Description

The HS module exercises/simulates these capabilities:
1. Hot swap board insertion
2. Hot swap board extraction
3. Slot information retrieval
4. PCI tree information retrieval
5. Catching and printing of notification messages

4.5.16 Hot Swap Board Insertion

Hot swap board insertion can be simulated by the demo application.

4.5.17 Hot Swap Board Extraction

Hot swap board extraction can be simulated by the demo application.

4.5.18 Slot Information Retrieval

If this functionality is performed, information on the board is retrieved based on the slot path. The type of information retrieved from the selected PCI device is described in chapter 9, “High Availability Slot Control Interface,” in the *PICMG 2.12 CompactPCI Hot Swap Infrastructure Interface Specificatio*n. For more details on PICMG, see Section H.1, “CompactPCI” on page 131.

4.5.19 PCI Tree Information Retrieval

When this functionality is executed, the related API call returns a list of PCI devices available on the system. Flags are set for each device to determine its state at that particular time. See the “PCI Tree Information Retrieval Flags” table.
The following information is displayed:

- Slot path
- Vendor ID
- Device ID
- SubsystemVendor ID
- SubsystemDevice ID
- Class code D
- Sub class code
- Programming interface
- Header type
- Flag

4.5.20 Catching and Printing Notification Messages

This demo application covers the capability of catching and printing notification messages sent by HSSD when an event is triggered. The following event types are supported. The flags supported are defined in the “Event Notification” topic of the *Intel ® NetStructure ™ Hot Swap Kit for Linux 4.2 software manua*l. For details on obtaining this manual, see Section H.2, “User Documentation” on page 131.

4.5.20.1 Slot Information Structure

The slot information structure contains information about a specific slot, identified by a slot path. It includes the following pieces of information:

- Path to the slot

• Current bus number, slot number and function number of the slot
• Physical slot number
• Physical slot depth

Table 3. PCI Tree Information Retrieval Flags

Flags	Meanings
PRES	Device is present.
CONN	Device is software connected.
CONF	Device's software connection failed.

Table 4. Events that Generate Notification Messages

Event types	Meanings
EXTR REQ	Device extraction request.
EXTR CONF	Device extraction confirmed.
REMOVAL	Device removed.
INSERTION	Device inserted.

Demonstration Utilities

• Slot state flags

If the slot is not empty, the following fields are also present:

• Vendor ID
• Device ID
• Subsystem vendor ID
• Subsystem ID
• Revision ID
• Class, subclass, programming interface
• Header type
• HS-CSR, if any

4.5.21 Slot State

When the slot information structure is filled in as a result of a call, the HsStateFlags field contains a set of flags representing the current state of the slot. The following flags are defined:

4.5.22 Slot Control Interface (ZT5524 specific feature)

Slot control capability is defined in the Redundant Host System Model, where the capabilities of the system are extended to allow software control of a board's hardware connection state. The system software adds drivers and services for this greater degree of control. This allows software to electrically isolate the board from the system until an operator is available to do so physically.

• Slot control enables:
• Capability to perform reset on the board
• Change the board's power state to ON
• Checks the presence and health of the board

Slot State Flags

HS_STATE_DEVICE_PRESENT A device is present in the slot

HS_STATE_SW_CONNECTED A device is present in the slot and software connected

HS_STATE_EXTRACTION_PENDING Extraction request pending for the device in the slot

HS_STATE_READY_FOR_EXTRACTION Device is ready for extraction, the blue LED is lit

5. The Hot Swap Kit Application Programming Interface

5.1 General

This chapter defines the HSK API for Windows 2000/XP. It is mostly oriented towards information services about the state and population of the CompactPCI bus. Also the API includes facilities to simulate toggling the board handle (which involves initiating software connection and disconnection) on a particular CompactPCI board.

5.1.1 Slot Path

Slot path is the means of slot identification used in Windows 2000/XP irrespective of possible bus number changes. Each slot is identified by the sequence of slot numbers: the first is its own, the second is the slot number of the nearest bridge on the upper bus, the third is the slot number of the next bridge, etc. up to bus 0. Slot numbers are in BIOS format (slot << 3 | function, in hexadecimal). Example: suppose that we identify device at (2,12,0), and bus 2 is the secondary bus for the bridge at (0,17,0). The slot path for this device will be "6088". The first slot number is 60 hex, that corresponds to the device itself (slot 12, 0xC << 3); the second slot number is 88 hex and corresponds to its parent bridge (slot 17, 0x11 << 3).

5.1.2 Physical Slot Numbers

The HS infrastructure supports physical slot numbers, treated as arbitrary non-zero 32-bit unsigned numbers (but recommended to correspond to the physical slot numbers established by the platform and intended for communication with the user). Mapping between physical slot numbers and slot paths is assumed to be known a priori to the infrastructure. For PCI slots that reside behind actual physical slots (like PMCs) the infrastructure provides a pair of numbers: the number of the nearest physical slot above this slot and the depth (number of levels) between this slot and the nearest physical slot. For actual physical slots, the depth is 0. For all other slots, the physical slot number is 0.

5.1.3 The Slot Information Structure

The slot information structure contains information about a specific slot, identified by a slot path. It includes the following pieces of information:

- slot path to the slot
- current bus number, slot number and function number of the slot
- physical slot number
- physical slot depth
- slot state flags

If the slot is not empty, the following fields are also present:

- vendor ID
- device ID
- subsystem vendor ID
- subsystem ID
- revision ID
- class, subclass, programming interface
- header type
- HS-CSR, if any

The slot information structure is represented by the following C type definition:

```
typedef struct tagHS_SLOT_INFO
{
    USHORT Version;                              // now 1
    USHORT Size;   // Size of the fixed part
// Information about the slot location – more in the text part
    UCHAR Bus;
    UCHAR Device;
    UCHAR Function;
    UCHAR PhysSlotDepth;
    ULONG PhysicalSlot;
// Device information - does not make sense if the slot is empty
    USHORT VendorID;
    USHORT DeviceID;
    USHORT SubVendorID;
    USHORT SubSystemID;
    UCHAR RevisionID;
    UCHAR BaseClass;
    UCHAR SubClass;
    UCHAR ProgIf;
    UCHAR HeaderType;
// Hot Swap related information, including slot state
    UCHAR Reserved;
    UCHAR HsCsr;
    UCHAR HsStateFlags;
    // The slot path follows
    TCHAR SlotPath[1];
} HS_SLOT_INFO;
```

5.1.4 Slot State

When the Slot Information structure is filled in as a result of a call, the HsStateFlags field will contain a set of flags representing the current state of the slot.
The following flags are defined:

HS_STATE_DEVICE_PRESENT	a device is present in the slot
HS_STATE_SW_CONNECTED	a device is present in the slot and software connected
HS_STATE_EXTRACTION_PENDING	extraction request pending for the device in the slot

HS_STATE_READY_FOR_EXTRACTION	device is ready for extraction, the blue LED is lit

5.1.5 Unicode and ANSI Versions of the API Functions

All functions that accept parameters or yield results of type TCHAR or structures including fields of type TCHAR, are present in two versions: for Unicode and ANSI character sets. The actual version is conditionally chosen during compilation depending on the current Unicode settings.

The naming conventions are the same as in the rest of Windows 2000/XP: the Unicode functions have the suffix W after their names, ANSI functions have suffix A. Generic names, listed below, are defined to one of the two explicit names with the suffix, depending on the Unicode settings for the current compilation.

For example, there are two versions of the API function HsGetSlotInfoBySlotPath: the ANSI version HsGetSlotInfoBySlotPathA and the Unicode version HsGetSlotInfoBySlotPathW. The application may call a specific version of the API function by its explicit name, if it needs to override Unicode settings for the current compilation for a specific call.

5.1.6 Persistent Nature of the Extraction Request in the HSK

In the HSK API model an extraction request is persistent. If the device extraction request is rejected, the Hot Swap System Driver will submit another request after the specified timeout. Extraction can be cancelled only by the handle physically going up or by calling the API function HsRequestInsertion. If a board is unlocked, all registered parties must agree to extract the device before the device will be software disconnected. During the period of time where the device has not been software disconnected, the user can lock the board and the extraction will be cancelled. So, if an application holds a handle to the device and ignores the extraction notification, it actually rejects the extraction request, but it will receive another request in some period of time.

The timeout between subsequent requests issued by the Hot Swap System Driver is a configurable parameter which can be set by the Hot Swap Kit Configuration Utility. It is possible, however, to suppress the behavior described above by setting this parameter to 0. In that case, only one extraction request will be issued; if it is rejected, no additional actions will be taken by the Hot Swap System Driver until the handle is toggled back up. The blue LED will not be illuminated.

5.2 Event Notifications

Notifications are sent by the infrastructure to applications as custom notifications when a change of state occurs for some slot. To receive these notifications, the application should obtain a handle to the Hot Swap System Driver's logical device (via the initialization func-

tion) and then register for notifications. The notifications will arrive as Windows messages WM_DEVICECHANGE, and the parameter *lParam* of this message will point to the structure defined below.

```
typedef struct tag HS_EVENT_INFO
{
    DWORD Size;               // Size of the whole structure in bytes
    DWORD Reserved1;          // Reserved field
    DWORD Reserved2;          // Reserved field2
    HANDLE Handle;            // The file handle to the Hot Swap device
    HS_NOTIFY_HANDLE NotifyHandle;// The notification handle
    GUID EventGuid;           // GUID of the event
    LONG NameOffset;          // Offset in bytes to the beginning of
                              // slot path
    HS_SLOT_INFO SlotInfo;    // Slot information structure
} HS_EVENT_INFO;
```

In Windows 2000/XP, the notifications are implemented as custom notifications for the Hot Swap System Driver. The Hot Swap System Driver registers a special device interface that should be accessed by applications to obtain the file handle, which is necessary for subscription to custom notifications. The HS_EVENT_INFO structure, defined above, is layout-compatible with the corresponding system-defined structure DEV_BROADCAST_HANDLE, which is the only custom notification parameter. Notifications correspond to events generated by the Hot Swap System Driver. A separate GUID corresponds to each of those events. The following GUIDs are defined.

```
GUID_HSPCI_INSERTION             - device inserted
GUID_HSPCI_REMOVAL               - device removed (physically)
GUID_HSPCI_EXTRACTION_REQUESTED  - device extraction requested
GUID_HSPCI_EXTRACTION_CONFIRMED  - device extraction confirmed
GUID_HSPCI_SW_CONNECTED          - software connection happened for the
                                   device
GUID_HSPCI_SW_DISCONNECTED       - software disconnection happened for
                                   the device
GUID_HSPCI_SW_CONNECTION_FAILED  - software connection failed for the
                                   device
```

The custom notifications provided by the Hot Swap System Driver augment the standard device interface activation-deactivation notifications of Windows 2000/XP. In particular. the custom notifications provide information about the hot swap events for a device, even if the driver for that device does not exist or cannot be started. The application that has subscribed to these events may in that case take some actions to fix the situation or notify maintenance personnel about the problem.

5.2.1 Comparison of HSK Event Notifications and Built-in Windows 2000/XP Event Notifications

The Windows 2000/XP operating system provides built-in facilities for notifying applications about device state changes. Since generality was a significant motivation for the

designers of this facility, they are based on a rather abstract and widely applicable concept of device interfaces.
An application can receive the following general notifications:

- Notification of device interface arrival (when a new interface is being registered); at this moment an application can open a handle to the device.
- Notification of a physical device removal.

An application can receive the following notifications, if it has an open handle to the device:

- Device query for removal;
- Cancellation of device removal.

For some applications in the CompactPCI environment, these notifications may be too general. If, for example, a driver does not register any device interfaces when starting the device, the application will never learn about the arrival of this device. Also, the application will never learn about the device arrival if the system cannot find resources for a newly inserted device or cannot find the corresponding driver. For some applications, the information about the physical device insertion may be quite necessary, even if the device driver cannot be found.

Hence the HSK provides its own set of notifications, which is intended to bridge the gap between the abstraction of device interfaces and the concrete picture of CompactPCI slots and devices. These notifications are based on the PCI slot path as the device identification attribute and provide more detailed information about Hot Swap events in the CompactPCI environment. For example, notifications are sent when a new device is physically inserted or when the resource allocation fails for the new device.

This set of facilities can be considered complementary to the Windows 2000/XP built-in notification facilities. That is, the application is encouraged to use built-in notifications, based on device interfaces, and the HSK internal core collaborates with the Windows 2000/XP kernel to better support them in the CompactPCI environment. Both facilities use the same built-in delivery mechanism (WM_DEVICECHANGE messages). The application may use the HSK notifications to obtain information that is more detailed and more specific to the CompactPCI environment.

From another point of view, the HSK notification set is functionally complete and can be used by itself. The two sets are functionally equivalent, if the device interfaces are managed by the functional driver in a standard way (enabled at logical device start and disabled at logical device removal). If the functional driver manages its device interfaces more creatively, the application should use the built-in facilities to track device interfaces. Appendix B provides an overall summary of the native Windows 2000/XP facilities for making applications aware of board arrivals and departures.

5.3 Hot Swap API Functions

This group supports hot swapping of CompactPCI devices in Basic and Full Hot Swap system models. Systems for which this function group is implemented should support at least the Basic Hot Swap model, though support of the Full Hot Swap model is highly recommended. These functions are defined in the header file hsapi.h and are implemented in the dynamic link library hsapi.dll.

5.3.1 Initialization and Termination Functions

```
DWORD HsInitialize
   (  OUT   HS_HANDLE *handle )
```

Synopsis:
This function should be called in the beginning to initialize communication between the application and the infrastructure. This function creates and returns a handle to the infrastructure that should be used in subsequent requests. The handle is the file handle opened via CreateFile() to the device interface alias exported by the hot swap infrastructure.

Parameters:

handle pointer to the variable that will hold the handle to the infrastructure of type HS_HANDLE. This type is generally opaque, but is typedef'ed to HANDLE.

Return Value:
The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsTerminate
   (  IN    HS_HANDLE handle )
```

Synopsis:
This function should be called at the end to gracefully terminate the communication between the application and the infrastructure.

Parameters:

handle the handle to the infrastructure obtained in HsInitialize.

Return Value:
The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsRegisterForNotifications
   (  IN    HS_HANDLE handle,
      IN    HWND hWnd,
      OUT   HS_NOTIFY_HANDLE *notifyHandle )
```

Synopsis:
This function registers the application to receive notifications about hot swap events in the system. These events will be delivered to the application as WM_DEVICECHANGE window messages to the specified window. These notifications are custom events provided by the hot swap infrastructure, and this function is just a thin wrapper to RegisterDeviceNotifications.

