

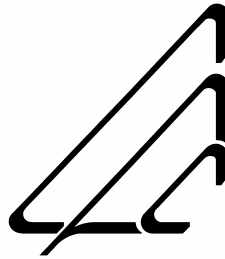


QuickKit  
Telephony®  
PCI  
DPT4, DPT5 & DPT6  
Programmer  
Reference  
Manual

Version 1.5.5  
For Linux, Solaris, and  
Windows 2000 & XP

**Email:** [support@cacdsp.com](mailto:support@cacdsp.com)

## DPT Programmer Reference



© 2002 - 2009 Communication Automation Corporation  
West Chester, PA (USA)

### License Agreement

The international copyright laws that pertain to computer software and hardware protect this Software/Hardware. It is illegal to duplicate the design and implementation of the Hardware and/or to make copies of the Software except as provided in this license agreement. It is illegal to give copies of CAC Software to another person, or to duplicate the Software by any other means, including electronic transmission, except as provided in this license agreement. CAC Software and Hardware contains trade secrets and, to protect them, you may not decompose, reverse engineer, disassemble, or otherwise reduce the applicable object code or binary portions of the Software or Hardware to human perceivable form.

Our software is a product of Communication Automation Corporation (CAC) and is licensed for unrestricted use WITH CAC HARDWARE PRODUCTS ONLY. CAC software may be reproduced and used by the customer only if this legend is included on all distribution media and this legend is included as a part of the software comments, whether the CAC software is used in whole or in part.

Users may copy or modify CAC software without royalty, but are not authorized to license, sub-license, or distribute this copied or modified CAC software to any other person or organization except as part of a hardware product or software developed by the user that incorporates CAC hardware products. You are permitted, however, to freely distribute your own derived software that communicates to the CAC boards through these software drivers and libraries, free of any royalty to CAC.

### Warranty

Communication Automation Corporation reserves the right to make changes to these products, including any software and/or hardware described herein, without notice. No warranty of merchantability or fitness for a particular purpose is expressed or implied. CAC shall not be held liable for incidental or consequential damages in connection with, or arising out of, the use of this Software. CAC does not recommend the use of any of its products, Software or Hardware, for medical or life support applications wherein a failure or malfunction of the product may threaten life or cause injury and will not knowingly license or sell its products for either such use. No rights under any patent accompany the sale of any such products.

### Trademarks

QuicKit Telephony® and smPCI® are registered trademarks of Communication Automation Corporation.  
SPARC, SunOS and Solaris are registered trademarks of Sun Microsystems Computer Corporation and SunSoft.  
Unix is a registered trademark of Santa Cruz Operations.

Use of a term in this manual should not be regarded as affecting the validity of any trademark or service mark.

# TABLE OF CONTENTS

<b>1</b>	<b>NOMENCLATURE</b>	<b>1-1</b>
<b>2</b>	<b>OVERVIEW</b>	<b>2-1</b>
2.1	Channelization	2-1
2.2	Memory Mapped I/O	2-1
2.3	Resource Specific Functions	2-2
2.4	Utility Functions	2-2
<b>3</b>	<b>CHANNELIZATION</b>	<b>3-1</b>
3.1	Read Channels	3-1
3.2	Write Channels	3-2
3.3	Channel Loopback	3-3
<b>4</b>	<b>MEMORY MAPPED I/O</b>	<b>4-1</b>
<b>5</b>	<b>RESOURCE SPECIFIC FUNCTIONS</b>	<b>5-1</b>
5.1	Flash Memory	5-1
5.2	Serial Eerom	5-1
5.3	TDM Subsystem	5-1
5.4	Computer Telephony (CT) Bus Subsystem	5-1
<b>6</b>	<b>ADDRESS MAPS</b>	<b>6-1</b>
<b>7</b>	<b>VERSION CHECKING</b>	<b>7-1</b>
<b>8</b>	<b>SOFTWARE DEVELOPMENT</b>	<b>8-1</b>
8.1	Board insertion order makes a difference	8-1
8.2	UNIX (Solaris and Linux)	8-2
8.2.1	Unix Include Files	8-2
8.2.2	Unix Library	8-2
8.3	Windows 2000 and XP	8-3
8.3.1	Microsoft Visual Studio Projects	8-3

## DPT Programmer Reference Manual

8.3.2	Windows Include Files	8-3
8.3.3	Windows Library	8-3
8.3.4	GNU compatible import libraries are	8-4

## 9 API FUNCTIONS 9-1

<b>9.1</b>	<b>Function Descriptions</b>	<b>9-1</b>
9.1.1	dp_close_signal	9-1
9.1.2	dp_get_signal	9-2
9.1.3	dp_open_signal	9-3
9.1.4	dp_read_signal	9-4
9.1.5	dp_write_signal	9-5
9.1.6	flash_erase_block	9-6
9.1.7	flash_read_a16b	9-7
9.1.8	flash_read_i16b	9-8
9.1.9	flash_write_a16b	9-9
9.1.10	flash_write_i16b	9-10
9.1.11	htob_16	9-11
9.1.12	htob_32	9-12
9.1.13	htol_16	9-13
9.1.14	htol_32	9-14
9.1.15	mod_eerom_erase	9-15
9.1.16	mod_eerom_read	9-16
9.1.17	mod_eerom_write	9-17
9.1.18	pci_chan_autoidle	9-18
9.1.19	pci_chan_blocking	9-19
9.1.20	pci_chan_buf_usage	9-20
9.1.21	pci_chan_check_versions	9-21
9.1.22	pci_chan_forcexmt	9-22
9.1.23	pci_chan_get_idle	9-23
9.1.24	pci_chan_get_signal	9-24
9.1.25	pci_chan_set_idle	9-25
9.1.26	pci_chan_set_signal	9-26
9.1.27	pci_chan_source	9-27
9.1.28	pci_chan_waitsync	9-28
9.1.29	pci_check_versions	9-29
9.1.30	pci_clear_driver_versions	9-30
9.1.31	pciclose	9-31
9.1.32	pci_close_chan	9-32
9.1.33	pci_comet_log2phys	9-33
9.1.34	pci_comet_read	9-34
9.1.35	pci_comet_write	9-35
9.1.36	pci_ctrl	9-36
9.1.37	pci_dl_a8b	9-37
9.1.38	pci_dl_a16b	9-38
9.1.39	pci_dl_a32b	9-39
9.1.40	pci_dl_cfg8	9-40
9.1.41	pci_dl_cfg8le	9-41
9.1.42	pci_dl_cfg16	9-42
9.1.43	pci_dl_cfg16le	9-43
9.1.44	pci_dl_cfg32	9-44
9.1.45	pci_dl_i8b	9-45
9.1.46	pci_dl_i16b	9-46
9.1.47	pci_dl_i32b	9-47
9.1.48	pci_eerom_erase	9-48

## DPT Programmer Reference Manual

9.1.49	pci_eerom_read	9-49
9.1.50	pci_eerom_write	9-50
9.1.51	pci_flush_chan	9-51
9.1.52	pci_framer_get_idle	9-52
9.1.53	pci_framer_get_mode	9-53
9.1.54	pci_framer_set_idle	9-54
9.1.55	pci_framer_set_mode	9-55
9.1.56	pci_framer_tx_source	9-57
9.1.57	pci_get_dpbuffer	9-58
9.1.58	pci_get_info	9-59
9.1.59	pci_get_location	9-61
9.1.60	pci_get_version	9-62
9.1.61	pci_get_version_checking	9-63
9.1.62	pcihold	9-64
9.1.63	pci_hw_fast_clk	9-65
9.1.64	pci_hw_tdmdb_clk	9-66
9.1.65	pciopen	9-67
9.1.66	pciopenex	9-68
9.1.67	pci_open_chan	9-70
9.1.68	pci_open_chanex	9-73
9.1.69	pci_open_chan_rcv_host_start	9-74
9.1.70	pci_pad_chan	9-75
9.1.71	pci_peek8	9-76
9.1.72	pci_peek8a	9-77
9.1.73	pci_peek16	9-78
9.1.74	pci_peek16a	9-79
9.1.75	pci_peek32	9-80
9.1.76	pci_peek32a	9-81
9.1.77	pci_poke8	9-82
9.1.78	pci_poke8a	9-83
9.1.79	pci_poke16	9-84
9.1.80	pci_poke16a	9-85
9.1.81	pci_poke32	9-86
9.1.82	pci_poke32a	9-87
9.1.83	pci_read_chan	9-88
9.1.84	pci_reg_rmw	9-89
9.1.85	pcireset	9-90
9.1.86	pcirun	9-91
9.1.87	pci_run_chan	9-92
9.1.88	pci_set_dpbuffer	9-93
9.1.89	pci_sleep	9-94
9.1.90	pci_start_receive	9-95
9.1.91	pci_up_a8b	9-96
9.1.92	pci_up_a16b	9-97
9.1.93	pci_up_a32b	9-98
9.1.94	pci_up_cfg8	9-99
9.1.95	pci_up_cfg8le	9-100
9.1.96	pci_up_cfg16	9-101
9.1.97	pci_up_cfg16le	9-102
9.1.98	pci_up_cfg32	9-103
9.1.99	pci_up_i8b	9-104
9.1.100	pci_up_i16b	9-105
9.1.101	pci_up_i32b	9-106
9.1.102	pci_usleep	9-107
9.1.103	pci_version_checking	9-108
9.1.104	pci_write_chan	9-109

<b>10</b>	<b>EXPANSION AND TDM API FUNCTIONS</b>	<b>10-1</b>
<b>10.1</b>	<b>smPCI Module Functions</b>	<b>10-1</b>
10.1.1	pci_get_module_type	10-2
10.1.2	pci_load_code, pci_verify_code, pci_read_code	10-3
<b>10.2</b>	<b>TDM Control Functions</b>	<b>10-4</b>
10.2.1	pci_tdmclose	10-6
10.2.2	pci_tdmopen	10-7
10.2.3	pci_tdm_clock_cfg	10-8
10.2.4	pci_tdm_clrmap	10-10
10.2.5	pci_tdm_dst_add	10-11
10.2.6	pci_tdm_dst_del	10-12
10.2.7	pci_tdm_getmap	10-13
10.2.8	pci_tdm_multiframe	10-14
10.2.9	pci_tdm_run	10-15
10.2.10	pci_tdm_src_add	10-16
10.2.11	pci_tdm_src_del	10-17
10.2.12	pci_tdm_stop	10-18
10.2.13	pci_tdm_updatemap	10-19
<b>10.3</b>	<b>H.100 Control Functions</b>	<b>10-20</b>
10.3.1	h100_init	10-21
10.3.2	h100_master_pll	10-22
10.3.3	h100_master	10-23
10.3.4	h100_compat	10-24
10.3.5	h100_netref	10-25
10.3.6	h100_slave	10-26
10.3.7	h100_set_leds	10-27
10.3.8	h100_route_write	10-28
10.3.9	h100_route_source	10-29
10.3.10	h100_route_lcl	10-30
10.3.11	h100_par_write	10-31
10.3.12	h100_par_read	10-32
<b>11</b>	<b>SPECIAL SUPPORT FUNCTIONS</b>	<b>11-1</b>
<b>11.1</b>	<b>Physical to Virtual Memory Mapping</b>	<b>11-1</b>
11.1.1	clearPhyToVirCache	11-2
11.1.2	enablePhyToVirCache	11-3
<b>12</b>	<b>CONTACT INFORMATION</b>	<b>12-1</b>

## 1 Nomenclature

The CAC QuicKit Telephony® - PCI hardware and software library are referred to with the acronym DPT (Desktop PCI Telephony). This generally refers to the board types, DPT4, DPT5 and DPT6. Header files and executables for DPT products use the shortened prefix “dp” while the library is named in an OS dependent manner with the base name “dp” (i.e. libdp.a for UNIX systems).

In some cases, descriptions and files names may refer specifically to DPT4, for example the Windows installer or the Windows DLL file, named cacdpt4.dll. Such references generally pertain to DPT5 and DPT6 boards as well.

The library and expansion hardware are based in part on the existing CAC VME product line (V6M6, V6M6HS, etc) allowing the sharing of smPCI® (Small Mezzanine PCI) expansion modules between these product lines. To ease porting from VME based products to the DPT product line, the library functions use the same naming conventions as the V6M6 library. Thus, most functions user either “pci” or a resource type (“flash”, “dm12c549”, “dm5420”, etc) as a prefix.

This page left intentionally blank.

## 2 Overview

The CAC Desktop PCI Telephony (DPT) library consists of several families of functions. These include channelization, memory mapped I/O, and resource specific functions. These functions are provided to cover the needs of all application areas for the DPT boards – including DPT4, DPT5 and DPT6 boards.

### 2.1 Channelization

The channelization functions are used to provide the host access to E1/T1 data streams. These functions allow the host to combine a set of timeslots from an E1/T1 stream to create a channel. The channel can then be read from or written to as a single serial stream, independent of the number of slots comprising the channel. Support is provided for both blocking and non-blocking I/O. While there are certain requirements for buffer size and alignment to achieve optimal performance, the library supports arbitrary size transfers to arbitrarily aligned buffers if necessary.

In addition to the I/O functions, several functions are available for error detection and handling, changing the mode of an already open channel, controlling idle codes for a transmit channel, and access to signaling data.

### 2.2 Memory Mapped I/O

Memory mapped I/O allows the host direct access to all memories and control registers on a DPT board. This includes expansion modules on boards with expansion support. For boards without expansion support the memory mapped I/O functions are primarily for use by CAC for testing and configuration.

Any region of memory on a DPT board has up to four different addresses. The first (and most commonly used within user code) is the PAR address. This is the address an expansion module would use in its PCI Address Register to access the region. The use of this type of address is preferred in user code as it means that host and DSP code use the same address. The `pci_up_xxx` and `pci_dl_xxx` functions use PAR addresses.

The second type of address used is the Map Address. These are the addresses passed to the device driver to map device memory into the host address space (using the `mmap(2)` system call on UNIX platforms). This address is a function of the host PCI bus address used to address the region. These addresses are primarily for use by the library and CAC diagnostics and utilities through the `pci_peek` and `pci_poke` family of functions.

The third type of address used is the host address. This is the address in host memory mapped to the memory region. It is used in the library to transfer the requested data. These memory addresses should never be used directly.

The final type of address is the physical address. This is the address used on the memory bus on a DPT board. The address used by the embedded code differs from the physical address only in the upper bits.

## 2.3 Resource Specific Functions

A large family of resource specific functions is also provided. This includes functions for reading and programming the nonvolatile memories (a flash and up to three serial eeproms), configuring framer modes (when channelization is not used), managing the on-board TDM bus and the interboard H.100 bus, and configuring and managing expansion modules, such as the DM5420 DSP board. Each of these subfamilies will be documented in its own section.

## 2.4 Utility Functions

Several functions are provided to ease writing of robust, platform independent code. Currently these functions are limited to thread-safe sleep functions. In the future, they will be expanded to include thread creation and management functions, synchronization functions, and other functions required to make full use of the DPT library on all supported platforms. See the DPT Utility Manual for additional resources.

### 3 Channelization

The channelization functions allow the host to send and receive E1 or T1 data. A channel is formed by bundling the data from one or more timeslots on a single framer into a logical data stream, using dedicated hardware to reduce processor overhead. The channel can then be read from or written to in a manner similar to file I/O. A single channel is opened either for reading or for writing; bi-directional data transfer requires the use of two channels. All of the channelization functions require the embedded processor to be running. The processor can be started with either the `dprun` utility or the `pcirun()` function call.

A channel is opened with the `pci_open_chan()` function by specifying the board, framer, timeslots, framer mode, and channel mode. All timeslots within a channel must be on the same framer and the slot list must be in ascending order (i.e. you can form a channel from slots 1, 7, and 19 but not from 1, 19, and 7). Additionally, the slot must not be part of another channel in the same direction. If these requirements are met, the embedded processor will attempt to set the framer to the specified mode. The call will fail if the framer is already in use in a different mode. Otherwise, the channel is opened and ready for I/O. When a channel is no longer needed it should be closed with `pci_close_chan()`.

#### 3.1 Read Channels

After opening a channel for read, `pci_read_chan()` is used to receive data. If the buffer is aligned on an 8-byte boundary and the transfer length is a multiple of 256 bytes then the hardware will copy the data directly to the user's buffer. If the buffer or transfer length is not acceptable then data will be transferred through an intermediate buffer. This allows maximum performance when the buffers meet the hardware restrictions and maximum flexibility when arbitrary buffers must be used.

A set of secondary functions is provided for checking channel status. The `pci_flush_chan()` function discards any unread data from all buffers. This is potentially useful in error recovery. The `pci_chan_waitsync()` function can be used either to check for framer sync (with a timeout of zero) or to block until a channel is synchronized to the incoming data stream (with a non-zero timeout). The `pci_chan_buf_usage()` function reports the current buffer usage. The mode parameter allows the user to request information about the double buffer used by `pci_read_chan()` or the hardware buffer. The function can be used to determine the amount of data in the buffer, the amount of free space in the buffer, or the size of the buffer. The units parameter is used to request a value in units of bytes, frames, or microseconds.

The `pci_chan_set_signal()` function controls delivery of signals on asynchronous events including data available, loss of sync, overflow, and processor reset. The default behavior is to send a SIGKILL on processor reset and no signal on the other conditions. To check the current signal setting use `pci_chan_get_signal()`.

Because a single application can have multiple channels open the `pci_chan_rcv_signal()` function is provided for use in the signal handler. The signal handler should call this function once for each open channel. The function returns the set of pending signals for that channel.

A channel can operate in either blocking or non-blocking mode. The default mode is blocking. Non-blocking mode can be selected by passing the `CHAN_NBLOCK` flag to `pci_open_chan()`. For an existing channel `pci_chan_blocking()` can be used to change the mode.

When in blocking mode a read does not return until the requested number of bytes has been transferred or an error occurs. When in non-blocking mode a read will return only the data that is currently buffered. The return value from `pci_read_chan()` indicates the number of bytes transferred.

### 3.2 Write Channels

Write channels operate similar to read channels. Data is sent to the board with `pci_write_chan()`. Unlike `pci_read_chan()` there are no restrictions on buffer alignment or transfer length. Because there is never double buffering for write channels `pci_chan_buf_usage()` will always return zero for double buffer statistics. The asynchronous event functions `pci_chan_xxx_signal()` operate as described except that the loss of sync signal will not be sent to write channels and an overflow condition for a write channel should be interpreted as an underflow. Because it is always possible to send data `pci_chan_waitsync()` will always return sync immediately. The `pci_flush_chan()` function blocks until all buffered data has been sent. The `pci_open_chan()` function can be used to open a transmit channel in either blocking or non-blocking mode and `pci_chan_blocking()` can be used to change the mode.

A channel can operate in either blocking or non-blocking mode. The default behavior is blocking. Non-blocking mode can be selected by passing the `CHAN_NBLOCK` flag to `pci_open_chan()`. For an existing channel `pci_chan_blocking()` can be used to change the mode. The write will only block if there is not enough room to buffer the entire request. When in non-blocking mode a write the amount of data that could be copied to the buffer. The return value from `pci_write_chan()` indicates the number of bytes transferred.

Some applications have bursty data transmission with extended periods of idle being transmitted. Whenever the transmit buffer is empty the hardware will repeat an idle character on all timeslots of the channel. The default idle character is all ones (255 decimal, FF hex). This can be changed using the `pci_chan_set_idle()` function or the `pci_chan_autoidle()` function. To check the current idle character setting use the `pci_chan_get_idle()` function. The `pci_chan_autoidle()` function is used to enable and disable auto-fill mode which uses the last transmitted byte as the idle character. Note that after an underflow occurs the `pci_write_chan()` function will return with *errno* set to `E_OVERFLOW` (indicating the underflow). In this case the call to `pci_write_chan()` should be repeated. The error should not occur twice in a row.

Bursty data transmission also requires some other special considerations. One is that the DPT can only transmit whole frames of data and whole multiples of four bytes. The `pci_pad_chan()` function may be used to ensure that the amount of data in the transmit buffer meets both criteria. Another consideration is the transmit delay. Once the embedded code has started sending the idle code it will not resume transmission of new data until either one second of data is buffered or one second elapses after at least a frame's worth of transmit data is successfully written. This means that if a short message

is written to the transmit buffer, it will not normally be transmitted for 1 second. To avoid this delay use the `pci_chan_forcexmt()` function (available with version 1.4.1 or higher of the DPT API and embedded code)

When a write channel is closed any pending transmit data is discarded. To ensure that all data has been transmitted, call `pci_chan_flush()` before `pci_chan_close()`.

### 3.3 Channel Loopback

Applications that need to monitor and retransmit data can make use of the `pci_chan_source()` function. This function allows a receive channel to be set as the source for any number of transmit channels on the same board. Once the channel loopback is started the embedded processor will retransmit the received data without host intervention. The data may still be monitored by the host using `pci_read_chan()`.

## 4 Memory Mapped I/O

The memory mapped I/O functions serve two purposes.

- First, they allow CAC to test the hardware and to diagnose problems.
- Second, they provide generic access mechanisms to expansion modules on boards so equipped.

Before using any of the memory mapped I/O functions a resource must be opened. This is done with either the `pciopen()` or `pciopenex()` functions, both of which return a pointer to a `PCI_MOD` structure required for all other calls. The latter function allows the user to specify additional flags such as whether the resource should be opened exclusively or in a shared mode.

There are five resources available for memory mapped I/O. The ‘a’ and ‘b’ resources provide access to the corresponding expansion module site as well as global memory. The ‘g’ and ‘h’ resources provide access to both module sites and global memory. The ‘q’ resource allows access to the entire board, including baseboard registers. When the resource is no longer needed `pciclose()` should be used to close the device and free the `PCI_MOD` structure.

The family of `pci_up_xxx()` and `pci_dl_xxx()` functions provide access to an opened resource. Variants are provided for transferring 8, 16, and 32 bit items as individual values or as arrays. All functions in this family use the PAR address map. This means that the addresses used in these functions are the same addresses that would be used by a processor or DSP expansion module to access the same resource.

The `pci_up_cfgxxx()` and `pci_dl_cfgxxx()` functions provide access to PCI configuration space. The module resources can access the configuration space for that module and for the expansion bridge. The global memory resources can access both modules’ configuration space as well as the expansion bridge. The ‘q’ resource can access the host bridge, expansion bridge, and module configuration space. Please note that changing configuration space settings can cause the board or the entire system to stop responding.

The final set of functions is the `pci_peekxxx()` and `pci_pokexxx()` functions. These functions provide access to device control registers and other resources not available to devices using PAR addressing. The `pci_reg_rmw()` function uses locking to safely change individual bits within a register. The lock is system wide, including the embedded software. These functions are primarily for use by CAC for configuring and testing hardware.

This page left intentionally blank.

## 5 Resource Specific Functions

There are several specialized resources available on DPT devices. These resources include flash memory, serial eeproms, the TDM subsystem, and the H.100 Computer Telephony (CT) bus system. The DPT library provides functions for managing all of these resources.

### 5.1 Flash Memory

All DPT boards use nonvolatile flash memory to store firmware and Field Programmable Gate Array (FPGA) configuration data. Flash memory is organized into blocks. Before flash contents can be changed the appropriate block must be erased with `flash_erase_block()`. After erasure data can be written as 16-bit words using `flash_write_i16b()` or as arrays of 16-bit words using `flash_write_a16b()`. Flash data can be read using `flash_read_i16b()` and `flash_read_a16b()`.