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.
hWnd	handle of the window to receive notifications.
notifyHandle	pointer to the variable that will hold the handle to the infrastructure of type HS_NOTIFY_HANDLE. This type is generally opaque, but is typedef'ed to HANDLE.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsUnregisterForNotifications
   (  IN    HS_NOTIFY_HANDLE notifyHandle )
```

Synopsis:

This function closes the existing registration for notifications about hot swap events. This function just calls UnregisterDeviceNotifications directly.

Parameters:

notifyHandle	the handle to the notification registration; this handle should have been obtained by a previous call to HsRegisterForNotifications.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

5.3.2 Informational and Enumeration Functions

```
DWORD HsGetSlotInfoBySlotPath
   (  IN    HS_HANDLE handle,
      IN    TCHAR slotPath[],
      OUT   HS_SLOT_INFO *slotInfo,
      IN    ULONG slotInfoSize )
```

Synopsis:

This function retrieves and fills in the slot information structure by slot path.

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.
slotPath	the slot path of the relevant PCI slot, in the form of a null-terminated character string.
slotInfo	pointer to the buffer where the slot information structure will be written;
slotInfoSize	the size of the buffer; if this size is too small for the output, this function will fail. The size of the maximum possible output is: FIELD_OFFSET(HS_SLOT_INFO, SlotPath) + 513*sizeof(TCHAR) (for the longest slot path in the system with 256 buses plus the null termination character).

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

Notes:

If the target slot is not occupied, one of two possible results can be generated: the call will fail or the call will succeed but the resulting slot information structure will contain HS_STATE_DEVICE_PRESENT with a value of FALSE and VendorID=Devi-

ceID=0xFFFF. The first result will be returned if no parent bridge exists for the specified slot. If the parent bridge exists but there is no device in the slot, the second result will be returned, even if the specified slot does not physically exist on the bus.

```
DWORD HsGetFirstOccupiedChildSlot
   (  IN    HS_HANDLE handle,
      IN    TCHAR slotPath[],
      OUT   HS_SLOT_INFO *slotInfo,
      IN    ULONG slotInfoSize )
```

Synopsis:

This function retrieves and fills in the slot information structure of the first child device behind the specified PCI-PCI bridge. The supplied slot path should identify a PCI-PCI bridge. This function will retrieve information about the immediate child device of the specified bridge with the least slot number. If the slot path is an empty string, the information about the first device under the root of the PCI tree will be returned.

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.
slotPath	the slot path of the PCI slot, in the form of a null-terminated character string; there should be a PCI-PCI bridge device in this slot. To enumerate devices on the PCI bus 0, use an empty string as the value of this parameter.
slotInfo	pointer to the buffer where the slot information structure will be written;
slotInfoSize	the size of the buffer; if this size is too small for the output, this function will fail. The size of the maximum possible output is: FIELD_OFFSET(HS_SLOT_INFO, SlotPath) + 513*sizeof(TCHAR) (for the longest slot path in the system with 256 buses plus the null termination character).

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsGetNextOccupiedSiblingSlot
   (  IN    HS_HANDLE handle,
      IN    TCHAR slotPath[],
      OUT   HS_SLOT_INFO *slotInfo,
      IN    ULONG slotInfoSize )
```

Synopsis:

This function retrieves and fills in the slot information structure of the next sibling device. The supplied slot path should identify a slot located immediately behind a PCI-PCI bridge. This function will retrieve information about the next device located immediately behind the same bridge and having the least slot number greater than the slot number of the slot identified by the slot path.

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.

slotPath the slot path of the PCI slot, in the form of a null-terminated character string.

slotInfo pointer to the buffer where the slot information structure will be written;

slotInfoSize the size of the buffer; if this size is too small for the output, this function will fail. The size of the maximum possible output is: FIELD_OFFSET(HS_SLOT_INFO, SlotPath) + 513*sizeof(TCHAR) (for the longest slot path in the system with 256 buses plus the null termination character).

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsGetSlotStateFlags
   (  IN    HS_HANDLE handle,
      IN    CHAR* slotPath,
      OUT   UCHAR* stateFlags )
```

Synopsis:

This function retrieves slot state flags for board with specified slot path. Slot state flags are reflecting the hot-swap state of the board.

Parameters:

handle the handle to the infrastructure obtained in HsInitialize.

slotPath the slot path of the PCI slot, in the form of a null-terminated character string.

stateFlags pointer to the variable where the slot state will be stored.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

5.3.3 Physical Slot Related Functions

```
DWORD HsTranslatePhysicalToSlotPath
   (  IN    HS_HANDLE handle,
      IN    ULONG physicalSlot,
      OUT   TCHAR slotPath[],
      IN    ULONG slotPathSize )
```

Synopsis:

This function retrieves and fills in the slot path corresponding to the specified physical slot. This function fails if the specified physical slot does not exist.

Parameters:

handle the handle to the infrastructure obtained in HsInitialize.

physicalSlot the physical slot number;

slotPath the buffer where the slot path of the PCI slot will be written as a null-terminated string.

slotPathSize the size of the buffer; if this size is too small for the output, this function will fail. The size of the maximum possible output is 513 characters (for

the longest slot path in the system with 256 buses plus the null termination character).

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsTranslateSlotPathToPhysical
   (  IN HS_HANDLE handle,
      IN    TCHAR slotPath[],
      OUT   ULONG *physicalSlot,
      OUT   ULONG *physicalSlotDepth )
```

Synopsis:

This function retrieves and fills in the physical slot number corresponding to the specified slot path. This function fails if the specified slot path does not identify any physical slot and does not identify a slot behind any physical slot.

This function yields the pair (physical slot number, depth). For actual physical slots, depth will be 0. For slots falling behind actual physical slots, the depth will indicate how many levels of PCI bridge hierarchy are between the specified slot and the physical slot above it.

Parameters:

handle the handle to the infrastructure obtained in HsInitialize.

slotPath the slot path of the PCI slot, in the form of a null-terminated character string.

physicalSlot the buffer where the physical slot number will be written.

physicalSlotDepth the buffer where the depth will be written.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsGetFirstPhysicalSlot
   (  IN    HS_HANDLE handle,
      OUT   ULONG *physicalSlot)
```

Synopsis:

This function retrieves the first (the least) physical slot number in the system.

Parameters:

handle the handle to the infrastructure obtained in HsInitialize.

physicalSlot the buffer where the physical slot number will be written.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsGetNextPhysicalSlot
   (  IN     HS_HANDLE handle,
      IN OUT ULONG *physicalSlot)
```

Synopsis:

This function retrieves the next physical slot number in the system (the least slot number greater than specified).

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.
physicalSlot	initially points to the reference physical slot number; the next physical slot number will be stored back into this buffer pointed to by physicalSlot.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsTranslateBdfToSlotPath
   (  IN    HS_HANDLE handle,
      IN    DWORD bus,
      IN    DWORD device
      IN    DWORD function,
      OUT   CHAR *buffer,
      IN    ULONG bufferSize );
```

Synopsis:

This function converts the physical PCI coordinates of a device (bus/device/function) to a slot path that may be used in subsequent calls to the HS APIs.

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.
bus	physical PCI coordinates (bus) of a device.
device	physical PCI coordinates (device) of a device.
function	physical PCI coordinates (function) of a device.
buffer	the buffer where the slot path will be stored.
bufferSize	the size of the buffer.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

5.3.4 Action Functions

```
DWORD HsRequestInsertion
   (  IN    HS_HANDLE handle,
      IN    TCHAR slotPath[] )
```

Synopsis:

This function simulates the insertion for the specified slot and cancels the extraction state for the slot (if an extraction request is pending for the slot). Actually, this function simulates the board being locked. This function initiates software connection for the device in the specified slot. The specified slot should not be empty.

This function does not indicate the success of insertion; successful return value only means that the insertion has been initiated.

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.

slotPath the slot path of the relevant PCI slot, in the form of a null-terminated character string.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsRequestExtraction
   (  IN    HS_HANDLE handle,
      IN    TCHAR slotPath[] )
```

Synopsis:

This function for the specified slot requests software disconnection as part of this process. Actually, this function simulates the board being unlocked.

The extraction, once started, is persistent; that is, the hot swap infrastructure tries to perform software disconnection repeatedly until it either succeeds or the extraction is cancelled manually by the operator or programmatically by calling HsRequestInsertion.

This function does not indicate the success of software connection; successful return value only means that the extraction has been initiated. The persistence of extraction is affected by the Software Disconnection Timeout parameter described in Chapter 4.

Parameters:

handle the handle to the infrastructure obtained in HsInitialize.

slotPath the slot path of the relevant PCI slot, in the form of a null-terminated character string.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsIsAvailableForExtraction
   (  IN    HS_HANDLE handle,
      IN    CHAR *slotPath )
```

Synopsis:

This function checks whether the board with the specified slot path is available for physical extraction from the chassis.

Parameters:

handle the handle to the infrastructure obtained in HsInitialize.

slotPath the slot path of the relevant PCI slot, in the form of a null-terminated character string.

Return Value:

The Win32 error code (0=ERROR_SUCCESS if the board can be extracted).

5.3.5 Device Name Functions

```
DWORD HsGetDeviceNameBySlotPath
   (  IN    HS_HANDLE handle,
      IN    TCHAR slotPath[],
      IN    GUID *devInterfaceGuid OPTIONAL,
```

```
    IN OUTULONG *Context,
    OUT   TCHAR devName[],
    IN    ULONG devNameSize )
```

Synopsis:

This function retrieves in an iterative way the device names that can be used by the application to open a handle to the specified device.
The iteration is controlled by the argument *Context*, which indicates the position of the next returned name in the list of names. To enumerate the whole list, the application should set *Context* to 0 before the first call and then call this function repeatedly without changing *Context* until it returns FALSE. The *Context* value will be updated automatically by each subsequent call.
To open the handle to the device, the application should submit one of the names returned by this function to the system function CreateFile().

Parameters:

handle	the handle to the infrastructure obtained in HsInitialize.
slotPath	the slot path of the relevant PCI slot, in the form of a null-terminated character string.
devInterfaceGuid	an optional interface GUID; if specified, it will limit the list of returned names to those generated from this GUID.
Context	the context field that is updated by this function; it indicates which name of all possible names this function should return. To get the first name in the list, context should be set to 0 before the call to this function; then it will be updated and the next call will return the next name from the list.
devName	the character buffer where the next name will be placed.
devNameSize	the size of the buffer; the function will fail if the size of the buffer is not sufficient. The size of the maximum possible output is 257 characters (the maximum *devName* size of 256 characters plus the null termination character).

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

5.3.6 Device Node Translation Functions

```
DWORD HsGetDevnodeBySlotPath
   (  OUT   DEVINST *pDevinst,
      IN    TCHAR slotPath[] )
```

Synopsis:

This function translates the slot path for a CompactPCI device to the device node handle for this device.
The device node handle is the handle used to identify the device in the Configuration Manager. After having obtained the device node handle, the application can use the configuration management APIs to control the device and get additional information about the device.

Parameters:

pDevinst pointer to the variable, which gets the device node handle.

slotPath the slot path of the relevant PCI slot, in the form of a null-terminated character string.

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

```
DWORD HsGetSlotPathByDevnode
   (  OUT   TCHAR slotPath[],
      IN    ULONG slotPathSize,
      IN    DEVINST devnode )
```

Synopsis:

This function translates the device node handle for a CompactPCI device to the slot path for that device.

The device node handle is the handle used to identify the device in the Configuration Manager. After having obtained the device node handle, the application can use the configuration management APIs to control the device and get additional information about the device.

Parameters:

devnode the device node handle.

slotPath the buffer where the slot path of the PCI slot will be written as a null-terminated string.

slotPathSize the size of the buffer; if this size is too small for the output, this function will fail. The size of the maximum possible output is 513 characters (for the longest slot path in the system with 256 buses plus the null termination character).

Return Value:

The Win32 error code (0=ERROR_SUCCESS on success).

6. The RHOST API

6.1 General

This chapter describes the Redundant Host API. Included in this section is a discussion on the data structures and naming conventions RHOST software uses to manage the CompactPCI chassis, domains, buses and events.

6.2 Intel-Specific APIs

6.2.1 RhSetHostName

Prototype:

```
RH_API_DEF HSI_STATUS RhSetHostName(
      IN    RH_HANDLE Handle,
      IN    uint32 Host,
      IN    char HostName[ ])
```

Synopsis:

This function sets the symbolic name of the specified host; this should be some kind of network name (for example, NETBIOS name or TCP/IP host name) that can be used to establish a network connection to that host.

Arguments:

Handle	The handle of the current session
Host	The host number
HostName	The character buffer where the host name is stored as a 0-terminated character string

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
RH_INVALID_HANDLE invalid session handle
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure
Other HSI_STATUS values if errors occurred during execution of this function such as nonexistent host

6.2.2 RhGetHwDestinationHostAndReset

Prototype:

```
HSI_STATUS RhGetHwDestinationHostAndReset (
      IN    RH_HANDLE Handle,
      IN    uint32 SourceHost,
      IN    uint32 Domain,
      OUT    uint32 *pDestinationHost,
      OUT    BOOL*pbReset )
```

Arguments:

Handle	The handle of the current session

SourceHost	The number of the source host
Domain	The domain number
pDestinationHost	pointer to the variable receives the number of the host that should own the specified domain if the source host fails and hardware-initiated switchover takes place for it
pbReset	pointer to the variable receives the state of the flag that indicates whether the specified destination host will perform a reset if the host receives control of a segment

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function gets the destination host that owns the specified domain and the reset flag value if a hardware-initiated switchover takes place due to the failure of the source host.

6.3 Redundant Host PICMG* 2.12 APIs

This chapter describes a supplementary API for domain management and switchover from the application level.
The interface described in this section is implemented as a set of functions exported to an application.
These functions allow the client to perform the following operations:
• Initialize and terminate an instance of this interface
• Enumerate the hosts, domains and slots in the system
• Get information about devices in slots
• Initiate domain switchovers among hosts
• Enable and disable notifications regarding switchover operations
• Specify actions that result from hardware-initiated alarms and control notifications about alarms
The following topics specify each of the interface functions.

6.3.1 Definitions and Types

The following definitions are provided for terms used in the remaining topics of this chapter.
RH (Redundant Host) System. An RH system consists of two or more hosts and one or more domains. An ownership relationship is defined between hosts and domains: hosts own domains. At any given moment of time, no more than one host can own one domain. If a host owns the domain, software on the host has access to PCI devices in (or behind) the PCI slots of the domain.

RH (Redundant Host) Infrastructure. An RH Infrastructure is an implementation of the Redundant Host PICMG 2.12 APIs for a specific RH System. It provides the Redundant Host API defined in this topic to applications. Multiple RH Systems of the same type may be serviced by a single infrastructure.
Domain. A Domain is a specific collection of peripheral PCI slots whose ownership can be transferred as a group among system hosts. PCI-PCI bridges can populate these slots. Therefore, a domain is generally a collection of PCI trees (a forest). Domains in the system are identified by 32-bit arbitrary quantities – domain numbers. The number of domains in the system and their domain numbers are assumed static during system operation.
Host. A Host is an active entity in the system that can run software that uses this API. Hosts in the system are identified by 32-bit arbitrary quantities – host numbers. The number of hosts in the system and their host numbers are assumed static during the system operation. The host number RH_NO_DESTINATION_HOST means "no host" and is used, for example, to say that no host owns the specified domain.
Domain parent bridge. A PCI-PCI bridge is called the parent for a domain if all slots behind that bridge belong to that domain and all slots of the domain are behind that bridge.
Domain slot path. The slot path for a PCI device is the sequence of device/function numbers from this device up the PCI tree to its root through the sequence of PCI-to-PCI bridges. Usually (but not necessarily) each domain has a domain parent bridge. When a host owns the domain, the slot path of the domain parent bridge is the domain slot path with respect to the host. Domain slot path may be defined with respect to a host even if that host does not own the domain, provided that the domain is guaranteed to have the same slot path each time it is switched over to that host.
Switchover. Switchover is changing ownership of a domain from one host to another.
Destination Host. This is the host that receives the specified domains owned by a particular host if a hardware-initiated switchover takes place on the owning host.
Available Host. A host is available if it can own domains and communicate with the rest of the RH system. A host is unavailable, for example, if it is switched off or is in some special mode in which it is isolated from the rest of the RH system.
Owning Host. The host that currently owns a domain.
Current Host. The host on which the specific API call has been made.
Root Bus. The root bus number of the PCI tree this slot belongs to. This value is 0 for the first or a single PCI tree. For additional PCI trees, this value is implementation-dependent, but is guaranteed to be non-zero.
RH Instance ID. A host can be a member of several RH systems simultaneously, similar to multi-homed hosts in networking. In that case, the application can use the Redundant Host API from several RH infrastructures. To select a specific RH system, the application uses the RH Instance ID when obtaining the handle to the RH system via RhOpen. RH Instance ID is an implementation-defined character string. To allow potential coexistence of multiple RH infrastructures on the same host, the RH Instance ID should consist of the RH infrastructure identifier and the identifier of a specific instance of the RH system (if multiple RH System instances are serviced by a single infrastructure).

The C definition of the associated types used by this interface is given below:

```
typedef enum {
      INACTIVE,
      DISCONNECTED,
      DISCONNECTING,
      CONNECTED,
      CONNECTING } RH_DOMAIN_SWC_STATE;
typedef enum {
      MINOR_ALARM,
      MAJOR_ALARM,
      CRITICAL_ALARM } RH_ALARM_SEVERITY;
typedef enum {
      ACTION_IGNORE = 0,
      ACTION_NOTIFY = 1,
      ACTION_SWITCHOVER = 2,
      ACTION_RESTART = 4 } RH_ALARM_ACTION;
typedef enum {
      NOTIFICATION_DOMAIN_STATE_CHANGE,
      NOTIFICATION_SWITCHOVER,
      NOTIFICATION_SWITCHOVER_REQUEST,
      NOTIFICATION_UNSAFE_SWITCHOVER,
      NOTIFICATION_ALARM } RH_NOTIFICATION_TYPE;
typedef enum {
      FULLY_COOPERATIVE,
      PARTIALLY_COOPERATIVE,
      FORCED,
      HOSTILE,
      HARDWARE_INITIATED } RH_SWITCHOVER_TYPE;
typedef struct PHYSICAL_SLOT_ID_STRUCT
{
      uint32 ShelfID;
      uint32 SlotID;
} PHYSICAL_SLOT_ID;
typedef void (*RH_DOMAIN_STATE_CALLBACK) (
      IN     uint32 Domain,
      IN     RH_DOMAIN_SWC_STATE State,
      IN     uint32 RequestingHost,
      IN     uint32 DestinationHost,
      IN     uint32 Timeout,
      IN     BOOLEAN Persist,
      IN     void *pContext );
typedef void (*RH_SLOT_STATE_CALLBACK) (
      IN     uint32 Domain,
      IN     PHYSICAL_SLOT_ID Slot,
      IN     RH_DOMAIN_SWC_STATE State,
      IN     void *pContext );
typedef void (*RH_SWITCHOVER_CALLBACK) (
      IN     uint32 Host,
```

```
      IN     uint32 Domain,
      IN     void *pContext );
typedef BOOLEAN (*RH_SWITCHOVER_REQUEST_CALLBACK) (
      IN     uint32 RequestingHost,
      IN     uint32 DestinationHost,
      IN     uint32 Domain,
      IN     void *pContext );
typedef enum {
      RESET_REQUIRED,
      RESET_NOT_REQUIRED,
      UNKNOWN } RH_SLOT_NEEDS_RESET;
typedef struct RH_SLOT_DESCRIPTOR_STRUCT {
      uint32 Size;
      PHYSICAL_SLOT_ID PhysicalSlot;
      uint8 PhysSlotDepth;
      uint32 OwningHost;
      uint16 BusNumber;
      uint8 DeviceNumber;
      uint8 FunctionNumber;
      uint16 VendorID;
      uint16 DeviceID;
      uint16 SubsystemVendorID;
      uint16 SubsystemID;
      uint8 RevisionID;
      uint8 BaseClass;
      uint8 SubClass;
      uint8 ProgIf;
      uint8 HeaderType;
      RH_SLOT_NEEDS_RESET NeedsReset;
      uint16 RootBus;
      char SlotPath[1];
} RH_SLOT_DESCRIPTOR, *PRH_SLOT_DESCRIPTOR;
typedef void (*RH_UNSAFE_SWITCHOVER_CALLBACK) (
      IN     uint32 Domain,
      IN     RH_SWITCHOVER_TYPE SwitchoverType,
      IN     BOOLEAN SlotResetSupported,
      IN     uint32 UnsafeSlotCount,
      IN    OUT    RH_SLOT_DESCRIPTOR *pUnsafeSlotDescriptors,
      IN     void *pContext );
typedef void (*RH_ALARM_CALLBACK) (
      IN     uint32 Host,
      IN     RH_ALARM_SEVERITY AlarmType,
      IN     void *pContext );

typedef void * RH_HANDLE;
```

6.3.2 Initialization /Termination

6.3.2.1 RhEnumerateInstances

Prototype:

```
HSI_STATUS RhEnumerateInstances(
      OUT    char *pInstanceID,
      IN     uint32 InstanceIDLength,
      OUT    uint32 *pActualSize );
```

Arguments:

pInstanceID	pointer to the character buffer where the list of RH Instance IDs are stored as a sequence of null-terminated character strings, terminated by two consecutive null characters
InstanceIDLength	the size of the buffer; if this size is too small for the output, this function fails.
pActualSize	this variable receives the actual size of the returned list of RH Instance IDs, in characters, including the terminating two null characters. In the case of the error code HSI_STATUS_INSUFFICIENT_BUFFER returned, this is the minimal required size of the buffer.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_NO_ DATA_DETECTED no known RH systems exist for the current host

HSI_STATUS_INSUFFICIENT_BUFFER returned if the buffer pInstanceID is too small to store the list of RH Instance IDs

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported on the current host

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function can be used to enumerate existing RH Systems on the current host, before doing an actual RhOpen call. The list of RH Instance IDs for RH Systems in which the current host participates is stored in the output buffer. Each RH instance ID in the list is a null-terminated character string that designates one RH system and can be used as a parameter in a subsequent call to RhOpen to specify the RH system that the application wants to work with. RH Instance IDs are stored in the buffer sequentially, separated by one null character. Two consecutive null characters designate the end of the list.

An RH infrastructure that implements this function **shall** return the list of RH Instance IDs only for those RH Systems that it services.

If multiple RH infrastructures are present on the current host, an intermediate layer of functionality between the application and infrastructures **may** be defined, that implements this function. If this is the case, that intermediate layer **should** consolidate together the lists of RH Instance IDs returned by separate RH infrastructures and present the consolidated list to the application as the result of the call to RhEnumerateInstances. The intermediate layer **may** change the RH Instance IDs returned by

separate infrastructures, qualifying them with textual identifiers of the corresponding infrastructures.

6.3.2.2 RhOpen

Prototype:

```
HSI_STATUS RhOpen(
      IN     char *InstanceId OPTIONAL,
      OUT   RH_HANDLE *pHandle );
```

Arguments:

InstanceId an RH Instance ID that chooses a specific RH system instance in the case where the calling host is attached to more than one RH system. This is an implementation-defined string. This parameter can be omitted (specified as NULL). In that case, the caller will be using the RH system, selected by default (defined by the first RH Instance ID, returned by RhEnumerateInstances).

pHandle pointer to the variable that holds the connection handle to the infrastructure of type RH_HANDLE. This type is generally opaque, but is typedef'ed to the handle type for the target OS.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid or unrecognized RH Instance ID
HSI_STATUS_NOT_SUPPORTED if the specified RH system is not available
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function initializes the connection between the application program and the RH infrastructure. It should be called in the beginning to initialize communication between the application and the infrastructure. This function creates a handle to the RH system and returns it to the application program. This handle is to be used in subsequent requests.

The current host may be attached to several RH systems. In that case, the parameter Instance ID should be used to specify the RH system that the application wants to work with. Specifying NULL as the value of the parameter InstanceID chooses the default RH system. If the function RhEnumerateInstances is supported, the default RH system **shall** be the one designated by the first RH Instance ID in the list returned by RhEnumerateInstances.

An RH infrastructure implementing this function **shall** recognize RH Instance IDs only for those RH systems that it services.

If multiple RH infrastructures are present on the current host, an intermediate layer of functionality between the application and infrastructures **may** be defined, that implements this function. If this is the case, that intermediate layer **should** choose the RH infrastructure that provides the API services to the application, based on the value of the parameter InstanceID. When doing this, the intermediate layer **may** process the

InstanceID before passing it to the infrastructure (removing, for example, the textual identifier of the infrastructure).

6.3.2.3 RhClose

Prototype:

```
HSI_STATUS RhClose (
      IN     RH_HANDLE Handle );
```

Arguments:

Handle	the handle to the infrastructure obtained via RhOpen.

Return Value :

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function closes the connection between the application program and the RH infrastructure and destroys the handle. It should be called at the end to gracefully terminate the communication between the application and the infrastructure.

6.3.2.4 RhGetInstanceID

Prototype:

```
HSI_STATUS RhGeInstanceID(
      IN     RH_HANDLE Handle,
      OUT    char *pInstanceID,
      IN     uint32 InstanceIDLength,
      OUT    ULONG *pActualSize );
```

Arguments:

Handle	the handle of the current session
pInstanceID	pointer to the character buffer where the RH Instance ID associated with the given handle is stored as a null-terminated character string
InstanceIDLength	the size of the buffer; if this size is too small for the output, this function fails.
pActualSize	this variable receives the actual size of the returned Instance ID; in the case of the error code HSI_STATUS_INSUFFICIENT_BUFFER returned, this is the minimal required size of the buffer.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
HSI_STATUS_INSUFFICIENT_BUFFER returned if the buffer pInstanceID is too small to store the RH Instance ID
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure.

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function returns the RH Instance ID for the given session handle. This is a character string that identifies the specific RH system with which the application communicates via the RH API in the specified session. The format of this string is implementation-dependent.

If multiple RH infrastructures are present on the current host, an intermediate layer of functionality between the application and infrastructures **may** be defined, that implements this function. If this is the case, that intermediate layer **should** ensure that the value returned to the application can be used to get access to the same RH System via RhOpen (for example, the intermediate layer may prepend the string returned to the application by the textual identifier of the infrastructure).

6.3.3 Domain and Host Information API

6.3.3.1 RhGetDomainCount

Prototype:

```
HSI_STATUS RhGetDomainCount(
      IN     RH_HANDLE Handle,
      OUT    uint32 *pCount );
```

Arguments:

Handle	the handle of the current session
pCount	pointer to the variable that receives the current number of domains in the system

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function returns the number of domains in the RH system that can be owned by the hosts in the system.

6.3.3.2 RhGetDomainNumbers

Prototype:

```
HSI_STATUS RhGetDomainNumbers(
      IN     RH_HANDLE Handle,
      OUT    uint32 *pDomainNumbersArray,
      IN     uint32 ArraySize,
      OUT    uint32 *pActualSize );
```

Arguments:

Handle	the handle of the current session
pDomainNumbersArray	pointer to the array where the list of domain numbers is

placed

ArraySize the size (in items of type uint32) of the buffer initially provided for the array by the caller

pActualSize pointer to the variable where the actual number of items in the list is stored (even if the initial size is too small and the function returns the error HSI_STATUS_INSUFFICIENT_BUFFER).

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
HSI_STATUS_INSUFFICIENT_BUFFER returned if the buffer provided for the array by the caller is too small; in that case, the array isn't filled in but the location pointed by pActualSize is set to the correct value to assist the caller in subsequent buffer allocation.
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function retrieves the list of numbers of known domains that comprise the RH system. Each domain number is an arbitrary uint32 value.
Before the call, the caller should allocate a buffer that can accommodate a sufficient number of uint32 values, and pass its address in the pDomainNumbersArray parameter. The parameter ArraySize should be set equal to the size of the buffer in uint32 items. The domain count returned from "RhGetDomainCount" can be used as the value of this parameter. On return, the function populates the buffer with the array of domain numbers for all domains in the system, and places the actual number of returned domain numbers into the output parameter *pActualSize. If the specified ArraySize is too small, the function returns status HSI_STATUS_INSUFFICIENT_BUFFER, and doesn't populate the buffer, but still sets the parameter *pActualSize to the required size of the buffer.

6.3.3.3 RhGetDomainOwnership

Prototype:

```
HSI_STATUS RhGetDomainOwnership(
      IN     RH_HANDLE Handle,
      IN     uint32 Domain,
      OUT    uint32 *pOwningHost );
```

Arguments:

Handle the handle of the current session

Domain the domain number

pOwningHost pointer to the variable that stores the number of the host (if any) that owns this domain; value RH_NO_DESTINATION_HOST means "not owned by any host"

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function returns the current owning host for the specified domain.

6.3.3.4 RhGetDomainSlotPath

Prototype:

```
HSI_STATUS RhGetDomainSlotPath (
      IN    RH_HANDLE Handle,
      IN    uint32 Host,
      IN    uint32 Domain,
      OUT   uint16 *pRootBus,
      OUT   char *pOutSlotPath,
      IN    uint32 SlotPathLength,
      OUT   ULONG *pActualSize );
```

Arguments:

Handle	the handle of the current session
Host	the target host number
Domain	the domain number
pRootBus	pointer to the variable where the infrastructure stores the root bus number of the PCI tree of this domain. This value is 0 for the first or single PCI tree. For additional PCI trees, this value is implementation-dependent, but is guaranteed to be non-zero.
pOutSlotPath	pointer to the buffer where the slot path of the root bridge of the specified domain is written as a null-terminated string
SlotPathLength	the size of the buffer; if this size is too small for the output, this function fails. The size of the maximum possible output is 513 characters (for the longest slot path in the system with 256 buses plus the null termination character).
pActualSize	this variable receives the actual size of the returned slot path; in the case of HSI_STATUS_INSUFFICIENT_BUFFER, this is the minimum required size of the buffer.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle or the domain number is invalid

HSI_STATUS_INSUFFICIENT_BUFFER returned if SlotPathLength is too small

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function returns the slot path of the domain parent bridge for the specified domain with respect to the target host. This function is guaranteed to return successfully only if the target host owns the specified domain.

If the target host does not own the specified domain, the function fails, unless the infrastructure knows in advance what slot path the domain will have when owned by

the target host. This slot path must not be affected by any switchovers that may take place in the RH system before the target host actually acquires the specified domain. The slot path is stored as a null-terminated sequence of two-character groups. Each group describes one item of the slot path and represents the number (DeviceNumber * 8 + FunctionNumber) for the corresponding PCI-PCI bridge in hexadecimal. The two hexadecimal digits of this number are represented by two characters from the set '0'..'9', 'A'..'F'.

6.3.3.5 RhGetDomainSlotCount

Prototype:

```
HSI_STATUS RhGetDomainSlotCount(
      IN     RH_HANDLE Handle,
      IN     uint32 Domain,
      OUT    uint32 *pPhysSlotCount);
```

Arguments:

Handle	the handle of the current session
Domain	the domain number
pPhysSlotCount	pointer to the variable where the number of physical slots in this domain is placed

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function returns the number of physical slots in the specified domain. This number can be used to specify the size of the buffer for the physical slot numbers in a subsequent call to RhGetDomainSlots.

6.3.3.6 RhGetDomainSlots

Prototype:

```
HSI_STATUS RhGetDomainSlots(
      IN     RH_HANDLE Handle,
      IN     uint32 Domain,
      OUT   PHYSICAL_SLOT_ID *pSlotNumbersArray,
      IN     uint32 ArraySize,
      OUT    uint32 *pActualSize );
```

Arguments:

Handle	the handle of the current session.
Domain	the domain number
pSlotNumbersArray	pointer to the array where the list of slot numbers for the specified domain is placed
ArraySize	the size (in items of type PHYSICAL_SLOT_ID) of the buffer initially provided for the array by the caller
pActualSize	pointer to the variable where the actual number of items in the list

is stored (even if the initial size is too small and the function returns the error HSI_STATUS_INSUFFICIENT_BUFFER).

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
HSI_STATUS_INSUFFICIENT_BUFFER returned if the buffer provided for the array by the caller is too small; in that case, the array isn't filled in but the location pointed by pActualSize is set to correct value to assist the caller in subsequent buffer allocation.
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function retrieves the list of physical slot numbers for the specified domain. Each physical slot number is an arbitrary (but system-wide, unique) combination of ShelfID and SlotID values.
Before the call, the caller should allocate a buffer that can accommodate a sufficient number of PHYSICAL_SLOT_ID structures, and pass its address in the pSlotNumbersArray parameter. The parameter ArraySize should be set equal to the size of the buffer in PHYSICAL_SLOT_ID items.
The slot count returned from "RhGetDomainSlotCount" can be used as the value of this parameter.
On return, the function populates the buffer with the array of slot numbers for all slots in the domain, and places the actual number of returned slot numbers into the output parameter *pActualSize. If the specified ArraySize is too small, the function returns status HSI_STATUS_INSUFFICIENT_BUFFER, and doesn't populate the buffer, but still sets the parameter *pActualSize to the required size of the buffer.

6.3.3.7 RhGetSlotDomain

Prototype:

```
HSI_STATUS RhGetSlotDomain(
      IN    RH_HANDLE Handle,
      IN    PHYSICAL_SLOT_ID PhysSlot,
      OUT   uint32 *pDomain);
```

Arguments:

Handle	the handle of the current session
PhysSlot	the physical slot number (represented as combination of Shelf ID and Slot ID)
pDomain	pointer to the variable where the number of the domain is placed

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
Other implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

Get the domain that owns the specified slot. This function is used to retrieve the number of the domain to which the specified physical slot currently belongs.
The physical slot is represented by its Shelf ID and the Slot ID inside the shelf.

6.3.3.8 RhGetCurrentHostNumber

Prototype:

```
HSI_STATUS RhGetCurrentHostNumber(
      IN     RH_HANDLE Handle,
      OUT   uint32 *pHost);
```

Arguments:

Handle	the handle of the current session
pHost	pointer to the variable where the current host number is placed

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function returns the number of the current host in an RH system (that is, the host on which this function has been called).

6.3.3.9 RhGetHostCount

Prototype:

```
HSI_STATUS RhGetHostCount(
      IN     RH_HANDLE Handle,
      OUT    uint32 *pHostCount);
```

Arguments:

Handle	the handle of the current session
pHostCount	pointer to the variable where the host count is placed

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function gets the number of hosts in the system. This function can be used to obtain the total number of hosts in a RH system.

6.3.3.10 RhGetHostNumbers

Prototype:

```
HSI_STATUS RhGetHostNumbers(
      IN            RH_HANDLE Handle,
      OUT           uint32 *pHostNumbersArray,
      IN            uint32 ArraySize,
```

```
    OUT          uint32 *pActualSize );
```

Arguments:

Handle	the handle of the current session
pHostNumbersArray	pointer to the array where the list of host numbers is placed
ArraySize	the size (in items of type uint32) of the buffer initially provided for the array by the caller
pActualSize	pointer to the variable where the actual number of items in the list is stored (even if the initial size is too small and the function returns the error HSI_STATUS_INSUFFICIENT_BUFFER).

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle

HSI_STATUS_INSUFFICIENT_BUFFER returned if the buffer provided for the array by the caller is too small; in that case, the array isn't filled in but the location pointed by pActualSize is set to a correct value to assist the caller in subsequent buffer allocation.

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function retrieves the list of numbers of known hosts that comprise the RH system. Each host number is an arbitrary uint32 value.

Before the call, the caller should allocate a buffer that can accommodate a sufficient number of uint32 values, and pass its address in the pHostNumbersArray parameter. The parameter ArraySize should be set equal to the size of the buffer in uint32 items. The host count returned from "RhGetHostCount" can be used as the value of this parameter. On return, the function populates the buffer with the array of host numbers for all hosts in the system, and places the actual number of returned host numbers into the output parameter *pActualSize. If the specified ArraySize is too small, the function returns status HSI_STATUS_INSUFFICIENT_BUFFER, and doesn't populate the buffer, but still sets the parameter *pActualSize to the required size of the buffer.

6.3.3.11 RhGetHostName

Prototype:

```
HSI_STATUSRhGetHostName(
      IN     RH_HANDLE Handle,
      IN     uint32 Host,
      OUT    char *pOutHostName,
      IN     uint32 HostNameLength,
      OUT    ULONG *pActualSize );
```

Arguments:

Handle	the handle of the current session
Host	the host number
pOutHostName	pointer to the character buffer where the host name is stored as a null-terminated character string
HostNameLength	the size of the buffer; if this size is too small for the output,

this function fails.

pActualSize this variable receives the actual size of the returned host name; in the case of the error code HSI_STATUS_INSUFFICIENT_BUFFER returned, this is the minimal required size of the buffer.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle.

HSI_STATUS_INSUFFICIENT_BUFFER returned if the buffer OutHostName is too small to store the host name

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function returns the symbolic name for the specified host. This is some kind of a network name (for example, NETBIOS name or TCP/IP host name) that can be used to establish a network connection to that host. This allows the hosts to communicate with each other over the network.

6.3.3.12 RhSetHostAvailability

Prototype:

```
HSI_STATUS RhSetHostAvailability(
      IN    RH_HANDLE Handle,
      IN    uint32 Host,
      IN    BOOLEAN Available);
```

Arguments:

Handle the handle of the current session

Host the host number

Available the new availability status of the host. Setting this argument to FALSE means that the host is brought into “isolation mode” in which it cannot own domains and cannot accept new domains via switchover. The host should not have any owned domains when its availability status is set to FALSE.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle, or invalid target host number, or Available=FALSE and the target host owns domains

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function changes the availability status of the target host for the RH infrastructure. Setting this status to FALSE brings the host into “isolation mode” in which the

host cannot own domains and cannot participate in domain switchovers. For such a host, the function RhGetHostAvailability returns FALSE. This mode can be used for configuration purposes, for example, to update system software on the host. Setting the status to TRUE brings the host back from the isolation mode to the state in which it can own and acquire domains.

If the parameter Available is FALSE, the target host must not own any domains when this function is called.

6.3.3.13 RhGetHostAvailability

Prototype:

```
HSI_STATUS RhGetHostAvailability(
      IN     RH_HANDLE Handle,
      IN     uint32 Host,
      OUT    BOOLEAN *pAvailable);
```

Arguments:

Handle	the handle of the current session
Host	the host number
pAvailable	pointer to the variable that receives a Boolean value: TRUE if the specified host is currently available and can own domains, FALSE otherwise (if the host is switched off or isolated from the rest of RH system).

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function can be used to determine whether the specified host in an RH system is up and running and can own domains. Returning *pAvailable=FALSE means that the specified host currently does not participate in RH activities and cannot own domains (for example, is switched off or runs in a special “isolation mode” or is unavailable due to some other reason).

The method of determining the availability status of the host is implementation-dependent. For example, the infrastructure may be able to determine that the host is physically present but does not have its inter-host communication queues initialized appropriately. In that case, it is considered not available. In other implementations, there may be a specific hardware register on the host that is visible to other hosts and has a bit that specifies host availability for RH activities (1=available, 0=not available). Other mechanisms are possible.

6.3.3.14 RhGetDomainAvailabilityToHost

Prototype:

```
HSI_STATUS RhGetDomainAvailabilityToHost(
      IN     RH_HANDLE Handle,
      IN     uint32 Host,
      IN     uint32 Domain,
      OUT   BOOLEAN *pAvailable);
```

Arguments:

Handle	the handle of the current session
Host	the host number
Domain	the domain number
pAvailable	pointer to the variable that receives a Boolean value: TRUE if the specified host can own the specified domain, FALSE otherwise.

Return Value :

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function is used in asymmetric RHost systems where some domains can be owned by some hosts but not by other hosts (for example, due to architectural constraints). This function returns a Boolean value (via pAvailable) that indicates whether the specified host can own the specified domain.

6.3.4 Slot Information API

6.3.4.1 RhGetPhysicalSlotInformation

Prototype:

```
HSI_STATUS RhGetPhysicalSlotInformation(
      IN     RH_HANDLE Handle,
      IN    PHYSICAL_SLOT_ID PhysSlot,
      OUT    RH_SLOT_DESCRIPTOR *pInfoBuffer,
      IN     uint32 InfoBufferSize,
      OUT    uint32 *pActualSize );
```

Arguments:

Handle	the handle of the current session
PhysSlot	obtains information for given physical slot number
pInfoBuffer	pointer to the buffer where the information is placed
InfoBufferSize	he size (in bytes) of the buffer initially provided for the array by the caller
pActualSize	pointer to the variable where the required size of the buffer is stored (even if the initial size is too small and the function returns the error HSI_STATUS_INSUFFICIENT_BUFFER).

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
HSI_NO_SUCH_DEVICE if the specified slot is empty

HSI_STATUS_INSUFFICIENT_BUFFER returned if the information buffer provided by the caller is too small; in that case, the buffer isn't filled in but the location pointed by pActualSize is set to a correct value to assist the caller in subsequent buffer allocation.
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function retrieves information about the device in the specified physical slot. If the device consists of several PCI functions, several information records are placed in the buffer, one for each PCI function. The following information is provided in each record, all of type RH_SLOT_DESCRIPTOR:
Size. This is the size of a particular RH_SLOT_DESCRIPTOR value including the variable-length SlotPath field.
Device addressing attributes:
PhysicalSlot. The number of the physical slot in the format (shelf-ID, physical-slot-ID); the device described by this descriptor resides in this slot.
PhysSlotDepth. The number of bridging levels between this device and the physical slot; this value is 0 for this call (since this call returns information about devices directly placed in the physical slots).
OwningHost. The number of the host that currently owns the domain this device belongs to.
RootBusNumber. The PCI bus number of the root bus of the PCI hierarchy the device resides in; is 0 for single-root PCI hierarchies. This value is 16 bit to accommodate possible future extensions to PCI that allow more than 256 PCI buses in the system.
SlotPath. The slot path from the root bus to the device. The slot path is stored as a null-terminated sequence of two-character groups. Each group describes one item of the slot path and represents the number (DeviceNumber * 8 + FunctionNumber) for the corresponding PCI-PCI bridge in hexadecimal. The two hexadecimal digits of this number are represented by two characters from the set '0'..'9', 'A'..'F'.
BusNumber. The bus number for the device. This value is 16 bit to accommodate possible future extensions to PCI that allow more than 256 PCI buses in the system.
DeviceNumber. The device number for the device.
FunctionNumber. The function number for the device.

Device configuration attributes (all based on PCI configuration space attributes):
VendorID/DeviceID/RevisionID
Identifies the manufacturer of the device that provides the PCI interface for the slot, the specific device product among those made by that manufacturer, and the revision level of that device.
SubsystemVendorID/SubsystemID
Identifies the manufacturer of the board and the specific board product among those made by that manufacturer.
BaseClass/SubClass/ProgIF
Identifies the type of device and its programming interface.

HeaderType
Identifies the layout of the second part of the pre-defined header of the device that provides the PCI interface for the slot (for example, 0 for conventional PCI device, 1 for PCI-PCI bridge).
The field *NeedsReset* indicates whether this device in its current state needs to be reset if switchover takes place. The value RESET_NOT_REQUIRED in this field means one of the following things:
• The device is already prepared for switchover.
• The device is not in use.
• The driver for the device is switchover-aware and is able to correctly bring it into a safe state after the switchover.
The value UNKNOWN means that the infrastructure does not know whether or not the device needs reset.

6.3.4.2 RhGetSlotChildInformation

Prototype:

```
HSI_STATUS RhGetSlotChildInformation(
      IN    RH_HANDLE Handle,
      IN    PHYSICAL_SLOT_ID PhysSlot,
      IN    char *pSlotPath,
      OUT   RH_SLOT_DESCRIPTOR *pInfoBuffer,
      IN    uint32 InfoBufferSize,
      OUT   uint32 *pActualSize );
```

Arguments:

Handle	the handle of the current session
PhysSlot	the physical slot number below which the devices in question are nested
pSlotPath	the slot path to the parent bridge for the devices
pInfoBuffer	pointer to the buffer where the information about devices is placed
InfoBufferSize	the size (in bytes) of the buffer initially provided for the array by the caller
pActualSize	pointer to the variable where the required size of the buffer is stored (even if the initial size is too small and the function returns the error HSI_STATUS_INSUFFICIENT_BUFFER).

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
HSI_NO_SUCH_DEVICE if there are no child devices below the specified bridge
HSI_STATUS_INSUFFICIENT_BUFFER returned if the information buffer provided by the caller is too small; in that case, the buffer isn't filled in but the location pointed by pActualSize is set to a correct value to assist the caller in subsequent buffer allocation.
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function retrieves information about child devices below the specified bridge that occupies the specified physical slot or is nested below it. The bridge is specified by the input parameter SlotPath.

The following information is provided for each device as an RH_SLOT_DESCRIPTOR structure:

Size. This is the size of a particular RH_SLOT_DESCRIPTOR value including the variable-length SlotPath field.

Device addressing attributes:

PhysicalSlot. The number of the physical slot in the format (shelf-ID, physical-slot-ID); the device described by this descriptor is nested below this slot.

PhysSlotDepth. The number of bridging levels between this device and the physical slot.

OwningHost. The number of the host that currently owns the domain this device belongs to.

RootBusNumber. The PCI bus number of the root bus of the PCI hierarchy the device resides in; is 0 for single-root PCI hierarchies. This value is 16 bit to accommodate possible future extensions to PCI that allow more than 256 PCI buses in the system.

SlotPath. The slot path from the root bus to the nested device. The slot path is stored as a null-terminated sequence of two-character groups. Each group describes one item of the slot path and represents the number (DeviceNumber * 8 + FunctionNumber) for the corresponding PCI-PCI bridge in hexadecimal. The two hexadecimal digits of this number are represented by two characters from the set '0'..'9', 'A'..'F'.

*BusNumber.*The bus number for the device. This value is 16 bit to accommodate possible future extensions to PCI that allow more than 256 PCI buses in the system.

DeviceNumber. The device number for the device.

FunctionNumber. The function number for the device.

Device configuration attributes (all based on PCI configuration space attributes of a PCI device nested within the slot):

VendorID/DeviceID/RevisionID

Identifies the manufacturer of the device that provides a nested PCI interface within the slot, the specific device product among those made by that manufacturer, and the revision level of that device.

SubsystemVendorID/SubsystemID

Identifies the manufacturer of a subsystem nested within the slot (say, a PMC module) and the specific subsystem product among those made by that manufacturer.

BaseClass/SubClass/ProgIF

Identifies the type of nested PCI device and its programming interface.

HeaderType

Identifies the layout of the second part of the pre-defined header of the nested PCI device (for example, 0 for a conventional PCI device, 1 for a PCI-PCI bridge.

The field *NeedsReset* indicates whether this device in its current state needs to be reset if switchover takes place. The value RESET_NOT_REQUIRED in this field means one of the following things:
• The device is already prepared for switchover.
• The device is not in use.
• The driver for the device is switchover-aware and is able to correctly bring it into a safe state after the switchover.
The value UNKNOWN means that the infrastructure does not know whether the device needs reset or not.This function can be used to enumerate devices nested below physical slots if a PCI-PCI bridge occupies the physical slot. To get information about all devices at the next nesting level, this function should be called with the physical slot number and slot path to the immediate parent bridge. This slot path is taken from the slot information structure for the immediate parent. To enumerate devices immediately nested below the bridge in the physical slot, the caller should pass the slot path from the slot information structure obtained via RhGetPhysicalSlotInformation.
The returned information is represented by the array of structures of variable length. Each structure describes one device located immediately below the parent PCI-PCI bridge. The total length of the array is returned in the location pointed by pActualSize. If some of the information structures identify corresponding devices as PCI-PCI bridges, the caller can go deeper and enumerate the PCI devices below that bridge using this function.

6.3.5 Switchover API

6.3.5.1 Switchover Scenarios and Theory of Operation

6.3.5.1.1 Fully Cooperative Switchover

In the cooperative switchover scenario, before giving up control over a domain, the owning host first prepares the PCI devices on this domain for switchover by gracefully shutting down operation on them and stopping the device drivers working with these devices. This operation is called software disconnection. This step is taken to ensure that the PCI devices appear to the new owner in a known state and that no transactions in progress are lost.
The exact meaning of software disconnection depends on the devices in the domain and their drivers. For device drivers that are not switchover-aware, software disconnection means shutdown of the corresponding devices and removal of all their software representations (device objects and so forth). Switchover-aware drivers may use “warmer” methods of preparation for switchover, keeping the device active to some degree during the switchover but preventing it from doing any damage to the new owning host immediately after the switchover (for example, preventing outstanding DMA transactions from this device to the host during the switchover).
Cooperative switchover can be initiated either by the owning host (in which case it voluntarily gives up control of this domain), or by the new owner of the domain, or by some third-party host.

In the last two cases, an inter-host communication channel is used to request the owning host to initiate software disconnection. In all cases, software disconnection is initiated by calling the RhPrepareForSwitchover function.
Once started, software disconnection can be rejected by software (if, for example, a PCI device in the domain performs an important operation that cannot be interrupted). Software disconnection can also be left pending for a long time (for example, awaiting completion of an important transaction). The function RhPrepareForSwitchover is asynchronous and does not wait for completion of software disconnection. The current software connection state, associated with the domain, can be used to track the progress of the software disconnection operation.
The initial software connection state of the domain is INACTIVE. For a domain in the normal state, the software connection state is CONNECTED. When software disconnection is initiated for a domain, the corresponding state becomes DISCONNECTING and stays DISCONNECTING while software disconnection is pending for the domain. When software disconnection completes successfully, the state goes to DISCONNECTED. If software disconnection is terminated unsuccessfully, the state goes back to CONNECTED. Software connection is the inverse action to software disconnection: it starts the drivers for PCI devices in the domain and resumes normal operation. When initiated for a domain in the DISCONNECTED state, it brings the domain to CONNECTED state through the intermediate CONNECTING state. Software connection can be used to cancel the effect of software disconnection for a domain during switchover preparation. For example, suppose that two domains should be switched over simultaneously in an atomic transaction; software disconnection succeeded for the first domain but was rejected for the second domain. As a result, the switchover is not possible and the first domain should be brought back into operation by software connection.
The same states apply to separate slots in the domain. They can be retrieved on a per-slot basis by separate polling functions or the caller can subscribe for asynchronous notifications about slot state changes. This makes it possible to invoke partially cooperative switchovers, in which the switchover is initiated when software disconnection is complete for some (more important) devices in the domain but not yet for other (less important) devices. These last devices should be reset during or immediately after the switchover to prevent possible damage to the new owning host.
The requesting host may specify a timeout for software disconnection. This value, expressed in milliseconds, serves as an indication to the owning host of the time interval during which the software disconnection should be completed. The requesting host indicates that after the expiration of the timeout it intends to either abandon the switchover or perform forced switchover.
After the software disconnection of the relevant domains is complete, switchover is initiated to change ownership of the domains. To trigger the switchover, the RhPerformSwitchover function should be called.
After the switchover, software connection is automatically initiated for the relevant domains on the receiving hosts. It is not necessary to call any functions after the switchover to software connect the received domains.

6.3.5.1.2 Partially Cooperative Switchover

With this type of the switchover, software disconnection takes place for some but not all of the devices in the domain. It may be considered that some devices need to be prepared for switchover while other devices may be switched over without preparation.
Another possible scenario is that some devices are considered “more important” and the others “less important”. The switchover is initiated as soon as software disconnection completes for “more important” devices, without waiting for completion of preparation for “less important” devices.
In all these cases, at the moment of switchover some devices are prepared for switchover, while other devices are not and may need to be brought into a known initial state after the switchover.
After the switchover, software connection is automatically initiated for the relevant domains on the receiving hosts; so it is not necessary to call any functions after the switchover to software connect the received domains.

6.3.5.1.3 Forced Switchover

In the forced switchover scenario, the domains are not software disconnected before the switchover, so device operation is not quiesced and for the device drivers and other software on the resigning host the PCI devices physically disappear, possibly in the middle of transactions. PCI devices are generally in an unknown state after the switchover. However, if the parameter Reset is used in the RhPerformSwitchover function, the PCI buses of the domain are reset, which brings the devices into the known initial state on the new owner host. Hence, forced switchover is potentially destructive for the owning host and should be used with care.
To perform forced switchover, it is sufficient to call the RhPerformSwitchover function. Forced switchover can be initiated either by the owning host (in which case it voluntarily gives up control of this domain), or by the new owner of the domain, or by some third-party host. In the last case, an inter-host communication channel may be needed to request one of the immediately participating hosts to perform the switchover.
After the switchover, software connection is automatically initiated for the relevant domains on the receiving hosts.

6.3.5.1.4 Hostile Switchover

Even in the case of a forced switchover request, there may be a possibility for the owning host to intercept the hardware switchover request and prevent it via hostile actions with respect to the destination host (for example, powering it off). An additional parameter (“Hostile”) to the RhPerformSwitchover function can be used to perform unconditional (hostile) switchover without any possibility for the owning host to prevent it.
After the switchover, software connection is automatically initiated for the relevant domains on the receiving hosts.

6.3.5.1.5 Hardware-Initiated Switchover

This type of switchover is initiated by hardware in the case of a hardware-initiated alarm (for example, a watchdog timer expiration) on the owning host. The new owning hosts for domains in this case are specified in advance via RhSetHwDestinationHost function. The parameter Reset in this function controls whether the PCI buses of the domain are reset after the switchover. If this parameter is TRUE, the PCI buses of

the domain are reset after the hardware-initiated switchover, which brings the devices into a known initial state on the new owning host.
The RhSetHwDestinationHost function can be called either on the owning host or on some third-party host. In the last case, an inter-host communication channel may be needed to request the owning host to register the destination host in hardware.
During hardware-initiated switchover, device operation is not quiesced and for the device drivers and other software on the resigning host the PCI devices disappear, possibly in the middle of transactions. However, this is not very important for this type of switchover, since the usual reason for switchover in this case is a malfunction of the owning host that requires some corrective actions, possibly including host reset.
After the switchover, software connection is automatically initiated for the relevant domains on the receiving hosts.

6.3.5.2 RhPrepareForSwitchover

Prototype:

```
HSI_STATUS RhPrepareForSwitchover(
      IN     RH_HANDLE Handle,
      IN    uint32 *pDomains,
      IN     uint32 DomainCount,
      IN     uint32 DestinationHost,
      IN     uint32 Timeout,
      IN     BOOLEAN Persist );
```

Arguments:

Handle	the handle of the current session
pDomains	pointer to the array of numbers of the domains to disconnect; all domains must be owned by the same host
DomainCount	the number of elements in the array of domain numbers
DestinationHost	the number of the destination host for the intended switchover of the specified domains; value RH_NO_DESTINATION_HOST meaning that no host owns the domains.
Timeout	the time interval (in milliseconds) that the requestor agrees to wait for the completion of disconnection. After this time expires, the requestor either forces the switchover or abandons it. This parameter is advisory and can be ignored by the target host. The special value 0 means that the requestor does not impose any time constraints to the software disconnection.
Persist	this parameter specifies what should be done in the case that software disconnection is not immediately possible for some slots. TRUE means that the target host should continue repeating attempts to software disconnect offending devices until software disconnection succeeds for all devices or the software disconnection request is cancelled by the requestor. FALSE means that the software disconnection of all requested domains should fail in that case and devices that have been

software disconnected already should be reconnected.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist

HSI_STATUS_REQUEST_DENIED returned if the software disconnection request issued by the current host has been denied

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function requests a domain software disconnection on the owning host in preparation for a switchover. The exact meaning of software disconnection depends on the devices in the domains and their drivers. For the device drivers that are not switchover-aware, this means shutdown of the corresponding devices and removal of all devices' software representations (device objects and so forth). Switchover-aware drivers may use "warmer" methods for preparation for switchover.This function just initiates the software disconnection and does not wait for its completion. The function RhGetDomainSwConnectionStatus can be used to track the progress of the pending disconnection.

In the cooperative switchover scenario, the domains should be software disconnected before the switchover; this guarantees that the former owning host software does not crash because of devices unexpectedly disappearing and that device activity does not crash the newly owning host immediately after the switchover.

The function can be called on a host that does not own the specified domains; in that case, the request may be forwarded to the owning host via an applicable inter-host communication channel.

However, all specified domains must be owned by the same host.

The caller can specify how urgent the software disconnection request is by using the Timeout parameter. This value specifies the time interval (in milliseconds) during which the owning host should try to complete the software disconnection. The caller assumes that after this timeout expires:

• It stops waiting for the software disconnection to complete

• It either abandons the switchover attempt or initiates a forced switchover that may be partially cooperative if software disconnection succeeds for some device(s) by that time.

6.3.5.3 RhCancelPrepareForSwitchover

Prototype:

```
HSI_STATUS RhCancelPrepareForSwitchover(
      IN     RH_HANDLE Handle,
      IN     uint32 *pDomains,
      IN     uint32 DomainCount );
```

Arguments:

Handle the handle of the current session
pDomains pointer to the array of numbers of the domains to connect; all domains must be owned by the same host
DomainCount the number of elements in the array of domain numbers

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist
HSI_STATUS_REQUEST_DENIED returned if the software connection request issued by the current host has been denied
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function.

Synopsis:

This function requests domain software connection. It initiates software connection for the specified domains:
• Startup of all devices in the domain
• Creation of corresponding software representation for devices (device objects and so forth)
If software disconnection is in progress for this domain, this function cancels the software disconnection.
This function just initiates the software connection–it does not wait for its completion. The function RhGetDomainSwConnectionStatus can be used to poll the progress of the pending connection. Alternatively, the notification functions provide a callback-based notification approach. See Section 6.2.6, “Notification, Reporting and Alarms” on page 70 for more information on these functions.
In the cooperative switchover scenario, this function should be called for the domains that have been software disconnected if the switchover is being cancelled (for example, because another domain specified in the switchover request cannot be software disconnected).
The function can be called on a host that does not own the domain; in that case, the request may be forwarded to the owning host via an applicable inter-host communication channel. However, the same host must own all specified domains.

6.3.5.4 RhGetDomainSwConnectionStatus

Prototype:

```
HSI_STATUS RhGetDomainSwConnectionStatus(
      IN     RH_HANDLE Handle,
      IN     uint32 Domain,
      OUT    RH_DOMAIN_SWC_STATE *pState );
```

Arguments:

Handle the handle of the current session
Domain the number of the domain to query state
pState pointer to the variable that receives the state

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

Get domain software connection status. This function returns the current state of the specified domain with respect to software connection/disconnection. There exist two stable (DISCONNECTED, CONNECTED) and two transitional (DISCONNECTING, CONNECTING) states.

This function can be used during a cooperative switchover to track progress of a pending software connection or disconnection request.

The function can be called on a host that does not own the domain.

6.3.5.5 RhGetSlotSwConnectionStatus

Prototype:

```
HSI_STATUS RhGetSlotSwConnectionStatus(
      IN     RH_HANDLE Handle,
      IN     PHYSICAL_SLOT_ID Slot,
      OUT    RH_DOMAIN_SWC_STATE *pState );
```

Arguments:

Handle	the handle of the current session
Slot	the physical slot number to query state for
pState	pointer to the variable that receives the state

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified slot does not exist

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

Get physical slot software connection status. This function returns the current state of the specified slot with respect to software connection/disconnection. There exist two stable (DISCONNECTED, CONNECTED) and two transitional (DISCONNECTING, CONNECTING) states.

This function can be used during a cooperative switchover to track progress of a pending software connection or disconnection request on a per-slot basis.

The function can be called on a host that does not own the domain to which the slot belongs.

6.3.5.6 RhPerformSwitchover

Prototype:

```
HSI_STATUS RhPerformSwitchover(
      IN    RH_HANDLE Handle,
      IN    uint32 DestinationHost,
      IN    uint32 *pDomains,
      IN    uint32 DomainCount,
      IN    BOOLEAN Reset,
      IN    BOOLEAN Hostile );
```

Arguments:

Handle	the handle of the current session
DestinationHost	the number of the host that should own the domains after the switchover; value RH_NO_DESTINATION_HOST means "no host should own the specified domains"
pDomains	the array of domain numbers that should be taken over. Passing NULL as this parameter requests that all existing domains should be switched over to the destination host.
DomainCount	the number of items in the array pDomains.
Reset	if TRUE, the PCI buses of domains are reset after the switchover
Hostile	if TRUE, the switchover is performed in a hostile way (the owning host is not given any opportunity before the switchover to be notified and to prevent it).

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle or any of the specified domains do not exist, or wrong parameters are specified (for example, DomainCount=0 and Domains != NULL).

HSI_STATUS_REQUEST_DENIED the switchover request for the specified domains by the current host has been denied

HSI_STATUS_NOT_SUPPORTED returned if this function with the specified set of parameters is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function performs the switchover. It is called on a host that currently owns the specified domains or on some other host to request switchover of the specified domains to the destination host. If the parameter Reset is TRUE, the corresponding domains are initially reset after the switchover by the new owning host.

6.3.5.7 RhSetHwDestinationHost

Prototype:

```
HSI_STATUS RhSetHwDestinationHost(
      IN    RH_HANDLE Handle,
      IN    uint32 SourceHost,
      IN    uint32 *pDomains,
```

```
      IN     uint32 DomainCount,
      IN     uint32 DestinationHost,
      IN     BOOLEAN Reset );
```

Arguments:

Handle	the handle of the current session
SourceHost	the number of the host for which domain destination is specified
pDomains	the array of domain numbers identifying the group of domains that is passed to the specified destination host if the source host fails and hardware-initiated switchover takes place for it
DomainCount	the size of the array pDomains
*DestinationHos*t	the number of the host that owns the specified domains if the source host fails and hardware-initiated switchover takes place for it; value RH_NO_DESTINATION_HOST means “no host owns the domains”
Reset	if TRUE, the PCI buses of domains are reset after the switchover

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle or invalid parameters (wrong or non-existent host or domain numbers)

HSI_STATUS_REQUEST_DENIED the request for the specified domains by the current host has been denied

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function is used to specify the destination host that obtains specified domains if a hardware-initiated switchover occurs due to the failure of the source host. In the case of such failure, domains owned by that host should be transferred to some other host; this function specifies the destination host on a per-domain group basis.

If this function is not called before the hardware-initiated switchover actually takes place, the domain is either passed to some predefined host or left unattached to any host. This predefined arrangement is specified by some entity beyond the scope of this specification (like BIOS or hardware default). However, the function RhGetHwDestinationHost can be used to obtain this predefined arrangement, even if RhSetHwDestinationHost has not yet been called for this domain/host pair.

6.3.5.8 RhGetHwDestinationHost

Prototype:

```
HSI_STATUS RhGetHwDestinationHost(
      IN     RH_HANDLE Handle,
      IN     uint32 SourceHost,
      IN     uint32 Domain,
      OUT    uint32 *pDestinationHost );
```

Arguments:

Handle	the handle of the current session
SourceHost	the number of the source host
Domain	the domain number
pDestinationHost	pointer to the variable receives the number of the host that should own the specified domain if the source host fails and hardware-initiated switchover takes place for it

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function gets the destination host that owns the specified domain if a hardware-initiated switchover takes place due to the failure of the source host.

6.3.6 Notification, Reporting and Alarms

6.3.6.1 RhEnableDomainStateNotification

Prototype:

```
HSI_STATUS RhEnableDomainStateNotification(
      IN    RH_HANDLE Handle,
      IN    RH_DOMAIN_STATE_CALLBACK DomainCallback,
      IN    RH_SLOT_STATE_CALLBACK SlotCallback OPTIONAL,
      IN     void *pContext );
```

Arguments:

Handle	the handle of the current session
DomainCallback	pointer to the callback function that tracks state of the domain
SlotCallback	pointer to the optional callback function that tracks state of separate slots during software connection and disconnection.
pContext	an opaque context pointer; passed unchanged to the callback function.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function establishes a callback that is called when the software connection state of one of the domains changes. The callback function is called with the domain number and the new state as parameters. Another parameter, pContext, is passed unchanged from the function that establishes the callback to the callback itself and can be used to pass some context information.
Four additional parameters, “RequestingHost”, “DestinationHost”, “Timeout” and “Persist,” are passed to the domain state notification callback when software disconnection is requested for the domain and the domain state becomes DISCONNECTING. They are passed unchanged from the parameter list for the RhPrepareForSwitchover function.
Values of these parameters are not meaningful when the new domain state is different from DISCONNECTING.
The parameter SlotStateCallback, if specified as non-NULL, should be an address of the slot state change notification callback. This callback is called when the state of a specific slot in the domain changes and allows the caller to track software connection and disconnection on a per-slot basis.
This function can be used to get notification about the progress of a pending software connection or disconnection request during a cooperative switchover.
The function can be called on a host that does not own the specified domain.

6.3.6.2 RhEnableSwitchoverNotification

Prototype:

```
HSI_STATUS RhEnableSwitchoverNotification(
      IN     RH_HANDLE Handle,
      IN     RH_SWITCHOVER_CALLBACK Callback,
      IN     void *pContext,
      IN     BOOLEAN Systemwide);
```

Arguments:

Handle	the handle of the current session
Callback	pointer to the callback function
Context	an opaque context pointer; passed unchanged to the callback function.
Systemwide	a Boolean flag; if set to TRUE, notification happens for each switchover even if the current host is neither the source nor the destination of the switchover; if set to FALSE, the host is notified only of those switchovers in which it participates.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function establishes the callback that is called when any domain is switched over from one host to another. The callback function is called with the new owner host number and the domain number as parameters. Another parameter, pContext, is passed unchanged from the function that establishes the callback to the callback itself and can be used to pass some context information.
An application may subscribe for notifications about all domain switchovers in the system by setting parameter Systemwide to TRUE.

6.3.6.3 RhEnableSwitchoverRequestNotification

Prototype:

```
HSI_STATUS RhEnableSwitchoverNotification(
      IN     RH_HANDLE Handle,
      IN     RH_SWITCHOVER_REQUEST_CALLBACK Callback,
      IN     void *pContext );
```

Arguments:

Handle	the handle of the current session
Callback	pointer to the callback function
pContext	an opaque context pointer; passed unchanged to the callback function.

Return Value :

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist
HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure.
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function establishes the callback that is called when an attempt is made to take over any domain from the current host. The callback function is called with the requesting host number, new owning host number, and the domain number as parameters. Another parameter, pContext, is passed unchanged from the function that establishes the callback to the callback itself and can be used to pass some context information.
If the switchover request callback is called, the requested switchover isn't successfully completed in hardware until the callback returns. The callback can request the infrastructure to prevent the requested switchover from happening by returning FALSE. In that case, the infrastructure may perform hostile actions to the new owning host (for example, power it off).

6.3.6.4 RhEnableUnsafeSwitchoverNotification

Prototype:

```
HSI_STATUS RhEnableUnsafeSwitchoverNotification(
      IN     RH_HANDLE Handle,
      IN     RH_UNSAFE_SWITCHOVER_CALLBACK Callback,
```

```
        IN     void *pContext );
```

Arguments:

Handle the handle of the current session

Callback pointer to the callback function

pContext an opaque context pointer; passed unchanged to the callback function.

Return Value:

HSI_STATUS_SUCCESS returned in the case of success

HSI_STATUS_INVALID_PARAMETER invalid session handle or the specified domain does not exist

HSI_STATUS_NOT_SUPPORTED returned if this function is not supported by the infrastructure

Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function establishes the callback that is called when a new domain is acquired by the current host. In that case, some (or all) devices in the domain may be in an unsafe state. To prevent immediate corruption of the new owning host after the switchover, a bus lock is usually implemented in RH systems. This lock prevents outgoing transactions from the domain devices to the host and interrupts from the domain devices. However this lock should not be held for a long time, but should be cleared by software soon after the switchover to allow normal operation of domain devices.

The corresponding callback can be used to handle this situation. The callback is called with the bus lock held. Parameters to the callback include a list of entries identifying domain devices in unsafe states. These devices should be reset before the domain can be software connected and the device drivers can be started. However, reset may not be necessary for a specific device if it is known that this device is harmless for the system or the device driver can bring the device into a safe state before the bus lock is cleared.

If registered, the callback is called after a switchover even if no devices are considered unsafe by the RH infrastructure. In that case, the list of entries, passed as a parameter, is empty.

The callback has the following prototype:

```
typedef void (*RH_UNSAFE_SWITCHOVER_CALLBACK) (
      IN uint32 Domain,
      IN RH_SWITCHOVER_TYPE SwitchoverType,
      IN BOOLEAN SlotResetSupported,
      IN uint32 UnsafeSlotCount,
      IN OUT RH_SLOT_DESCRIPTOR *pUnsafeSlotDescriptors,
      IN void *pContext );
```

The callback has the following parameters:

Domain - the number of the domain that has been acquired by the current host

SwitchoverType the switchover type

SlotResetSupported the Boolean flag that indicates whether the infrastructure supports per-slot resets on the domain

UnsafeSlotCount the number of descriptors for unsafe slots provided with the call
pUnsafeSlotDescriptors the array of descriptors, each of which describes one slot that contains a device in unsafe state
pContext the opaque context pointer passed unchanged from RhEnableUnsafeSwitchoverNotification
Each descriptor describes a device that is directly installed in a physical slot or nested below a physical slot in the PCI hierarchy (if the physical slot is occupied by a PCI-PCI bridge device), and has the following fields:
Size this is the size of the structure including the variable-length SlotPath field. To get to the next structure in the array, the caller should add this value to the address of the current structure.
Device addressing attributes:
PhysicalSlot the number of the physical slot in the format (shelf-ID, physical-slot-ID); the device described by this descriptor resides in this slot or below this slot PhysSlot-*Depth* The number of bridging levels between this device and the physical slot; if the device occupies the physical slot, this value is 0, otherwise it indicates is the depth of the device below the physical slot
OwningHost is set to the number of the current host
RootBusNumber the PCI bus number of the root bus of the PCI hierarchy the device resides in; is 0 for single-root PCI hierarchies. This value is 16 bit to accommodate possible future extensions to PCI that allow more than 256 PCI buses in the system
SlotPath the slot path from the root bus to the device; represented as a null-terminated character string
BusNumber the bus number for the device. This value is 16 bit to accommodate possible future extensions to PCI that allow more than 256 PCI buses in the system
DeviceNumber the device number for the device
FunctionNumber the function number for the device
Device configuration attributes:
Attributes from *VendorID* to *HeaderType* represent the PCI configuration space attributes of the device with the same names.
The field *NeedsReset* has a special meaning. It is set to RESET_REQUIRED or UNKNOWN before the callback is called. The callback should set this field to RESET_NOT_REQUIRED or RESET_REQUIRED on return.
The callback should set this field to RESET_NOT_REQUIRED if it considers that no reset is necessary for this device before releasing the bus lock (for example, if the device can be set to a safe state by the device driver or is in a safe state already).
The callback should set this field to RESET_REQUIRED if it considers that the reset is necessary for the device.
No descriptors are submitted for empty slots or for the slots occupied by devices that the infrastructure considers safe for the host.
If a PCI-PCI bridge occupies some physical slot, and some devices below this bridge are in unsafe state, both descriptors for the bridge and for the devices below it in unsafe state are present. In the array, the descriptor for the bridge precedes descriptors for the devices below it.

The actions of the infrastructure after the callback returns are specified by the following rules:
• If the parameter SlotResetSupported = FALSE (the infrastructure does not support per-slot resets), and at least one descriptor has NeedsReset = RESET_REQUIRED, the whole domain is reset before releasing the bus lock.
• Otherwise, for each physical slot in the domain, if SlotResetSupported = TRUE, and there is a descriptor for the given physical slot in the array with NeedsReset = RESET_REQUIRED, this physical slot is reset.
• Otherwise, if there is a PCI-PCI bridge device in the given physical slot, and there is at least one descriptor in the array for a device below this bridge (or for this bridge itself) with NeedsReset = RESET_REQUIRED, this physical slot is reset.
As a consequence, there is no reset if the callback clears NeedsReset in all descriptors submitted to it.

6.3.6.5 RhDisableNotification

Prototype:

```
HSI_STATUS RhDisableNotification(
      IN     RH_HANDLE Handle
      IN     RH_NOTIFICATION_TYPE NotificationType );
```

Arguments:

Handle	the handle of the current session
NotificationType	this enumeration specifies the type of notifications to disable

Return Value:

HSI_STATUS_SUCCESS returned in the case of success
HSI_STATUS_INVALID_PARAMETER invalid session handle
Other, implementation-defined HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function disables the notification callback that has been previously established via one of the RhEnableNotification functions. The specific type of notifications to disable is specified by the parameter NotificationType.

7. Slot Control API (ZT5524 specific feature)

7.1 HsiOpenSlotControl

Prototype:

```
HSI_STATUS HsiOpenSlotControl(
      OUT    HSI_SLOT_CONTROL_HANDLE *pHandle);
```

Arguments:

PHandle pointer to the location where this function places the session handle for the new session

Return Value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_NO_MEMORY returned if there is not enough memory to allocate the handle or other internal structures

HSI_STATUS_NO_SUCH_DEVICE returned if the Hot Swap Controller can't be found

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function is called by the client to open a logical session between the client and the HA Slot Control Driver. The session handle is returned to the client from this function. In all of the following calls related to the new session, the session handle shall be passed as one of the parameters.

This function shall be called before calling any other functions of this interface.

7.2 HsiCloseSlotControl

Prototype:

```
HSI_STATUS HsiCloseSlotControl(
      IN     HSI_SLOT_CONTROL_HANDLE Handle);
```

Arguments:

Handle The session handle to close.

Return Value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_INVALID_PARAMETER returned if the handle passed as a parameter is invalid

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function is called by a client to terminate a logical session between the client and the Hot Swap Controller driver, established by the call to HsiOpenSlotControl(). Upon return, the handle is no longer valid.

7.3 HsiGetSlotCount

Prototype:

```
HSI_STATUS HsiGetSlotCount(
      IN     HSI_SLOT_CONTROL_HANDLE Handle,
      OUT    UINT32 *pCount)
```

Arguments:

Handle	The handle of the current session
pCount	Pointer to the location where the number of physical slots is placed

Return Value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_INVALID_PARAMETER returned if the handle passed as a parameter is invalid

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

A client calls this function to retrieve the number of physical slots managed by the Hot Swap Controller. The physical slots are the same as geographical CompactPCI addresses and are numbered from 1 to this number, inclusive. However, the slot numbers need not be consecutive; there may be gaps in the sequence of physical slot numbers.

7.3.0.1 HsiGetBoardPresent

Prototype:

```
HSI_STATUS HsiGetBoardPresent(
      IN     HSI_SLOT_CONTROL_HANDLE Handle,
      IN     UINT32 Slot,
      OUT    BOOLEAN *pPresent )
```

Arguments:

Handle	The handle of the current session
Slot	The physical slot number
pPresent	Pointer to the location where the board presence flag is placed TRUE means a board is present in the slot; FALSE means no board is present in the slot

Return value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid

HSI_STATUS_NO_DATA_DETECTED returned if the board presence status cannot be currently determined (the slot is powered)

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function detects whether any board is present in the specified slot and returns the board presence status in the pPresent parameter. TRUE is returned if a board is present in the slot; FALSE is returned if no board is present in the slot.

Note: According to the Hot Swap Specification, if the slot power is on, it is not possible to detect whether the slot is occupied; this function returns status HSI_STATUS_NO_DATA_DETECTED in this case.

7.4 HsiGetBoardHealthy

Prototype:

```
HSI_STATUS HsiGetBoardHealthy(
      IN    HSI_SLOT_CONTROL_HANDLE Handle,
      IN    UINT32 Slot,
      OUT   BOOLEAN *pHealthy );
```

Arguments:

Handle	The handle of the current session.
Slot	The physical slot number
pHealthy	pointer to the location where the board health status is placed: TRUE means the board is present and healthy; FALSE means the board is not healthy

Return value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid

HSI_STATUS_NO_DATA_DETECTED returned if the board health status cannot be currently determined (the slot is not powered)

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function detects the health status of the board in the specified slot and returns it in the pHealthy parameter as a logical value. TRUE means the board is present and healthy; FALSE means the board is either not healthy or absent.

Note: The board health status cannot be determined if the slot power is off; this function returns status HSI_STATUS_NO_DATA_DETECTED in this case.

7.5 HsiGetSlotPower

Prototype:

```
HSI_STATUS HsiGetSlotPower(
      IN    HSI_SLOT_CONTROL_HANDLE Handle,
      IN    UINT32 Slot,
      OU    BOOLEAN *pPower );
```

Arguments:

Handle	The handle of the current session
Slot	The physical slot number
pPower	Pointer to the location where the slot power status is placed: TRUE

means the slot power is on; FALSE means the slot power is off

Return Value:

HSI_STATUS_SUCCESS if successful.
HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid
Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function detects the power status of the specified slot and returns it in the pPower parameter as a logical value: TRUE means the slot power is on; FALSE means the slot power is off.

7.6 HsiSetSlotPower

Prototype:

```
HSI_STATUS HsiSetSlotPower(
      IN     HSI_SLOT_CONTROL_HANDLE Handle,
      IN     UINT32 Slot,
      IN     BOOLEAN Power );
```

Arguments:

Handle	The handle of the current session
Slot	The physical slot number
Power	The new power state for the slot: TRUE means ON, FALSE means OFF

Return Value:

HSI_STATUS_SUCCESS if successful
HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid
Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function enables or disables power for the specified slot. The new power state of the slot is specified by the value of the parameter Power: TRUE means switch the power on; FALSE means switch the power off.

7.7 HsiGetSlotReset

Prototype:

```
HSI_STATUS HsiGetSlotReset(
      IN     HSI_SLOT_CONTROL_HANDLE Handle,
      IN     UINT32 Slot,
      OUT    BOOLEAN *pReset );
```

Arguments:

Handle	The handle of the current session
Slot	The physical slot number
pReset	Pointer to the location where the slot reset status is placed: TRUE means the slot is in the reset state; FALSE means the slot is not in the reset state

Return Value:

HSI_STATUS_SUCCESS if successful
HSI_STATUS_NOT_IMPLEMENTED returned if slot reset functionality is not implemented for the given platform
HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid
Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function detects the reset status of the specified slot and returns it in the pReset parameter as a logical value: TRUE means the slot is in the reset state; FALSE means the slot is not in the reset state.
This function is optional for the Hot Swap Controller; if it is not implemented by the hardware, HSI_STATUS_NOT_SUPPORTED is returned.

7.8 HsiSetSlotReset

Prototype:

```
HSI_STATUS HsiSetSlotReset(
      IN    HSI_SLOT_CONTROL_HANDLE Handle,
      IN    UINT32 Slot,
      IN    BOOLEAN Reset );
```

Arguments:

Handle	The handle of the current session
Slot	The physical slot number
Reset	The new reset state for the slot: TRUE means the slot is placed in the reset state; FALSE means the slot is taken out of the reset state

Return Value:

HSI_STATUS_SUCCESS if successful
HSI_STATUS_NOT_IMPLEMENTED returned if slot reset functionality is not implemented for the given platform
HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid
Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function sets the reset status for the specified slot. The reset status is specified by the Reset parameter: TRUE means assert reset to the slot; FALSE means de-assert reset to the slot.
Reset is considered a state rather than an action: that is, if a board is put into the reset state, it remains in the reset state until it is taken out of the reset state.
This function is optional for the Hot Swap Controller; if it is not implemented by the hardware, HSI_STATUS_NOT_SUPPORTED is returned.

7.9 HsiGetSlotM66Enable

Prototype:

```
HSI_API_DEF HSI_STATUS HsiGetSlotM66Enable(
      IN HSI_SLOT_CONTROL_HANDLE Handle,
      IN UINT32 Slot,
       OUT BOOLEAN *pM66Enable )
```

Arguments:

Handle	the handle of the current session
Slot	the physical slot number
pM66Enable	pointer to the location where the state of the M66EN line for the specified slot is placed (TRUE: 66 MHz operation is enabled for the slot; FALSE: 66 MHz operation is not enabled for the slot).

Return Value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_NOT_IMPLEMENTED returned if slot reset functionality is not implemented for the given platform.

HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function detects the state of the M66EN signal line for the specified slot (reflecting whether 66 MHz operation is enabled for the specified slot) and returns it in the pM66Enable parameter as a logical value: TRUE means that the signal is asserted (66 MHz operation is enabled for the slot); FALSE means that the signal is deasserted (66 MHz operation is not enabled for the slot).

This functionality is optional for the Hot Swap Controller; if it is not supported by the hardware, HSI_STATUS_NOT_SUPPORTED is returned.

7.10 HsiSetSlotM66Enable

Prototype:

```
HSI_API_DEF HSI_STATUS HsiSetSlotM66Enable(
    IN HSI_SLOT_CONTROL_HANDLE Handle,
    IN UINT32 Slot,
    IN BOOLEAN pM66Enable )
```

Arguments:

Handle	The handle of the current session.
Slot	The physical slot number.
M66Enable	The Boolean parameter that controls the state of the M66EN line for the specified slot (TRUE: M66EN is not driven for the slot by the Hot Swap Controller; FALSE: M66EN is driven low for the slot by the Hot Swap Controller, disabling 66 MHz operation).

Return Value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_NOT_IMPLEMENTED returned if slot reset functionality is not implemented for the given platform

HSI_STATUS_INVALID_PARAMETER returned if the physical slot number does not correspond to any actual slot or if the handle is invalid

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function controls the state of the M66EN signal line for the specified slot (reflecting whether or not 66 MHz operation is enabled for the specified slot), depending on the value of the parameter M66Enable. M66Enable = TRUE means that the signal line is not driven by the Hot Swap Controller (potentially enabling 66 MHz operation for the slot); M66Enable = FALSE means that the signal is driven low by the Hot Swap Controller (disabling 66 MHz operation for the slot).

7.11 HsiSetSlotEventCallback

Prototype:

```
HSI_STATUS HsiSetSlotEventCallback(
      IN     HSI_SLOT_CONTROL_HANDLE Handle,
      IN     HSI_SLOT_EVENT_CALLBACK Callback,
      IN     void *pContext )
```

Arguments:

Handle	The handle of the current session
Callback	Address of the callback function that is called in the case of a Hot Swap Control event. Pass NULL to cancel the callback registration.
pContext	Opaque context pointer. This value is passed unchanged to the callback function.

Return Value:

HSI_STATUS_SUCCESS if successful

HSI_STATUS_INVALID_PARAMETER returned if the arguments or handle is invalid

HSI_STATUS_NOT_SUPPORTED returned if slot event functionality is not implemented for the given platform

Other HSI_STATUS values returned if other errors occurred during execution of this function

Synopsis:

This function registers or unregisters a client callback function that is called by the HA Slot Control Driver in the case of one of the following events:

• State of one of the slots changes: a board is inserted or extracted, board health state changes, etc.

• Hardware error is detected in the Hot Swap Controller.

To register the callback, the client should call this function with a valid, non-zero callback address and an opaque context pointer. To unregister the callback, the client should call this function with NULL as the callback address; the context pointer is ignored in that case and may be any value.

The callback function has the following prototype:

```
VOID (*HSI_SLOT_EVENT_CALLBACK)(
      IN     void *pContext,
```

```
  IN     BOOLEAN HscError,
  IN     HSI_SLOT_EVENT_INFO *pSlotInfo );
```

The arguments have the following semantics:

pContext Opaque context pointer. This is the same value that was originally passed to HsiSetSlotEventCallback().

HscError The value TRUE indicates that a hardware error has been detected in the Hot Swap Controller, and FALSE indicates a state change in one of the slots.

pSlotInfo If HscError is FALSE, this argument is the pointer to the structure that contains the slot number and the new state of the slot that has changed its state. If HscError is TRUE, the value of this argument is reserved and undefined.

The slot event information structure is defined as follows.

```
typedef struct
HSI_SLOT_EVENT_INFO_STRUCT
{
      UINT32 SlotNumber;
      BOOLEAN Present;
      BOOLEAN Powered;
      BOOLEAN Healthy;
      BOOLEAN InReset;
} HSI_SLOT_EVENT_INFO;
```

with the fields specified as:

SlotNumber the number of the slot that has changed its state

Present the board presence status for the slot

Powered the power status for the slot

Healthy the health status for the board in the slot

InReset the reset status of the slot

Note: If Powered is TRUE, the value for Present is not valid, and that if Powered is FALSE, the value for Healthy is not valid.

This function shall be implemented as part of the HA Slot Control Interface on platforms where the Hot Swap Controller can automatically detect and signal the occurrence of slot status changes.

8. IPMI API (ZT5524 specific feature)

8.1 imbOpenDriver

Prototype:

```
int imbOpenDriver(void)
```

Parameters:

None

Returns:

Int - 0 for Fail and 1 for Success, sets hDevice

Description:

Establish a link to the IMB driver.

8.2 imbCloseDriver

Prototype:

```
void imbCloseDriver()
```

Parameters:

None

Returns:

None

Description:

Close a link to the IMB driver.

8.3 imbDeviceIoControl

Prototype:

```
static BOOL imbDeviceIoControl(
      HANDLE dummy_hDevice,
      DWORD dwIoControlCode,
      LPVOID lpvInBuffer,
      DWORD cbInBuffer,
      LPVOID lpvOutBuffer,
      DWORD cbOutBuffer,
      LPDWORD lpcbBytesReturned,
      LPOVERLAPPED lpoOverlapped )
```

Parameters:

dummy_hDevice	handle of device
dwIoControlCode	control code of operation to perform
*lpvInBuffe*r	address of buffer for input data
cbInBuffer	size of input buffer
lpvOutBuffer	address of output buffer
cbOutBuffer	size of output buffer
lpcbBytesReturned	address of actual bytes of output
lpoOverlapped	address of overlapped struct

Returns:

BOOL - FALSE for fail and TRUE for success. Same as standard NTOS call as it also sets Ntstatus.status.

Description:

Simulate NT imbDeviceIoControl using Unix calls and structures

8.4 imbSendTimedI2cRequest

Prototype:

```
ACCESN_STATUS imbSendTimedI2cRequest (
      I2CREQUESTDATA *pI2CReq,
      Int timeOut,
      BYTE   * pRespData,
      int * pRespDataLen,
      BYTE * pCompCode
)
```

Parameters:

pI2Creq	I^2C request
timeOut	how long to wait, mSec units
respDataPtr	where to put response data
respDataLen	size of response buffer/size of returned data
completionCode	request status from xMC

Returns:

ACCESN_STATUS - ACCESN_OK else error status code

Description:

Sends a request to an I^2C device

8.5 imbSendIpmiRequest

Prototype:

```
ACCESN_STATUS imbSendIpmiRequest (
      IMBPREQUESTDATA *pImbReq,
      BYTE * pRespData,
      int * pRespDataLen,
      BYTE * pCompCode,
      BOOL bWaitForResponse
)
```

Parameters:

pImbReq	request info and data
pRespData	where to put response data
pRespDataLen	how much response data there is
pCompCode	request status from destination controller
bWaitForResponse	Wait for a response

Returns:

ACCESN_STATUS ACCESN_OK else error status code

Description:

Sends a request to an I^2C device

8.6 imbGetAsyncMessage

Prototype:

```
ACCESN_STATUS imbGetAsyncMessage (
      ImbRespPacket *pMsg,
      DWORD *pMsgLen,
      ImbAsyncSeq *pSeqNo
)
```

Parameters:

pMsg	response packet
pMsgLen	IN - length of buffer, OUT - msg len
pSeqNo	previously returned sequence number (or ASYNC_SEQ_START)

Returns:

ACCESN_STATUS - ACCESN_OK else error status code

Description:

This function gets the next available async message with a message ID greater than SeqNo. The message looks like an IMB packet and the length and Sequence number are returned

8.7 imbIsAsyncMessageAvailable

Prototype:

```
ACCESN_STATUS imbIsAsyncMessageAvailable (
      unsigned int eventId)
```

Parameters:

eventId	EventID handle returned from imbRegisterForAsyncMsgNotification

Returns:

ACCESN_STATUS - ACCESN_OK when message available else error status code

Description:

This function waits for an Async Message to arrive in the queue. It blocks indefinitely until a message arrives.

8.8 imbRegisterForAsyncMsgNotification

Prototype:

```
ACCESN_STATUS imbRegisterForAsyncMsgNotification (
      unsigned int *handleId)
```

Parameters:

eventId	EventID handle returned once registered

Returns:

ACCESN_STATUS - ACCESN_OK else error status code

Description:

This function registers the calling application for Asynchronous notification when an SMS message is available with the IMB driver.

8.9 imbUnregisterForAsyncMsgNotification

Prototype:

```
ACCESN_STATUS imbUnregisterForAsyncMsgNotification (
      unsigned int *handleId)
```

Parameters:

eventId EventID handle to unregister

Returns:

ACCESN_STATUS - ACCESN_OK else error status code

Description:

This function unregisters the calling application for Asynchronous notification when an SMS message is available with the IMB driver.

8.10 imbGetLocalBmcAddr

Prototype:

```
ACCESN_STATUS imbGetLocalBmcAddr (BYTE *iBmcAddr)
```

Parameters:

iBmcAddr - OUT - value of current local BMC address

Returns:

ACCESN_STATUS - ACCESN_OK else error status code

Description:

This function gets the local xMC Address as determined by the driver init

8.11 imbSetLocalBmcAddr

Prototype:

```
ACCESN_STATUS imbSetLocalBmcAddr (BYTE iBmcAddr)
```

Parameters:

iBmcAddr IN - value of current local xMC address

Returns:

ACCESN_STATUS - ACCESN_OK else error status code

Description:

This function is used when the xMC does not support the PICMG 2.16 GetAddress-Info IPMI command to force the local xMC Address.

8.12 imbGetIpmiVersion

Prototype:

```
BYTE imbGetIpmiVersion()
```

Parameters:

None

Returns:

BYTE - Current determined IPMI version

Description:

This function is returns the current IPMI version as either IPMI_09_VERSION, IPMI_10_VERSION, or IPMI_15_VERSION

9. Appendix A: Sample Programs

The HSK provides two sample programs which utilize the API. We believe that these examples will aid the developer during implementation of their applications. The sample applications can be found in the target directory under the “samples” folder (Default: C:\Program Files\PPS\Hot Swap Kit\Samples).

9.1 Hsctl

9.1.1 Summary

This is the source code for the hsctl utility – the command-line utility, which allows you to interactively invoke some of the HSK control functions. See the Chapter entitled “Using Hot Swap Kit Support Utilities“ for detailed description of hsctl functionality. Most of the hsctl commands are specific to CPX8000 systems and requires Pigeon Point Hot Swap Controller Kit to function properly.

The main() function parses the command and invokes the appropriate command handler for parsing of subsequent arguments and for command execution.

Commands correspond usually to specific HSK API calls. The typical command handler verifies the command parameters, establishes connection with the appropriate component of the HSK API, invokes the corresponding API function and closes the connection.

9.1.2 Building the Sample

To build the sample, Visual C++ 6 IDE (msdev.exe) is needed. Use the following command for the debug build:

msdev hsctl.dsp /make “hsctl – Win32 Debug”

Use the following command for the release build:

msdev hsctl.dsp /make “hsctl – Win32 Release”

9.1.3 File Manifest

hsctl.cpp

The source file, implementing all functionality

hsctl.dsp

The Visual C++ 6 project file.

9.2 Regnot

9.2.1 Summary

This is a Windows application that monitors hot swap related events, generated by the HSK, and allows you to obtain information about the target system via the HSK API.

This application demonstrates the usage of the Hot Swap-related part of the HSK API and will run on any supported platform.
The information about the events is shown in the main window in text format (two lines of text per event).
The menu items in the “Edit” menu allow you to do the following:

- Get the list of all known CompactPCI devices in the system;
- Get the list of physical slots and their slot paths;
- Get slot information by slot path;
- Get the device names by slot path;
- Simulate extraction request for a CompactPCI device;
- Simulate insertion request for a CompactPCI device.

9.2.2 Building the Sample

To build the sample, Visual C++ 6 IDE (msdev.exe) is needed. Use the following command for the debug build:

msdev regnot.dsp /make “REGNOT – Win32 Debug”

Use the following command for the release build:

msdev regnot.dsp /make “REGNOT – Win32 Release”

9.2.3 File Manifest

regnot.clw

The class wizard file for the application

DeviceNameDialog.h

Definition file of the “Get device names by slot path” dialog

DeviceNameDialog.cpp

Implementation file of the “Get device names by slot path” dialog

GetSlotInfoDialog.h

Definition file of the “Get slot information” dialog

GetSlotInfoDialog.cpp

Implementation file of the “Get slot information” dialog

InsExtDialog.h

Definition file of the “Request Insertion”, “Request extraction” dialogs

InsExtDialog.cpp

Implementation file of the “Request Insertion”, “Request extraction” dialogs

MainFrm.h

Main frame window, definition, generated by MFC

MainFrm.cpp

Main frame window, implementation, generated by MFC

Regnot.h

Application, definition, generated by MFC

Regnot.cpp

Application, implementation, generated by MFC

Regnot.dsp

Visual C++ 6 project file

RegnotDoc.h

MFC document definition file, generated by MFC

RegnotDoc.cpp

MFC document implementation file, generated by MFC

RegnotView.h

The MFC view definition file

RegnotView.cpp

The MFC view implementation file; command handlers are implemented here

Resource.h

The resource definition file

StdAfx.h

Generated by Visual C++

StdAfx.cpp

Generated by Visual C++

Res\Toolbar.bmp

Application resource file

Res\Regnot.ico

Application resource file

Res\RegnotDoc.ico

Application resource file

Res\Regnot.rc2

Application resource file

10. Appendix B: Adding PnP Awareness to Windows 2000/XP Software

CompactPCI Hot Swap fits well into the Plug and Play model of Windows 2000/XP, expanding its applicability to the PCI bus. That's why Plug and Play capable functional device drivers automatically support Hot Swap, and Plug and Play aware applications become automatically Hot Swap aware. To achieve Plug and Play awareness, it is sufficient for the application to use the generic mechanisms provided by Windows 2000/XP for device drivers and applications to deal with the dynamic nature of a device population. These mechanisms are described below.

10.1 Device Interfaces

Any driver of a physical, logical, or virtual device to which user-mode code can direct I/O requests must supply some sort of name for its user-mode clients. Using the name, a user-mode application (or other system component) identifies the device from which it is requesting I/O.

In previous versions of the operating system, drivers named their device objects and then set up symbolic links in the registry between these names and a user-visible Win32 logical name.

Windows 2000/XP and WDM drivers, however, do not name device objects. Instead, these drivers register and enable a device interface for each device object to which user-mode I/O requests might be sent. A *device interface* is a way of exporting device and driver functionality to other system components, including other drivers as well as user-mode applications.

Device interfaces are grouped into classes. Each device interface class is associated with a GUID. When a driver registers a device interface, the I/O Manager associates its device interface class GUID with a symbolic link name. The link name is stored in the registry and persists across system boots. An application that uses the interface can query for its symbolic link name and save this to use as a target for I/O requests.

The I/O manager function **IoRegisterDeviceInterface** is used for device interface registration. This registration is performed only once. The function **IoSetDeviceInterfaceState** is used later for enabling or disabling the interface.

To get access to a device by the device interface class GUID, the application should call the **SetupDiGetClassDevs** or **SetupDiGetClassDevsEx** function to obtain a list of all devices in a specified device class. This list is known as a *device information set*. Then the application should call the **SetupDiEnumDeviceInterfaces** function to enumerate all devices of the specified class that export the interface. For each device, the function **SetupDiGetDeviceInterfaceDetail** should be called, that returns the symbolic link as one of

the output parameters. This symbolic link can be used as a file name to open a handle to the device.

A functional driver supporting Hot Swap should register its device interface, enable it when the device becomes available to clients (e.g. when device is started) and disable it when the device becomes unavailable (e.g. when it is being removed).
Windows 2000/XP provides notifications to drivers and applications about device interfaces being enabled or disabled. Thus, the clients are notified not when a specific *device* appears on or disappears from the PCI bus, but rather when the corresponding *device interface* becomes available or unavailable.

10.2 Notifications for Plug and Play Aware Applications

The PnP Manager provides a mechanism for drivers and applications to be notified when certain events occur on a specific device or on the system in general. These events include arrival and departure of device interfaces of the specified class, and device removal requests. Using notifications, applications can monitor changes in the device population and even control the device removal attempts.

For user-mode applications, notifications are delivered via the WM_DEVICECHANGE Windows message or via the extended service control request (for services only). User-mode applications subscribe for notifications by calling **RegisterDeviceNotifications** and unsubscribe by calling **UnregisterDeviceNotification**. The event type is designated by the value of the first parameter (wParam) of the WM_DEVICECHANGE message.
There are two main categories of notifications that can be used by Hot Swap aware applications:

- Interface arrival and departure notifications.
- Target device change notifications.

10.2.1 Interface Arrival and Departure Notifications

When an application registers for this category of events on a device interface, the PnP Manager notifies the application about the following events:

DBT_DEVICEARRIVAL
: Indicates that a device interface of the specified class has been enabled. For example, a new device of the specified class has appeared on the PCI bus and the appropriate driver has been successfully loaded and started. One of parameters to this notification is the symbolic link that can be used to open a handle to the device.

DBT_DEVICEREMOVECOMPLETE
: Indicates that a device interface of the specified class has been disabled.

10.2.2 Target Device Change Notifications

Unlike registering for device interface changes, which can be considered a “passive” interest in the interface, registering for target device changes indicates an “active” interest in a device.

To register notifications of this type, the application should possess a valid handle to the target. Usually the application obtains the handle by opening the symbolic link, which is provided as part of the notification structure for the device interface arrival event.
When an application registers for this category of events on a device, the PnP Manager notifies it when the following events occur on the device:

DBT_DEVICEQUERYREMOVE

Indicates that the PnP Manager is about to remove the drivers for the device. Several actions can cause this event, including:

- an operator requested the device removal by depressing the board handle;
- a user has requested to remove the specified device from the machine;
- a user has issued an update-driver request for the device.

This notification requests the application to either approve or veto the impending remove operation. If the client approves the removal, it should close all handles to the device. If handles remain open on the device, the PnP Manager cannot remove the device, and the PnP Manager aborts query-remove processing. However, the client should remain registered for future target device change notifications. This is important because the impending remove operation might be cancelled. The application returns TRUE from the WM_DEVICECHANGE message handler to allow removal or BROADCAST_QUERY_DENY to deny the removal.

DBT_DEVICEREMOVEPENDING

This event is broadcasted by the system as a last warning before a device is removed. At this point, the application cannot cancel the removal, so if it is using the device it must prepare for its removal to prevent loss of data.

DBT_DEVICEREMOVECOMPLETE

This event is sent as a target device change event when a surprise removal of the device takes place and notifies the application that the device has physically disappeared. The application should perform necessary surprise-remove processing, such as closing any handles to the device.

DBT_DEVICEQUERYREMOVEFAILED

Indicates that an impending remove operation on the specified device has been cancelled. The application can continue working with the device and may open a new handle to the device at this point, if necessary.

DBT_CUSTOMEVENT

This event represents a custom event, generated by the respective functional driver. Its semantics are functional driver-specific.

10.3 Configuration Management API

The configuration management functions provide access to the tree of device nodes, which represents the devices currently existing in the system. An application can browse the tree, using functions **CM_Get_Child**, **CM_Get_Parent**, **CM_Get_Sibling**, or directly locate the device node by the device instance ID, using the function **CM_Locate_DevNode**.

Having obtained the device node handle, the application can query its status using the function **CM_Get_DevNode_Status**, and its instance ID, using the function **CM_Get_Device_ID**. Using the function **CM_Request_Device_Eject,** an application can request removal of the specific device.

Other functions, provided by the configuration management API allow the application to obtain various information about devices in the system, and manage device resources and logical configurations.

11. Appendix C: Troubleshooting

11.1 Problem 1

The board is unlocked but the blue LED does not appear or blinks for a long time.

This behavior means that the board is not available for extraction and is usually caused by the board extraction being vetoed by some software component. This component may be a user-mode application or service or a kernel-mode driver. Usually (but not necessarily) this component will be working with the device and not willing to relinquish control over the device at that time.

If the extraction request is vetoed, the Hot Swap System Driver repeatedly retries the extraction requests, with certain time interval (which is configurable). The operator can either wait for one of these retries to succeed, or cancel the extraction request by locking the board.

Blinking of the blue LED will happen for boards, based on the Intel 21554 chip. This blinking happens due to the specifics of Hot Swap implementation in this chip and means that the extraction request is pending for the board, but the board is not available for extraction yet.

11.2 Problem 2

A device is inserted but the software connection does not happen for it.

There may be several reasons for this problem to happen. First, verify that the board is visible on the PCI bus. Run the Slot Information utility and see what it shows for the physical slot the board is in.

1. *The utility indicates that the slot is empty.* This means that the board is not visible on the PCI bus at all. You may need to extract the board and insert it again. It may also be possible, that the board just does not work in this chassis.

   On the CPX8000, the reason may be that the current host does not own the corresponding domain. You may need to update the BIOS or if you would like to support the HA facilities within the CPX8000 you may want to get the Pigeon Point Hot Swap Controller Kit product.

   Alternatively, the board may require the H.110 telephony bus on the backplane. The CompactPCI Computer Telephony specification requires that such a board not power up if the H.110 bus is not present. You may not be able to use the board in your system.
2. *The utility indicates that the slot is occupied, and correctly shows the PCI identification information for the board.* In that case, run the Device Manager utility supplied

with Windows 2000/XP (Control Panel | System | Hardware | Device Manager). Locate your device in the list of devices. The best way to do this is organize device by connection: View | Devices by connection and find your device in the PCI bus tree under Standard PC | PCI bus. It should have an exclamation mark in its icon. If it does not have an exclamation mark, then the device is software connected and working properly.

3. *The device has an exclamation mark in its icon*. Double-click the icon. The property sheet will open that will show the device status information in one of its fields.

4. *The device status indicates insufficient resources*. You may need to increase the I/O and memory windows in its parent bridge. Edit the relevant sizes with the Configuration Editor and restart the system, or, if the parent bridge is hot swappable, extract and reinsert it.

5. *The system cannot find a driver for the device.* In that case, reinstall the driver manually by clicking the button 'Reinstall Driver…' in the same property sheet.

6. Other reasons may be shown, which usually means that the appropriate driver was found but refused to start the device. The reasons for such behavior are driver-specific. For example, some Ethernet drivers require the network cable to be plugged in when the device is being software connected.

11.3 Problem 3 (Switchover questions)

Q. Switchover / takeover was performed successfully but the software connection does not happen for all devices in the systems on the new host:

A: Check whether the PCI resource assignment is the the same for both hosts

Q. Switchover failed with an error message.

A1. If the hostile switchover fails then it may be a hardware issue. The peer is not ready for operations usually.

A2. In case of fully cooperative or partially cooperative Switchovers the most common issue is the system rejects the device software disconnection and thus prevents the switchover from occurring.

Q. After a hostile switchover the host crashes.

A1. If the newly backup host crashes then it might crash due to the aggressive nature of operation - the physical devices are absent already but software is not informed still.

A2. If the newly active host crashes then it may be due to a hardware issue - in the most common case device asserts the interrupt line and the system can't deassert it because the corresponding drivers were not started.

Q. What to do after an unsuccessful fully or partially cooperative switchovers?

A. - First, it is necessary to check the domains attributes: their owners and their software connection status.

- It is recommended to perform an additional switchover for the "not switched yet" domain. Unless there is a hardware problems, hostile switchovers should always work.

- For "not software connected" domains it is recommended to perform the RhCancelPrepareForSwitchover API function to force domain software connection. If all devices refuse to go into "connection" state then it is likely to be a PCI resource allocation issue. As a last resort, it is possible to put the whole domain in reset, issue RhPrepareForSwitchover function, then deassert the reset line and issue RhCancelPrepareForSwitchover.

Q. Device disappears from the Device Manger after driver installation.

A. The most probable reason is that the driver information file specifies the device's Class GUID which is not defined in the system yet. Under such conditions the Device Manager refuses to show devices with undefined Class GUID. This is a software installation issue. Usually this bug appears when there was a software upgrade of specific components in the system: instead of removing and installing the whole software package – only drivers were updated. To resolve this issue it is necessary to add the Class GUID definition to the system registry manually or completely remove and then install the whole software package for the troublesome device.

Q. After switchover peripheral boards except one are not detected on the new active side but after the host is rebooted, all boards are visible.

A. The most probable reason is there may be a malfunction in the hardware of the peripheral board. It looks like the board starts responding to PCI requests which are issued to other devices thereby effectively concealing it from other devices on that PCI bus. A host reboot causes the board to do a cold reset and thus it reinitializes the board logic properly. To resolve this issue it is necessary to extract the faulty board from the chassis.