The contents of the flash are essential to the proper functioning of the hardware. If the firmware is corrupted the board will have to be reprogrammed with *dpflashup* before it will be fully functional. If the FPGA configurations are corrupted the board may need to be returned to CAC for reprogramming.

### 5.2 Serial Eerom

All DPT boards and all expansion modules have a serial eeprom used to store serialization data and configuration options. The contents of an eeprom can be read with `pci_eerom_read()`, using a resource name, or `mod_eerom_read()`, using an already open module. Similarly an eeprom can be written with either `pci_eerom_write()` or `mod_eerom_write()`. Before a location is written it must be erased with either `pci_eerom_erase()` or `mod_eerom_erase()`. All of these functions operate on individual 16-bit words.

### 5.3 TDM Subsystem

DPT boards with expansion support have a Time Division Multiplexed serial data bus for transfer of telecom data. Any expansion module can act as a source and destination on the TDM bus, as can the framers and the MIPS processor. The Expansion and TDM API is described in section 10.

### 5.4 Computer Telephony (CT) Bus Subsystem

DPT boards expansion also have an H.100 compliant Computer Telephony (CT) bus interface. This allows data exchange between the TDM subsystems on multiple boards in the same chassis, including CT bus devices from other manufacturers. The CT bus API is currently in development and will be documented in a future release.

**This page left intentionally blank.**

## 6 Address Maps

This section refers programmers to the DPT4 and DPT5/DPT6 Hardware Manuals for documentation of address maps for all DPT devices. This information is primarily for use in systems with expansion hardware and will therefore be included in a future release.

This page left intentionally blank.

## 7 Version Checking

The library function `pci_check_versions` is provided to check the consistency of various versions. Separate version numbers are maintained for the library, driver and hardware logic files stored in flash memory on the board. A version number has three parts, major, minor and update.

Two version numbers are considered consistent if their major and minor version numbers match. The functions `pciopen` and `pci_open_chan` by default call `pci_check_version` to check the consistency of the procedure, driver and library version used to build the application.

The default action is to fail if all of the versions are not consistent. The library function `pci_version_checking` gives an application program the ability to change the default behavior of `pciopen` and `pci_open_chan` when versions are not consistent. The application can choose to ignore version mismatch errors or to just have them reported to `stderr`.

This page left intentionally blank.

## 8 Software Development

DPT software includes all of the libraries and include files required to develop applications for the DPT hardware. This distribution also contains the full source code for the diagnostics, utility programs, examples and drivers provided with the distribution.

### 8.1 Board insertion order makes a difference

#### **!!! An important note for software developers !!!**

There are major differences between Unix and Windows systems in the way in which the operating system recognizes boards inserted into the system backplane.

For Solaris systems, boards are recognized at the time of insertion. The boards are recognized by the operating system as a function of board ID and not physical location in the backplane. Note also that DPT5 and DPT6 boards use a PCI bridge device to interface with the host's PCI bus. Placing a DPT5 board in a location previously populated with a DPT4 board (or vice-versa) will appear as a new DPT board instance.

For Linux and Windows systems, board insertion order changes as additional boards are added to the system backplane. This means that Linux and Windows systems will dynamically reassign the board ID as new boards are added or removed from the system.

The first board inserted in a Linux or Windows system, regardless of physical location, is defined as board "0" (zero). If a second board is inserted, the boards are automatically re-numbered based on location.

For example, if the backplane numbering system is left to right, a first board inserted in the middle of the backplane is still board "0" (zero). If a second board is inserted to the left of the first board, the operating system will automatically re-number the boards with the leftmost board designated as board "0" and the other board as board "1". To further complicate matters, some backplanes number the slots from left to right, and some are numbered from right to left.

The `pci_get_location()` API library function can be used to determine in which PCI slot a specific board is located.

## 8.2 UNIX (Solaris and Linux)

Support for building Unix applications is provided for GNU gcc on both Solaris and Linux systems and Sun's compiler tool set Forte[tm] C++ (formerly called Sun Visual WorkShop[tm] C++) on Solaris systems. The installation procedure for Unix includes compiling the libraries, programs and the device driver to account for differences in hardware architecture and operating system version.

The configuration and make files used to build the installation use a set of automation tools from GNU called automake and autoconf. These tools were used to create the configure and make file(s) which are included in the distribution. These files can be used as templates for developers to create configuration and make files for their own applications. However, much simpler make files can be used to build applications particularly in the case where a single processor architecture and operating system are to be targeted. Much of the complexity in the supplied files results from using tools intended to support multiple architectures and operating systems.

### 8.2.1 Unix Include Files

Compilation commands should include two directories for include search paths: \$CAC/include and \$CAC/dpt. The gcc command line options for these include paths are:

```
-I$CAC/include -I$CAC/dpt
```

### 8.2.2 Unix Library

All DPT application programs need to include the libraries libdp.a and libcacipc.a. In addition applications using the HDLC API need libdphdlc.a. These libraries are installed in the directory \$CAC/lib. The gcc linker options to use these libraries are:

```
-L$CAC/lib -ldphdlc -lcacipc -ldp
```

## 8.3 Windows 2000 and XP

Support for building Windows 2000 and XP applications is provided for Microsoft Visual Studio and GNU gcc.

### 8.3.1 Microsoft Visual Studio Projects

A workspace file and set of projects for Microsoft Visual Studio are included in the <installation directory>\dpt\win2k\project directory. Projects are included for each of diagnostic, utility and example programs included in the release. These projects can be used as examples for developing projects for user written applications.

### 8.3.2 Windows Include Files

Two directories should be included in the compilers default search paths:

<installation directory>\include, and

<installation directory>\dpt.

To set up the include paths in a Microsoft Visual Studio project:

1. Select the *Settings* menu item in the *Project* menu.
2. Select the *All configuration* entry in the *Settings for:* drop down.
3. Select the *C/C++* tab in the dialog box and *Preprocessor* in the *Category* drop down.
4. Enter *c:\cac\include, c:\cac\dpt* in the *Additional include directories:* text box.

### 8.3.3 Windows Library

The API library for Windows 2000 and XP is supplied as a DLL. Two versions of the DLL are included in the release.

<installation directory>\dpt\bin\cacdpt4.dll is the standard (non-debugging version).

<installation directory>\dpt\bin\debug\cacdpt4.dll was built with debug information for the API functions included.

Corresponding Microsoft Visual Studio compatible import libraries are also included. These libraries need to be used when application programs are linked.

<installation directory>\dpt\lib\release\cacdpt4.lib is the standard version, and

<installation directory>\dpt\lib\debug\cacdpt4.lib is the debug version.

### 8.3.4 GNU compatible import libraries are

<installation directory>\dpt\lib\release\cacdpt4.libdpt4.a , and  
<installation directory>\dpt\lib\debug\cacdpt4.libdpt4.a

To include the DPT library in a project with Microsoft Visual Studio:

1. Select the *Settings* menu item in the *Project* menu.
2. Select the desired configuration with the *Settings for:* drop down.
3. Select the *Link* tab in the dialog box and *Input* in the *Category* drop down.
4. Add cacdpt4.lib to the end of the libraries listed in the *Object/library module* text box.
5. Enter c:\cac\dpt\lib\release (assuming c:\cac was the installation directory) in the *Additional library path* text box if *Win32 Release* was selected for the configuration  
or
6. Enter c:\cac\dpt\lib\debug if *Win32 Debug* was selected for the configuration.

## 9 API Functions

This section describes the API library functions for both DPT channelization and generic DPT board operations.

### 9.1 Function Descriptions

#### 9.1.1 `dp_close_signal`

##### Summary

Closes a signaling channel and frees associated system resources.

##### Function Prototype

```
int dp_close_signal( PCI_SIGNAL * psignal)    Pointer to open signaling
                                                channel
```

##### Description

This function closes a signaling channel and frees associated system resources.

If this function is called on a channel opened for writing and the framer is configured for T1 mode then Robbed-bit signaling is disabled for the timeslots associated with the channel.

##### Return Values

Return	errno	Meaning
0	EINVAL	Invalid parameter
1	No Change	Success

### 9.1.2 dp\_get\_signal

#### Summary

Reads the framer signaling data registers (non-blocking reads).

#### Function Prototype

```
int dp_get_signal(    PCI_SIGNAL * psignal,    Pointer to signaling channel
                   signalBuf_t * sigBuf)      opened for reading
                                                Pointer to a signalBuf_t
                                                structure
```

#### Description

This function returns the signaling data for the timeslots associated with the channel. The signaling data is read from the framer signaling registers.

See the dp\_read\_signal() for the definition of *signalBuf\_t*.

The *data* array member of the *sigBuf* structure holds the signal data for the timeslots.

#### Return Values

Return	Errno	Meaning
-1	EINVAL	Invalid parameter
other	No Change	Success

### 9.1.3 dp\_open\_signal

#### Summary

Opens a channel to send or receive telecom signaling data.

#### Function Prototype

```
PCI_SIGNAL * dp_open_signal( int      pci_id,      Board number
                           char      framer_id,    Connector or framer ID
                           int      flags,        Open mode flags
                           uint32_t  signalMask)   Timeslot bit mask
```

#### Description

This function opens a channel for sending or receiving signaling data.

The *pci\_id* parameter gives the index of the desired board (the numeric portion of the board name).

The *framer\_id* parameter is 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. If *flags* contains CHAN\_NOMAP *framer\_id* is used directly. Otherwise it is interpreted as a connector ID using `pci_chan_log2phys()`. If *flags* contains CHAN\_WRITE the channel is opened for sending signaling data, otherwise the channel is opened for receiving signaling data.

The *flags* parameter consists of one or more of the following: either CHAN\_READ or CHAN\_WRITE to specify the direction of the channel and CHAN\_NOMAP to specify that the *framer\_id* is not to be interpreted as a connector ID.

The *signalMask* parameter is a bit mask indicating the timeslots associated with the channel. Multiple timeslots can be associated with one channel. Bit 0 indicates the first timeslot in the data stream. In T1 mode, bit 0 through bit 23 are valid. In E1 mode bits 1 through bit 15 and bit 17 through bit 32 are valid.

The selected framer must be configured prior to using this function. Also in E1 mode, the framer must be configured for Channel Associated Signaling (CAS). In T1 mode, Robbed-Bit signaling is enabled on the selected timeslots by the `dp_open_signal()` if the *flags* parameter contains CHAN\_WRITE. The framer can be configured using either `pci_open_chan()` or `pci_framer_set_mode()`.

#### Return Values

Return	Errno	Meaning
NULL	ENOMEM	Insufficient Memory
NULL	EINVAL	Invalid parameter, framer not configured
NULL	ENODEV	Device not available
NULL	EBUSY	All signaling channels in use
Other	No Change	Pointer to open signaling channel

### 9.1.4 dp\_read\_signal

#### Summary

Reads the framer signaling data registers (blocking reads).

#### Function Prototype

```
int dp_read_signal(    PCI_SIGNAL *   psignal,    Pointer to signaling channel
                    signalBuf_t *   sigBuf)     opened for reading
                                                Pointer to a signalBuf_t
                                                structure
```

#### Description

This function returns the signaling data for the timeslots associated with the channel. The signaling data is read from the framer signaling registers.

The first time this function is called on the channel, it will immediately return with the present state of the signaling data for the timeslots associated with the channel. Subsequent calls will block until a state change on any of the timeslots.

*signalBuf\_t* is a structure defined in *dpioctl.h* and has two members: *signalMask* and *data*. *signalMask* is defined as an *uint32\_t* and is a bit mask. *data* is defined as an array of 32 *uint8\_t* and holds the signaling data. Only the lower nibble of each element is valid. The bits of *signalMask* correspond to the elements of *data*, bit 0 of *signalMask* is associated with element 0 of *data*.

The *signalMask* member of *sigBuf* indicates the timeslots that caused the function to return.

#### Return Values

Return	Errno	Meaning
-1	EINVAL	Invalid parameter
other	No Change	Success

### 9.1.5 dp\_write\_signal

#### Summary

Updates the signaling data registers.

#### Function Prototype

```
int dp_write_signal(  PCI_SIGNAL *  psignal,      Pointer to signaling channel
                    signalBuf_t *  sigBuf)      opened for writing
                                                    Pointer to a signalBuf_t
                                                    structure
```

#### Description

This function writes the signaling data for the timeslots associated with the channel.

See the `dp_read_signal()` for the definition of *signalBuf\_t*.

Only the timeslots indicated in the *signalMask* member of the *sigBuf* structure are updated to the values in the *data* array member of the *sigBuf* structure. Only the lower nibble of each *data* element is valid. The *sigBuf* structure is updated with the present state of the transmit signaling registers

`dp_write_signal()` does not ensure all signaling data registers are updated in the same multi-frame.

#### Return Values

Return	Errno	Meaning
-1	EINVAL	Invalid parameter
0	No Change	Success

### 9.1.6 flash\_erase\_block

#### Summary

Erases a single block from baseboard flash memory.

#### Function Prototype

```
int flash_erase_block( PCI_MOD * pci,          Pointer to open baseboard
                      uint32_t  addr_of_block) Byte offset of block to be erased
```

#### Description

The flash is divided into multiple blocks of varying size. Although the flash can be read or written on 16-bit word boundaries it can only be erased on block boundaries. This function takes a pointer to an open baseboard and an offset within the flash. The block containing the specified offset is erased (all bits set to 1).

A positive return value indicates a hardware error in the flash. If such a condition persists, contact CAC for assistance.

#### Return Values

Return	Errno	Meaning
0	No Change	Success
-1	EINVAL	Invalid Argument
Small Positive	No Change	Error Code from Flash

### 9.1.7 flash\_read\_a16b

#### Summary

Reads an array of 16-bit words from baseboard flash memory.

#### Function Prototype

```
int flash_read_a16b( PCI_MOD * pci,      Pointer to open baseboard
                    uint32_t  addr,      Byte offset of data in flash
                    uint32_t  nshorts,   Number of 16-bit words to read
                    uint16_t * data)     Buffer to receive data
```

#### Description

This function reads an array of 16-bit words from baseboard flash memory into a user-supplied buffer. The *addr* parameter is a byte address and must be aligned on a 16-bit boundary.

#### Return Values

Return	Errno	Meaning
0	No Change	Success
-1	EINVAL	Invalid Argument

### 9.1.8 flash\_read\_i16b

#### Summary

Reads a single 16-bit word from baseboard flash memory.

#### Function Prototype

```
int flash_read_i16b( PCI_MOD * pci,      Pointer to open baseboard
                    uint32_t  addr)     Byte offset of data in flash
```

#### Description

This function reads a single 16-bit word from baseboard flash memory. The *addr* parameter is a byte address and must be aligned on a 16-bit boundary.

#### Return Values

Return	Errno	Meaning
16-bit value	No Change	Read Data
-1	EINVAL	Invalid Argument

### 9.1.9 flash\_write\_a16b

#### Summary

Writes an array of 16-bit words to baseboard flash memory.

#### Function Prototype

```
int flash_write_a16b( PCI_MOD * pci,      Pointer to open baseboard
                    uint32_t  addr,      Byte offset of data in flash
                    uint32_t  nshorts,   Number of 16-bit words to
                                        write
                    uint16_t * data)     Buffer containing data to write
```

#### Description

This function writes an array of 16-bit words from a user-supplied buffer to baseboard flash memory. The *addr* parameter is a byte address and must be aligned on a 16-bit boundary.

Due to the nature of flash memory devices bits that are 0 cannot be overwritten with a 1. If it is necessary to change the contents of flash the `flash_eerom_erase()` function should be used to erase the flash block to all 1's.

A positive return value indicates a hardware error in the flash. If such a condition persists, contact CAC for assistance.

#### Return Values

Return	errno	Meaning
0	No Change	Success
-1	EINVAL	Invalid Argument
Small Positive	No Change	Error Code from Flash

### 9.1.10 flash\_write\_i16b

#### Summary

Writes a single 16-bit word to baseboard flash memory.

#### Function Prototype

```
int flash_write_i16b( PCI_MOD * pci,   Pointer to open baseboard
                    uint32_t  addr,   Byte offset of data in flash
                    uint16_t  data)   Data to write
```

#### Description

This function writes a single 16-bit word to baseboard flash memory. The *addr* parameter is a byte address and must be aligned on a 16-bit boundary.

Due to the nature of flash memory devices, bits that are 0 cannot be overwritten with a 1. If it is necessary to change the contents of flash the `flash_eerom_erase()` function should be used to erase the flash block to all 1's.

A positive return value indicates a hardware error in the flash. If such a condition persists, contact CAC for assistance.

#### Return Values

Return	Errno	Meaning
0	No Change	Success
-1	EINVAL	Invalid Argument
Small Positive	No Change	Error Code from Flash

### 9.1.11 hto<sub>b</sub>\_16

#### Summary

Converts a 16-bit value from host byte ordering to big-endian byte ordering.

#### Function Prototype

```
uint16_t htob_16( uint16_t val) Value to convert
```

#### Description

This function converts a 16-bit value from the host's native byte order to big-endian byte order. If the host is big-endian this function is a no-op. Otherwise it swaps the bytes within *val*.

#### Return Values

Return	errno	Meaning
16-bit value	No Change	Big-endian value

### 9.1.12 hto<sub>b</sub>\_32

#### Summary

Converts a 32-bit value from host byte ordering to big-endian byte ordering.

#### Function Prototype

```
uint32_t htob_32( uint32_t val) Value to convert
```

#### Description

This function converts a 32-bit value from the host's native byte order to big-endian byte order. If the host is big-endian this function is a no-op. Otherwise it swaps the bytes within *val*.

#### Return Values

Return	errno	Meaning
32-bit value	No Change	Big-endian value

### 9.1.13 htol\_16

#### Summary

Converts a 16-bit value from host byte ordering to little-endian byte ordering.

#### Function Prototype

```
uint16_t htol_16( uint16_t val) Value to convert
```

#### Description

This function converts a 16-bit value from the host's native byte order to little-endian byte order. If the host is little-endian this function is a no-op. Otherwise it swaps the bytes within *val*.

#### Return Values

Return	errno	Meaning
16-bit value	No Change	Little-endian value

### 9.1.14 htol\_32

#### Summary

Converts a 32-bit value from host byte ordering to little-endian byte ordering.

#### Function Prototype

```
uint32_t htol_32( uint32_t val)  Value to convert
```

#### Description

This function converts a 32-bit value from the host's native byte order to little-endian byte order. If the host is little-endian this function is a no-op. Otherwise it swaps the bytes within *val*.

#### Return Values

Return	errno	Meaning
32-bit value	No Change	Little-endian value

### 9.1.15 mod\_eerom\_erase

#### Summary

Erases a single word in the baseboard or module eerom.

#### Function Prototype

```
int mod_eerom_erase(  PCI_MOD * pci,      Pointer to open resource
                    int      modnum,    Module number
                    int      romaddr)   Word address in eerom to erase,
                                        or -1 for entire device
```

#### Description

The eerom on DPT baseboards and modules provides 64 16-bit words of nonvolatile storage. Before a word can be written it must first be erased (all bits set to 1). This function takes a pointer to an open resource and erases a single word within one of the eeroms. The eerom is selected by the *modnum* parameter: 0 for module 'a', 1 for module 'b', and -1 for the baseboard. Values of *romaddr* from 0 through 63 select the word to be erased. A *romaddr* of -1 selects the entire device.

The behavior of this function is not affected by the module associated with *pci*.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid Argument
1	No Change	Success

### 9.1.16 mod\_eerom\_read

#### Summary

Reads a single word from the baseboard or module eeprom.

#### Function Prototype

```
int mod_eerom_read(   PCI_MOD * pci,      Pointer to open resource
                    int      modnum,    Module number
                    int      romaddr)   Word address in eeprom to read
```

#### Description

The eeprom on DPT baseboards and modules provides 64 16-bit words of nonvolatile storage. This function takes a pointer to an open resource and reads and returns as single word from one of the eeproms. The eeprom is selected by the *modnum* parameter: 0 for module 'a', 1 for module 'b', and -1 for the baseboard. Values of *romaddr* from 0 through 63 select the word to be read.

The behavior of this function is not affected by the module associated with *pci*.

#### Return Values

Return	errno	Meaning
-1	EINVAL	Invalid Argument
16-bit value	No Change	Read data

**9.1.17 mod\_eerom\_write****Summary**

Writes a single word to the baseboard or module eerom.

**Function Prototype**

```
int mod_eerom_write(  PCI_MOD * pci,      Pointer to open resource
                    int      modnum,   Module number
                    int      romaddr,  Word address in eerom to write
                    uint16_t romdata  16-bit value to write
```

**Description**

The eerom on DPT baseboards and modules provides 64 16-bit words of nonvolatile storage. This function takes a pointer to an open resource and writes a single word to one of the eeroms. The eerom is selected by the *modnum* parameter: 0 for module ‘a’, 1 for module ‘b’, and –1 for the baseboard. Values of *romaddr* from 0 through 63 select the word to be erased.

Before a word can be written it must first be erased (all bits set to 1) with `mod_eerom_erase()`.

The behavior of this function is not affected by the module associated with *pci*.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid Argument
1	No Change	Success

### 9.1.18 pci\_chan\_autoidle

#### Summary

Gets and sets auto-idle mode for a transmit channel.

#### Function Prototype

```
int pci_chan_autoidle(  PCI_CHAN *  chan,    Pointer to open transmit channel
                       int          enable)  New enable, or -1 to query
```

#### Description

This function can be used to check and set the auto-idle mode for an open transmit channel. Specify an *enable* of 0 to select a fixed idle code set by `pci_chan_set_idle()`, 1 to select automatic idle code selection (the last transmitted byte is used as the idle code), and -1 to leave the mode unchanged. The function returns the previous mode (1 for auto-idle, 0 for fixed idle). The default mode is 0 (fixed idle code).

#### Return Values

Return	errno	Meaning
0	No Change	Channel was using fixed idle mode
1	No Change	Channel was using auto-idle mode
<0	EBADF	<i>chan</i> is not a valid transmit channel

### 9.1.19 pci\_chan\_blocking

#### Summary

Gets and sets blocking mode for a channel.

#### Function Prototype

```
int pci_chan_blocking(  PCI_CHAN *  chan,      Pointer to open channel
                      int          mode)     New mode, or -1 to query
```

#### Description

This function can be used to check and set the blocking mode for an open channel. Specify a *mode* of 0 to select non-blocking I/O, 1 to select blocking I/O, and -1 to leave the mode unchanged. The function returns the previous mode (1 for blocking, 0 for non-blocking).

#### Return Values

Return	errno	Meaning
0	No Change	Channel was using non-blocking I/O
1	No Change	Channel was using blocking I/O
<0	EBADF	<i>chan</i> is not a valid channel

**9.1.20 pci\_chan\_buf\_usage****Summary**

Returns statistics about a channel's buffers.

**Function Prototype**

```
int pci_chan_buf_usage(  PCI_CHAN *  chan,    Pointer to open channel
                        int          mode,    Query mode
                        int          units)    Units for result
```

**Description**

This function returns information about a channel's buffers. If *mode* is 0, the amount of data in the hardware buffer is returned. With a *mode* of 1, the amount of free space in the hardware buffer is returned. If *mode* is 2 the size of the hardware buffer is returned. Statistics for the double buffer used for receive channels can be retrieved by adding 4 to the mode.

The *units* parameter selects the units of the result. A value of 0 selects bytes, 1 selects frames, and 2 selects milliseconds.

**Return Values**

Return	errno	Meaning
<0	EBADF	<i>chan</i> is not a valid channel
non-negative	No Change	Requested data

**9.1.21 pci\_chan\_check\_versions****Summary**

Consistency checking is performed between the library, driver and hardware versions..

**Function Prototype**

```
int      pci_chan_check_versions (
                PCI_CHAN*      channel,      channel
                int*           report)      report version mismatch
```

**Description**

Version numbers for the library, driver and hardware logic definitions are compared. If all of the major and minor versions are equal the function returns a non zero value. If a mismatch occurs for any of the versions then the function returns zero. If the value pointed to by the report parameter is non-zero then a line of output is written for each mismatch to stderr and the value pointed to by report is set to zero. This prevents the same message from being written if multiple opens are performed.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid Argument
0	EDEADLK	Version mismatch
1	No Change	Success

### 9.1.22 pci\_chan\_forcexmt

#### Summary

Force transmission of data waiting in transmit buffer.

#### Function Prototype

```
int pci_chan_forcexmt( PCI_CHAN * chan)    Pointer to open channel
```

#### Description

For a transmit channel this function request the DPT embedded code to send whatever data is waiting in the channel's transmit buffer.

#### Return Values

Return	errno	Meaning
0	unchanged	Success
-1	EINVAL	Attempt to use for a receive channel or the installed DPT embed code does not support this command.
-1	Various	An error occurred.

#### Discussion

This function is only required when sending data of a bursty nature. Normally, when the DPT has been sending idle bytes because the transmit buffer underflowed, new data is not sent until either one second (8000 frames) worth of data has been written to the transmit buffer or one second has elapsed since any new data has been written to the transmit buffer. If this delay is too long, the new data can be forced to be sent using this function. The application must ensure that the transmit buffer contains an appropriate multiple of bytes before calling this function. See the description of the pci\_pad\_chan() function.

Note: this function requires the DPT board to be running embedded code version 1.4.1 or higher.

### 9.1.23 pci\_chan\_get\_idle

#### Summary

Returns the idle code for a transmit channel.

#### Function Prototype

```
int pci_chan_get_idle(    PCI_CHAN * chan)    Pointer to open transmit channel
```

#### Description

This function returns the idle code for an open transmit channel. This value is transmitted in all time slots whenever there is no pending transmit data. Whenever this occurs a flag is set that causes the next call to `pci_write_chan()` to return an error indicating that an underflow occurred.

The embedded code will always switch between idle and data on an even frame boundary. Data transmission will not resume until either one second of data is buffered or one second elapses after at least a frame's worth of transmit data is successfully written.

The default idle code for a newly opened channel is the same as the corresponding framer's idle code as returned by `pci_framer_get_idle()`.

#### Return Values

Return	errno	Meaning
-1	EBADF	<i>chan</i> is not a valid transmit channel
8-bit value	No Change	Current idle code

### 9.1.24 pci\_chan\_get\_signal

#### Summary

Returns the event handle or software signal for the event

#### Function Prototype

```
int pci_chan_get_signal(  PCI_CHAN * chan,  Pointer to open channel
                        int mode)  event
```

#### Description

For the windows environment, the function returns the event handle installed by the user application. For the Linux environment, the function returns the software signal for the specified event.

Mode is one of the following events: . CHAN\_SIG\_IO, CHAN\_SIG\_LOS, CHAN\_SIG\_OVF or CHAN\_SIG\_PURGE.

Only one event should be specified in the mode parameter.

#### Return Values

Return	errno	Meaning
-1	EINVAL	Invalid parameter
-1	EBADF	chan is NULL
other	No Change	Success

### 9.1.25 pci\_chan\_set\_idle

#### Summary

Sets the idle code for a transmit channel.

#### Function Prototype

```
int pci_chan_set_idle(    PCI_CHAN    * chan,    Pointer to open transmit channel
                        uint8_t      idle)      New idle code
```

#### Description

This function sets the idle code for an open transmit channel. This value is transmitted in all time slots whenever there is no pending transmit data. Whenever this occurs a flag is set that causes the next call to `pci_write_chan()` to return an error indicating that an underflow occurred.

The embedded code will always switch between idle and data on an even frame boundary. Data transmission will not resume until either one second of data is buffered or one second elapses after at least a frame's worth of transmit data is successfully written.

The default idle code for a newly opened channel is the same as the corresponding framer's idle code as returned by `pci_framer_get_idle()`.

#### Return Values

Return	errno	Meaning
-1	EBADF	<i>chan</i> is not a valid transmit channel
8-bit value	No Change	Previous idle code

### 9.1.26 pci\_chan\_set\_signal

#### Summary

Install a user callback function.

#### Function Prototype

Windows:

```
int pci_chan_set_signal( PCI_CHAN * chan, Pointer to an open channel
                        int mode event
                        HANDLE signal ) CreateEvent Handle
```

Solaris and Linux:

```
int pci_chan_set_signal( PCI_CHAN * chan, Pointer to an open channel
                        int mode event
                        int signal ) software signal
```

#### Description

This function enables a user application to receive notification of the occurrence of the specified event. In the windows environment, a user supplied event handle is signaled when the specified event occurs. In the Solaris or Linux environments, the specified software signal is raised if the event specified by mode occurs.

Mode is one of the following event macros:

Event Macro	Write Channel	Read Channel	Event
CHAN_SIG_IO	Not Supported	Not Supported	Data ready for receive
CHAN_SIG_LOS	Not Supported (windows only)		Loss of synchronization
CHAN_SIG_OVF			Write Channel: Channel under flow error Read Channel: Channel overflow error
CHAN_SIG_PURGE			Channel close or channel reset

Only one event should be specified in the mode parameter.

#### Return Values

Return	Errno	Meaning
-1	EINVAL	Invalid parameter
-1	EBADF	chan is NULL
0		Success

### 9.1.27 pci\_chan\_source

#### Summary

Sets a receive channel as the data source for a transmit channel.

#### Function Prototype

```
int pci_chan_source(  PCI_CHAN *  dest,      Pointer to open transmit channel
                    PCI_CHAN *  src)       Pointer to open receive channel
```

#### Description

This function sets *src* as the data source for *dest*. The embedded software will copy received data from *src* to the transmit buffer for *dest*. While there is a registered source for a channel `pci_write_chan()` will fail, setting `errno` to `EBUSY`, but data can still be received from *src* with `pci_read_chan()`. The mapping can be cleared by calling `pci_chan_source()` with the *src* set to `NULL`.

The *src* and *dest* channels must be on the same board and must have the same number of timeslots.

All transmitters on a board share a single transmit clock. By default this clock comes from a local crystal. When retransmitting data it is important to lock the transmit clock to the receive clock. This can be accomplished by including the `E1_RX_FAST_CLK` flag in the *framer\_mode* parameter to `pci_open_chan()`. The transmit clock will be taken from the fastest receiver with this mode set.

#### Return Values

Return	errno	Meaning
-1	EBADF	<i>dest</i> is not a valid transmit channel
-1	EBUSY	<i>src</i> is not <code>NULL</code> and <i>dest</i> already has a source
-1	EINVAL	<i>src</i> is not a valid receive channel, is not on the same board as <i>dest</i> , or does not have the same number of slots as <i>dest</i>
0	No Change	Channel source updated successfully

### 9.1.28 pci\_chan\_waitsync

#### Summary

Waits for a receive channel to sync to the incoming data stream or checks the sync status of a channel.

#### Function Prototype

```
int pci_chan_waitsync(   PCI_CHAN   *chan,   Pointer to open receive channel
                        int         timeout) Number of seconds to wait
```

#### Description

This function blocks until the framer for a receive channel synchronizes to the incoming data stream. The conditions to declare sync depend on the framer mode set in the call to `pci_open_chan()`. With a *timeout* of zero the function will return immediately with the current sync status of the framer. A non-zero *timeout* specifies the maximum number of seconds that the function should wait for the framer to achieve sync.

#### Return Values

Return	errno	Meaning
-1	EBADF	<i>chan</i> is not a valid receive channel
0	No Change	<i>chan</i> is in sync
1	ETIMEOUT	Timed out waiting for sync

### 9.1.29 pci\_check\_versions

#### Summary

Consistency checking is performed between the library, driver and hardware versions..

#### Function Prototype

```
int pci_check_versions (  PCI_MOD*  board,      resource
                        int*      report)      report version mismatch
```

#### Description

.Version numbers for the library, driver and hardware logic definitions are compared. If all of the major and minor versions are equal the function returns a non zero value. If a mismatch occurs for any of the versions then the function returns zero. If the value pointed to by the report parameter is non-zero then a line of output is written for each mismatch to stderr and the value pointed to by report is set to zero. This prevents the same message from being written if multiple opens are performed.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid Argument
0	EDEADLK	Version mismatch
1	No Change	Success

### 9.1.30 pci\_clear\_driver\_versions

#### Summary

Clears the version information cached in the device driver.

#### Function Prototype

```
int pci_clear_driver_versions( PCI_MOD * dpt, Board resource
```

#### Description

The DPT device driver caches version information to avoid having to retrieve hardware version data every time a request is made. In some situations it may be necessary to clear the driver's version cache.

#### Return Values

Return	errno	Meaning
0	various	An error occurred
1	No Change	Success

### 9.1.31 **pciclose**

#### Summary

Closes a DPT resource and frees associated system resources.

#### Function Prototype

```
int pciclose( PCI_MOD * pci)  Pointer to open DPT board or module
```

#### Description

This function closes an open resource and frees the associated system resources. This function should always be called whenever a DPT resource is no longer needed.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid Arguments
1	No Change	Success

### 9.1.32 pci\_close\_chan

#### Summary

Closes a channel and frees associated system resources.

#### Function Prototype

```
int pci_close_chan( PCI_CHAN *chan)  Pointer to open channel
```

#### Description

This function closes an open channel and frees the associated system resources. This function should always be called whenever a channel is no longer needed.

If called for a transmit channel any buffered data is discarded. Use `pci_flush_chan()` before closing a transmit channel to ensure that all written data is transmitted across the physical link.

#### Return Values

Return	errno	Meaning
-1	EINVAL	<i>chan</i> is not a valid channel
0	No Change	Success

### 9.1.33 pci\_comet\_log2phys

#### Summary

Returns the framer number associated with a connector.

#### Function Prototype

```
int pci_comet_log2phys(  int      pci_id,   Board number
                        char      log_id,   Connector ID
                        int      is_tx)    0 for receive, 1 for transmit
```

#### Description

DPT boards with coaxial or triaxial connectors need to associate four unidirectional connectors with four bi-directional framers. Jumper settings are used to configure each connector for transmit or receive. The jumper settings are then recorded in the baseboard eeprom using the *dpserial* utility. This function is used by *pci\_open\_chan()* to determine which framer device is associated with a specified connector.

The *pci\_id* parameter gives the index of the desired board (the numeric portion of the board name). The *log\_id* parameter specifies the connector, starting from the top of the board, as 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. The *is\_tx* parameter indicates whether the receive (zero) or transmit (non-zero) mapping is being queried.

If the board is configured with RJ-45 (bi-directional) connectors or the eeprom cannot be read, the function returns *log\_id* converted to be in the range 0-3. Otherwise, if the specified connector is configured in the specified direction the function returns the framer id in the range 0-3.

#### Return Values

Return	errno	Meaning
-1	EINVAL	Invalid argument
-1	ENXIO	Connector is not configured in specified direction
0-3	No Change	Framer associated with given connector-direction pair

**9.1.34 pci\_comet\_read****Summary**

Read data from a Comet register.

**Function Prototype**

```
int pci_comet_read(   PCI_MOD *   pci,      Pointer to open resource
                    char         framer,   Framer number
                    int          reg       Register offset
                    uint8_t      datap)    Pointer to receive data
```

**Description**

This function provides read access to the registers in the Comet framer chip (PMC Sierra PM4351).

**Arguments**

- pci* pointer to a PCI\_MOD structure refereeing the DPT board for the Comet to be accessed.
- framer* specifies which of the four framers in the Comet to access. The framer may be specified using integer values 0 – 3 or ASCII letters ‘a’ – ‘d’.
- reg* specifies register to be read. Values range from 0 to 256. See the PM4351 data sheet for register descriptions.
- datap* pointer to the application’s variable where the data from the register should be stored.

**Return Values**

Return	errno	Meaning
1	No change	Register successfully read and data stored in <i>datap</i> .
0	EINVAL	Invalid argument.
0	various	Error occurred reading register

**9.1.35 pci\_comet\_write****Summary**

Write data to a Comet register.

**Function Prototype**

```
int pci_comet_write(  PCI_MOD *   pci,      Pointer to open resource
                    char         framer,   Framer number
                    int          reg       Register offset
                    uint8_t      datap)    Pointer to receive data
```

**Description**

This function provides write access to the registers in the Comet framer chip (PMC Sierra PM4351).

**Arguments**

- pci* pointer to a PCI\_MOD structure refereeing the DPT board for the Comet to be accessed.
- framer* specifies which of the four framers in the Comet to access. The framer may be specified using integer values 0 – 3 or ASCII letters ‘a’ – ‘d’.
- reg* specifies the register to be written. Values range from 0 to 256. See the PM4351 data sheet for register descriptions.
- datap* pointer to the application’s variable where the data to be written is stored.

**Return Values**

Return	errno	Meaning
1	No change	Register successfully written.
0	EINVAL	Invalid argument.
0	various	Error occurred reading register

**9.1.36 pci\_ctrl****Summary**

Interface for PCI ioctl requests.

**Function Prototype**

```
int pci_ctrl(          PCI_MOD*   pci,      pointer to open module structure
                   int          cmd       ioctl command code
                   int          val)      parameter for ioctl if needed
```

**Description**

The supported ioctl commands are:

PCI_ATOMIC_RMW	performs atomic read modify write
PCI_CHAN_CTL, PCI_CHAN_CLOSE, PCI_CHAN_SET_SIG, PCI_CHAN_GET_SIG, PCI_CHAN_BUFSTAT, PCI_CHAN_NBLOCK, PCI_CHAN_FLUSH, PCI_CHAN_ID	used by library channel functions
PCI_DRVVERSION	gets version number of driver
PCI_GETSIG	the current software signal for the board.
PCI_PURGESIGNAL	set and get purge signal for resource
PCI_RESET	sends purge signal to processes which have resource open
PCI_SETSIG	set the software signal for the board.
PCI_BOARDTYPE	PCI_V6M6, PCI_V6M6HS or 0 if unknown.

### 9.1.37 pci\_dl\_a8b

#### Summary

Writes an array of 8-bit values to a DPT resource.

#### Function Prototype

```
int pci_dl_a8b( PCI_MOD * pci,      Pointer to open resource
               uint32_t  addr,      Destination address
               uint32_t  nbytes,    Number of bytes to transfer
               uint8_t *  data)     Pointer to data to write
```

#### Description

This function writes an array of 8-bit items to a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer goes to the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer goes to the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Destination range does not fit within a valid memory region
1	No Change	Success

**9.1.38 pci\_dl\_a16b****Summary**

Writes an array of 16-bit values to a DPT resource.

**Function Prototype**

```
int pci_dl_a16b( PCI_MOD * pci,      Pointer to open resource
                uint32_t  addr,      Destination address
                uint32_t  nshorts,   Number of 16-bit words to transfer
                uint16_t * data)     Pointer to data to write
```

**Description**

This function writes an array of 16-bit items to a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer goes to the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer goes to the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Destination range does not fit within a valid memory region
1	No Change	Success

### 9.1.39 pci\_dl\_a32b

#### Summary

Writes an array of 32-bit values to a DPT resource.

#### Function Prototype

```
int pci_dl_a32b(  PCI_MOD * pci,      Pointer to open resource
                 uint32_t  addr,     Destination address
                 uint32_t  nlongs,   Number of 32-bit words to transfer
                 uint32_t * data)    Pointer to data to write
```

#### Description

This function writes an array of 32-bit items to a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer goes to the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer goes to the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 32-bit boundary
0	ENXIO	Destination range does not fit within a valid memory region
1	No Change	Success

### 9.1.40 pci\_dl\_cfg8

#### Summary

Writes an 8-bit value to PCI config space using big-endian addressing.

#### Function Prototype

```
int pci_dl_cfg8( PCI_MOD * pci,      Pointer to open resource
                uint32_t  off,      Offset in config space
                uint32_t * datap)   Pointer to data to write
```

#### Description

This function writes a single 8-bit value to the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses big-endian addressing compatible with CAC expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions (`pci_dl_cfg8le` to write 8-bit data).

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

### 9.1.41 pci\_dl\_cfg8le

#### Summary

Writes an 8-bit value to PCI config space using little-endian addressing.

#### Function Prototype

```
int pci_dl_cfg8(  PCI_MOD *   pci,      Pointer to open resource
                 uint32_t   off,      Offset in config space
                 uint32_t *  datap)    Pointer to data to write
```

#### Description

This function writes a single 8-bit value to the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses little-endian addressing compatible with standard PCI interfaces including the host and expansion bridges on the baseboard and the bridges on certain expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

### 9.1.42 pci\_dl\_cfg16

#### Summary

Writes a 16-bit value to PCI config space.

#### Function Prototype

```
int pci_dl_cfg16(  PCI_MOD *   pci,      Pointer to open resource
                  uint32_t   off,      Offset in config space
                  uint32_t *  datap)    Pointer to data to write
```

#### Description

This function writes a single 16-bit value to the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF. Misaligned offsets are silently coerced to 16-bit alignment.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses big-endian addressing compatible with CAC expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions (pci\_dl\_cfg16le to write 16-bit data). The data is in host byte order.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

**9.1.43 pci\_dl\_cfg16le****Summary**

Writes a 16-bit value to PCI config space using little-endian addressing.

**Function Prototype**

```
int pci_dl_cfg16(  PCI_MOD *   pci,      Pointer to open resource
                  uint32_t   off,      Offset in config space
                  uint32_t *  datap)   Pointer to data to write
```

**Description**

This function writes a single 16-bit value to the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF. Misaligned offsets are silently coerced to 16-bit alignment.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses little-endian addressing compatible with standard PCI interfaces including the host and expansion bridges on the baseboard and the bridges on certain expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions. The data is in host byte order.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

### 9.1.44 pci\_dl\_cfg32

#### Summary

Writes a 32-bit value to PCI config space.

#### Function Prototype

```
int pci_dl_cfg32(  PCI_MOD *   pci,      Pointer to open resource
                  uint32_t   off,      Offset in config space
                  uint32_t *  datap)   Pointer to data to write
```

#### Description

This function writes a single 32-bit value to the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF. Misaligned offsets are silently coerced to 32-bit alignment.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function is endian-neutral and may be used to access any device's config space. The data is in host byte order.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

**9.1.45 pci\_dl\_i8b****Summary**

Writes a single 8-bit value to a DPT resource.

**Function Prototype**

```
int pci_dl_a8b(  PCI_MOD * pci,      Pointer to open resource
                uint32_t  addr,      Destination address
                uint8_t   val)       Value to write
```

**Description**

This function writes a single 8-bit item to a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer goes to the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer goes to the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Destination address not within a valid memory region
1	No Change	Success

**9.1.46 pci\_dl\_i16b****Summary**

Writes a single 16-bit value to a DPT resource.

**Function Prototype**

```
int pci_dl_a16b(  PCI_MOD * pci,      Pointer to open resource
                 uint32_t  addr,     Destination address
                 uint16_t  val)      Value to write
```

**Description**

This function writes a single 16-bit item to a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer goes to the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer goes to the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Destination address not within a valid memory region
1	No Change	Success

**9.1.47 pci\_dl\_i32b****Summary**

Writes a single 32-bit value to a DPT resource.

**Function Prototype**

```
int pci_dl_a32b( PCI_MOD * pci,      Pointer to open resource
                uint32_t  addr,      Destination address
                uint32_t  val)       Value to write
```

**Description**

This function writes a single 32-bit item to a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer goes to the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer goes to the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 32-bit boundary
0	ENXIO	Destination address not within a valid memory region
1	No Change	Success

**9.1.48 pci\_eerom\_erase****Summary**

Erases a single word in the baseboard or module eeprom.

**Function Prototype**

```
int pci_eerom_erase(  char * boardname,  Resource name
                    int   modnum,      Module number
                    int   romaddr)     Word address in eeprom to erase,
                                       or -1 for entire device
```

**Description**

The eeprom on a DPT baseboard or module provides 64 16-bit words of nonvolatile storage. Before a word can be written it must first be erased (all bits set to 1). This function takes the name of a DPT resource (i.e. "dpt0") and erases a single word within one of the eeproms. The eeprom is selected by the *modnum* parameter: 0 for module 'a', 1 for module 'b', and -1 for the baseboard. Values of *romaddr* from 0 through 63 select the word to be erased. A *romaddr* of -1 selects the entire device.

If a pointer to a resource on the board is already available *mod\_eerom\_erase()* can be used instead. The behavior of this function is not affected by any module specified in *boardname*.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid Argument
1	No Change	Success

**9.1.49 pci\_eerom\_read****Summary**

Reads a single word from the baseboard or module eeprom.

**Function Prototype**

```
int pci_eerom_read(   char *   boardname,   Resource name
                    int       modnum,     Module number
                    int       romaddr)    Word address in eeprom to read
```

**Description**

The eeprom on a DPT baseboard or module provides 64 16-bit words of nonvolatile storage. This function takes the name of a DPT resource (i.e. “dpt0”) and reads and returns as single word from one of the eeproms. The eeprom is selected by the *modnum* parameter: 0 for module ‘a’, 1 for module ‘b’, and –1 for the baseboard. Values of *romaddr* from 0 through 63 select the word to be read.

If a pointer to a resource on the board is already available *mod\_eerom\_read* () can be used instead.

The behavior of this function is not affected by any module specified in *boardname*.

**Return Values**

Return	errno	Meaning
–1	EINVAL	Invalid Argument
16-bit value	No Change	Read data

**9.1.50 pci\_eerom\_write****Summary**

Writes a single word to the baseboard or module eerom.

**Function Prototype**

```
int pci_eerom_write ( char *   boardname,  Resource name
                    int      modnum,     Module number
                    int      romaddr,    Word address in eerom to write
                    uint16_t  romdata    16-bit value to write
```

**Description**

The eerom on a DPT baseboard or module provides 64 16-bit words of nonvolatile storage. This function takes the name of a DPT resource (i.e. “dpt0”) and writes a single word to one of the eeroms. The eerom is selected by the *modnum* parameter: 0 for module ‘a’, 1 for module ‘b’, and –1 for the baseboard. Values of *romaddr* from 0 through 63 select the word to be erased.

Before a word can be written it must first be erased (all bits set to 1) with `pci_eerom_erase()`.

If a pointer to a resource on the board is already available `mod_eerom_write ()` can be used instead.

The behavior of this function is not affected by any module specified in *boardname*.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid Argument
1	No Change	Success

### 9.1.51 pci\_flush\_chan

#### Summary

Discards buffered data from a receive channel; blocks until buffered data for a transmit channel has been sent.

#### Function Prototype

```
int pci_flush_chan( PCI_CHAN *chan)  Pointer to open channel
```

#### Description

For receive channels this function discards all buffered unread data.

For transmit channels the transmit buffer is padded, as required, data is forced to be transmitted and then it waits until all remaining data has been transmitted.. This function may be called immediately before `pci_close_chan()` to ensure that all written data is transmitted across the physical link.

#### Return Values

Return	errno	Meaning
-1	EBADF	Invalid <i>chan</i>
-1	EBUSY	I/O in progress on <i>chan</i>
-1	EINTR	Interrupted system call
0	No Change	Success

**9.1.52 pci\_framer\_get\_idle****Summary**

Returns the idle code for a framer.

**Function Prototype**

```
int pci_framer_get_idle( int      pci_id,      Board number
                       char      framer_id,    Connector or framer ID
                       uint32_t   flags)       0 or CHAN_NOMAP
```

**Description**

This function returns the idle code for a framer. This value is transmitted in all time slots not associated with a channel and also becomes the default idle code for newly opened transmit channels.

The *pci\_id* parameter gives the index of the desired board (the numeric portion of the board name). The *framer\_id* parameter is 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. If flags contains CHAN\_NOMAP *framer\_id* is used directly. Otherwise, *framer\_id* is interpreted as a connector ID using `pci_chan_log2phys()`. In this case, either the transmit or the receive connector may be specified.

**Return Values**

Return	errno	Meaning
-1	EINVAL	Invalid parameter
8-bit value	No Change	Current idle code

### 9.1.53 pci\_framer\_get\_mode

**Summary** Determine the current mode of a framer.

#### Function Prototype

```
int pci_framer_get_mode( int pci_id, Board number
                        char framer_id, Connector or framer ID
                        uint32_t flags, 0 or CHAN_NOMAP
                        uint32_t * framer_mode) Current framer mode
```

#### Description

The *pci\_id* parameter gives the index of the desired board (the numeric portion of the board name). The *framer\_id* parameter is 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. If flags contains CHAN\_NOMAP *framer\_id* is used directly. Otherwise, *framer\_id* is interpreted as a connector ID using *pci\_chan\_log2phys()*. In this case, either the transmit or the receive connector may be specified.

If *framer\_mode* is not NULL, the addressed location is set to the current framer mode if the function completes successfully. The mode is a bit wise combination of the following flags (from *dpchan.h*):

E1_RX_BYP_JAT	Bypass receive jitter attenuator
E1_RX_UNFRAMED	Bypass receive framer
E1_RX_HDB3	Receive using HDB3 line encoding
E1_RX_CRC_MF	Receive using CRC4 multiframing
E1_RX_CAS_MF	Receive using CAS multiframing
E1_RX_CRC_REFR	Reframe on excessive CRC errors
E1_RX_NO_REFR	Do not reframe after initial sync
E1_RX_FAST_CLK	Receiver is candidate for transmit clock source
E1_TX_BYP_JAT	Bypass transmit jitter attenuator
E1_TX_HDB3	Transmit using HDB3 line encoding
E1_TX_CRC_MF	Transmit using CRC4 multiframing
E1_TX_CAS_MF	Transmit using CAS multiframing
E1_LOOP_SDRAM	Enable SDRAM loopback (processor pre-dual-port)
E1_LOOP_DPRAM	Enable DPRAM loopback (processor post-dual-port)
E1_LOOP_SERIAL	Enable serial loopback (local pre-framer)
E1_LOOP_DIG	Enable digital loopback (local post-framer)
E1_LOOP_PAYLD	Enable payload loopback (remote post-framer)
E1_LOOP_LINE	Enable line loopback (remote pre-framer)

#### Return Values

Return	errno	Meaning
-1	EINVAL	Invalid parameter
0	No Change	Current idle code

**9.1.54 pci\_framer\_set\_idle****Summary**

Sets the idle code for a framer.

**Function Prototype**

```
int pci_framer_set_idle(  int      pci_id,      Board number
                        char      framer_id,   Connector or framer ID
                        uint32_t   flags,      0 or CHAN_NOMAP
                        uint8_t   new_idle)   New idle code
```

**Description**

This function sets the idle code for a framer. This value is transmitted in all time slots not associated with a channel and also becomes the default idle code for newly opened transmit channels.

The *pci\_id* parameter gives the index of the desired board (the numeric portion of the board name). The *framer\_id* parameter is 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. If flags contains CHAN\_NOMAP *framer\_id* is used directly. Otherwise, *framer\_id* is interpreted as a connector ID using `pci_chan_log2phys()`. In this case, either the transmit or the receive connector may be specified.

**Return Values**

Return	errno	Meaning
-1	EINVAL	Invalid parameter
8-bit value	No Change	Previous idle code

### 9.1.55 pci\_framer\_set\_mode

#### Summary

Sets the modes for a framer.

#### Function Prototype

```
int pci_framer_set_mode( int pci_id, Board number
                        char framer_id, Connector or framer ID
                        uint32_t flags, 0 or CHAN_NOMAP
                        uint32_t framer_mode Desired framer mode
                        uint32_t * old_mode) Old framer mode
```

#### Description

The *pci\_id* parameter gives the index of the desired board (the numeric portion of the board name). The *framer\_id* parameter is 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. If flags contains CHAN\_NOMAP *framer\_id* is used directly. Otherwise, *framer\_id* is interpreted as a connector ID using *pci\_chan\_log2phys()*. In this case, either the transmit or the receive connector may be specified.

The *framer\_mode* parameter specifies the desired framer mode. If there are no channels open on the framer the given mode will be set. If *old\_mode* is not NULL, the addressed location is set to the old framer mode if the function completes successfully.

The mode is a bit wise combination of the following flags (from *dpchan.h*):

E1_RX_BYP_JAT	Bypass receive jitter attenuator
E1_RX_UNFRAMED	Bypass receive framer
E1_RX_HDB3	Receive using HDB3 line encoding
E1_RX_CRC_MF	Receive using CRC4 multiframing
E1_RX_CAS_MF	Receive using CAS multiframing
E1_RX_CRC_REFR	Reframe on excessive CRC errors
E1_RX_NO_REFR	Do not reframe after initial sync
E1_RX_FAST_CLK	Receiver is candidate for transmit clock source (see note below)
E1_TX_BYP_JAT	Bypass transmit jitter attenuator
E1_TX_HDB3	Transmit using HDB3 line encoding
E1_TX_CRC_MF	Transmit using CRC4 multiframing
E1_TX_CAS_MF	Transmit using CAS multiframing
E1_LOOP_SDRAM	Enable SDRAM loopback (processor pre-dual-port)
E1_LOOP_DPRAM	Enable DPRAM loopback (processor post-dual-port)
E1_LOOP_SERIAL	Enable serial loopback (local pre-framer)
E1_LOOP_DIG	Enable digital loopback (local post-framer)
E1_LOOP_PAYLD	Enable payload loopback (remote post-framer)
E1_LOOP_LINE	Enable line loopback (remote pre-framer)

**Note:** that the E1\_RX\_FASTCLK flag should NOT be used when configuring the framers for use with the TDM subsystem or when the module expansion fast clock select logic is to be used. See `pci_tdm_clock_cfg()`, `pci_hw_fast_clk()` and `pci_hw_tdmdb_clk()` in section 9.

### Return Values

Return	errno	Meaning
-1	EINVAL	Invalid parameter
-1	EBUSY	Framer in use
-1	PORT_CONFIG_ERROR	Framer port not configured for RX or TX as requested.
0	No Change	Current idle code

### 9.1.56 pci\_framer\_tx\_source

#### Summary

Configures the framer transmitter data source.

#### Function Prototype

```
int pci_framer_tx_source( int pci_id, Board number
                        uint32_t source) Data source –
                                         TX_DATA_SRC_CHAN or
                                         TX_DATA_SRC_TDM
```

#### Description

This function configures the transmitter framer data source. The four framers are sourced from the same source. The framer transmitters can be sourced from either the channelization data hardware or from the TDM subsystem data bus. If the source is the TDM subsystem bus, the framer transmit clock is derived from the TDM subsystem and TDM time slots must be mapping using the `pci_tdm_src_add()` function in order for the framers to receive data from the TDM data bus. See the description of the TDM subsystem API in section 10.2

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid parameter
1	No Change	Success

### 9.1.57 pci\_get\_dpbuffer

#### Summary

Determines the current Dual Port RAM buffer size setting.

#### Function Prototype

```
int pci_get_dpbuffer ( int pci_id) Board number
```

#### Description

The function returns the current setting for the input and output Dual Port RAM buffer size.

## 9.1.58 pci\_get\_info

### Summary

Get the board type, revision and serial number

### Function Prototype

```
int    pci_get_info (
        PCI_MOD *    dev,          board handle
        int *        board_type,   pointer location to store the board type
        int *        board_rev,    pointer location to store the revision
        int *        board_serial ) pointer location to store the serial number.
```

### Description

This function retrieves the board type, hardware revision and serial number for the DPT board referenced by the board handle, *dev*. The values are stored in the respective pointer arguments. Any of the pointers may be NULL for information not required.

The value stored in *board\_type* is one of the following macros:

<u>Board Type Value</u>	<u>Board type description</u>
PCI_DPT4	a DPT4 board
PCI_DPT5	a DPT5 board
PCI_DPT6	a DPT6 board.
PCI_DPT4_UNCFG	a DPT4 board whose host FPGA is not configured

The value stored in *board\_rev* is a two part integer, representing a board revision such as 0.1, 1.10, etc. The major revision (typically the PCB art revision) is stored in the lower 8 bits. The minor revision (typically a modification to the PCB) is stored in the upper 8 bits of the value. The minor revision had not been used prior to April 2009 so most boards shipped before then will indicate a minor revision of 0, regardless of the actual revision of the board. If the value is not set in the board's EEROM the value reported will be the macro: UNINITIALIZED\_EEROM.

The value stored in *board\_serial* is an 8 digit integer. The lower portion of the number is obtained from the serial number information in the board's EEROM which is offset by a value dependent on the board type and revision. If the lower portion is missing from the board's EEROM the value reported will be 0.

### Return Values

Return	errno	Meaning
0	EINVAL	Invalid dev argument
	EIO	Error retrieving information
1	No Change	Success



### 9.1.59 pci\_get\_location

#### Summary

Get the PCI bus, device and function numbers to determine PCI slot location.

#### Function Prototype

```
int      pci_get_location (
        PCI_MOD *   dev,          board handle
        int *       pcibus,      pointer location to store bus number
        int *       pcidev,      pointer location to store device number
        int *       pcifn )     pointer location to store function number
```

#### Description

This function retrieves the PCI bus, device and function number for the DPT board referenced by the board handle, *dev*. The values for the bus, device and function numbers are stored in the respective pointer arguments. Any of the pointers may be NULL for information not required.

The value stored in *pcibus* and *pcidev* indicate the system bus number and the number of the device on that bus. This information may be used to determine which physical PCI slot in the system corresponds to the open device. This is most useful on Linux or Windows systems where the board instance numbers can change when boards are added, removed or moved in the system.

The value stored in *pcifn* will always be 0 for DPT boards on Solaris and Linux. On Windows systems the value is used to report the "DevicePropertyUINumber" as obtained from the operating system; typically a user-perceived slot number.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid dev argument
	EFAULT	System error retrieving information
1	No Change	Success

### 9.1.60 pci\_get\_version

#### Summary

Sets the fields of a struct with the current version numbers for the library, driver and hardware definitions.

#### Function Prototype

```
int      pci_get_version (
                PCI_MOD *   dev,      board handle
                pci_versions * version) pointer to struct to return versions
```

#### Description

The version numbers for the library, driver and hardware logic definitions are returned in the struct pointed to by the version parameter

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid Argument
1	No Change	Success

**9.1.61 pci\_get\_version\_checking****Summary**

Returns the current setting of the version checking options.

**Function Prototype**

```
int    pci_get_version_checking (
        int    board_number,
        int*   require_match
        int*   report_mismatch)
```

**Description**

The current values for the required match and report mismatches options are returned. The required match option applies to all boards. If it is non zero pciopen and pci\_open\_chan will fail if a version mismatch occurs. The report mismatches option is kept on a per board basis for each program. If it is non zero when a mismatch is detected a line of output indicating the mismatch will be written to stderr. The report mismatches option is kept on a per board basis so that once a report has been printed it can be turned off for that board. The two options are treated independently. Reports will still be generated if a mismatch is detected but the require match option is zero.

**Return Values**

<b>Return</b>	<b>errno</b>	<b>Meaning</b>
0	EINVAL	Invalid Argument
1	No Change	Success

### 9.1.62 pcihold

#### Summary

Stops the embedded processor from executing code.

#### Function Prototype

```
int pcihold( PCI_MOD * pci)  Pointer to open resource
```

#### Description

This function stops the embedded processor from executing code. This is used primarily for diagnostics when the global reset must be deasserted but the processor must be prevented from executing.

#### Return Values

Return	errno	Meaning
Non-zero	EINVAL	Invalid parameter
0	No Change	<i>pci</i> was in reset
1	No Change	<i>pci</i> was not in reset

### 9.1.63 pci\_hw\_fast\_clk

**Summary** Configures the DPT module expansion fast clock select hardware logic.

#### Function Prototype

```
int      pci_hw_fast_clk(      int      pciId,      Board number
                               char      framerId,      Connector or framer ID
                               uint32_t  flags,      Mode flags
                               uint32_t  framerMode,      Framer mode
                               uint32_t  enable)      Enable flag
```

#### Description

This function allows a user application to configure the DPT module expansion fast clock select hardware logic. When the fast clock select hardware is enabled, it selects the fastest framer receive clock out of the enabled receive clocks as the source for the TDM sub-system and framer transmit clocks. NOTE: only framers with a valid input signal should be included, otherwise the DPT's local clock could be selected as the fast receive clock. The first time this function is called with the enable flag set to TRUE, it configures the DPT hardware to derive the framer transmit clock from the module expansion fast clock select logic. When this function is called with the enable flag set to FALSE to remove a receive framer from inclusion in the fast clock select and the framer is the only framer currently enabled then the transmit clock is re-configured to derived its clock from the local oscillator. This function must be called once for each framer to be included in the fast clock select logic.

The receive fast clock select hardware can be used in either E1 or T1 framing modes. This hardware must be used in T1 mode if it is desired that the T1 transmit clock be synchronized to the fastest receive clock. This function requires that the DPT board be built with the module expansion option. For the receive fast clock select logic to correctly work, this function configures the TDM clock to be driven from the receive fast clock select logic and starts the TDM subsystem running. The user application should NOT call `pci_tdm_clock_cfg()` to reconfigure the source of the TDM clock or stop the TDM from running after this function as been called.

The *pciId* parameter gives the index of the desired board (the numeric portion of the board name).

The *framerId* parameter is 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. If *flags* contains CHAN\_NOMAP *framerId* is used directly. Otherwise it is interpreted as a connector ID using `pci_chan_log2phys()`.

The *flags* parameter is the same *flags* parameter used in the `pci_open_chan()`. Only the CHAN\_NOMAP flag is tested to determine the interpretation of the *framerId*.

The *framerMode* parameter is the same *framer\_mode* parameter used in the `pci_open_chan()`. This function only tests the T1\_MODE flag to determine the framer clock configuration.

The *enable* parameter is a flag indicating whether the *framerId* is be included or removed from being a candidate in the module expansion fast clock select logic. A non-zero value indicates that the framer should be included, zero indicates that the framer should be removed.

#### Return Values

Return	Errno	Meaning
FALSE	EINVAL	Invalid parameter, framer not included
FALSE	ENODEV	The DPT is configured without the module expansion option
TRUE	No Change	Module expansion fast clock select logic configured

### 9.1.64 pci\_hw\_tdmdb\_clk

**Summary** Configures the DPT module expansion to use TDM daughter-board clock.

#### Function Prototype

```
int      pci_hw_tdmdb_clk(      int      pciId,      Board number
                               uint32_t  framerMode,  Framers mode
                               uint32_t  enable)      Enable flag
```

#### Description

This function allows a user application to configure the DPT module expansion to select the TDM daughter-board clock as the source for the TDM subsystem and framer transmit clocks. This mode is only supported on DPT5 and DPT6 boards. There are currently no supported expansion modules that generate or provide the TDM daughter-board clock. The intended use for this mode is for slaving the TDM and framer transmit clocks to the clock selected on another DPT board. This functionality would require some added connectivity between the master and slave boards.

The *pciId* parameter gives the index of the desired board (the numeric portion of the board name).

The *framerMode* parameter is the same *framer\_mode* parameter used in the `pc_open_chan()`. This function only tests the T1\_MODE flag to determine the framer clock configuration.

The *enable* parameter is a flag indicating whether the daughter-board clock mode should be enabled (if the value is 1) or disabled (if the value is 0).

#### Return Values

Return	Errno	Meaning
FALSE	EINVAL	Invalid parameter, framer not included
FALSE	ENODEV	The board is not a DPT5 or DPT6 or the module expansion option is disabled.
TRUE	No Change	Module expansion daughter-board clock select logic configured

## 9.1.65 pciopen

### Summary

Opens a resource.

### Function Prototype

```
PCI_MOD * pciopen( char * devname) Name of resource to open
```

### Description

This function opens a DPT resource. The *devname* consists of an optional “/dev/” prefix, either “dpt” or “pci”, a board number (small integer starting from 0), and optionally a resource specifier. The resource specifier may be ‘a’ for Module A, ‘b’ for Module B, ‘g’ or ‘h’ for global memory, and ‘q’ or omitted for the baseboard. The baseboard resource is opened non-exclusively, all other resources are opened exclusively.

This function maps the specified resource into memory, allocates and populates the PCI\_MOD structure, and initializes the resource if necessary. When a resource is no longer needed *pciclose()* should be called to free the system resources and make the resource available to other callers.

### Return Values

Return	errno	Meaning
NULL	EBADF	Invalid <i>devname</i>
NULL	EBUSY	Resource already open in exclusive mode or exclusive mode specified and resource already open
NULL	ENXIO	Cannot access baseboard
NULL	ENODEV	Board not configured or resource does not exist
NULL	ENOMEM	Memory allocation error
NULL	EINVAL	Invalid board type
NULL	EDEADLK	Version mismatch
Other	No Change	Pointer to open resource

## 9.1.66 pciopenex

### Summary

Opens a resource with specific flags or options.

### Function Prototype

```
PCI_MOD * pciopenex( char * devname, Name of resource to open
                    int flags) Options
```

### Description

This function opens a DPT resource. The function maps the specified resource into memory, allocates and populates the PCI\_MOD structure, and initializes the resource if necessary. When a resource is no longer needed `pciclose()` should be called to free the system resources and make the resource available to other callers.

The *devname* consists of an optional “/dev/” prefix, either “dpt” or “pci”, a board number (small integer starting from 0), and optionally a resource specifier. The resource specifier may be ‘a’ for Module A, ‘b’ for Module B, ‘g’ or ‘h’ for global memory, and ‘q’ or omitted for the baseboard.

Possible flags include:

```
PCI_OPEN_EXCL,
PCI_OPEN_NEXCL,
PCI_OPEN_UNCONFIG,
PCI_OPEN_NO_EXP,
PCI_OPEN_NO_EEROM,
PCI_OPEN_NO_MMAP and
PCI_OPEN_IGNORE_INIT_FLAG.
```

If `PCI_OPEN_EXCL` is specified the resource is opened in exclusive mode. If `PCI_OPEN_NEXCL` is specified the resource is opened in non-exclusive mode. If neither is specified the baseboard resource is opened non-exclusively, all other resources are opened exclusively.

`PCI_OPEN_UNCONFIG`, `PCI_OPEN_NO_EXP`, `PCI_OPEN_NO_EEROM` and `PCI_OPEN_NO_MMAP` are intended for use by CAC utilities and diagnostics only.

`PCI_OPEN_IGNORE_INIT_FLAG` that can be passed to the `pciopenex` function. It allows a program to open a board that has not been initialized with `dpinit` since it was powered on. Without the flag `pciopenex` will fail for uninitialized boards and set `errno` to `UNINITIALIZE_BOARD`.

Both `UNINITIALIZE_BOARD` and `PCI_OPEN_IGNORE_INIT_FLAG` are defined in `dputil.h`.

**Return Values**

<b>Return</b>	<b>errno</b>	<b>Meaning</b>
NULL	EBADF	Invalid <i>devname</i>
NULL	EBUSY	Resource already open in exclusive mode or exclusive mode specified and resource already open
NULL	ENXIO	Cannot access baseboard
NULL	ENODEV	Board not configured or resource does not exist
NULL	ENOMEM	Memory allocation error
NULL	EINVAL	Invalid board type
NULL	EDEADLK	Version mismatch
NULL	UNINITIALIZE_BOARD	pciopen, pci_open_chanex and pci_open_chan will always fail for uninitialized boards.
Other	No Change	Pointer to open resource

### 9.1.67 pci\_open\_chan

#### Summary

Opens a telecom channel.

#### Function Prototype

```
PCI_CHAN * pci_open_chan(  int      pci_id,      Board number
                          char      framer_id,    Connector or framer ID
                          int      nslots,       Number of timeslots
                          int *     slotlist,    List of timeslots
                          uint32_t  flags,       Open mode flags
                          uint32_t  framer_modes) Desired framer modes
```

#### Description

This function opens and initializes a telecom channel on a DPT device.

The *pci\_id* parameter gives the index of the desired board (the numeric portion of the board name).

The *framer\_id* parameter is 0-3, ASCII '0'-'3', ASCII 'a'-'d', or ASCII 'A'-'D'. If *flags* contains CHAN\_NOMAP *framer\_id* is used directly. Otherwise it is interpreted as a connector ID using *pci\_chan\_log2phys()*. If *flags* contains CHAN\_WRITE *framer\_id* is taken to be a transmit connector. Otherwise it is taken to be a receive connector. If *framer\_mode* specifies a local loopback mode either connector may be specified.

A positive *nslots* parameter specifies the number of timeslots in the channel. The *slotlist* parameter is an array of *nslots* integers specifying the timeslot numbers (starting from zero) to be included in the channel. Timeslots must be in ascending order and no timeslot may occur more than once.

If *nslots* is negative then *slotlist* is interpreted as an integer bitmap indicating the slots to be opened. The least significant bit corresponds to timeslot zero. Any bits that are set to one indicate that the corresponding timeslot should be included in the channel.

The *flags* parameter consists of one or more of: CHAN\_READ or CHAN\_WRITE to specify the direction of the channel, CHAN\_NOMAP to specify that *framer\_id* is not to be interpreted as a connector ID, and CHAN\_NBLOCK to specify that non-blocking I/O is to be used for the channel.

The *framer\_modes* parameter specifies the desired framer modes. If there are no channels open on the framer the given mode will be set. If there are channels open on the given framer the open will only succeed if the current framer mode matches *framer\_mode*. The mode is a bit wise combination of the following flags (from *dpchan.h*):

**Framer Mode flags for E1 Operation**

E1_RX_BYP_JAT	Bypass receive jitter attenuator
E1_RX_UNFRAMED	Bypass receive framer
E1_RX_HDB3	Receive using HDB3 line decoding
E1_RX_CRC_MF	Receive using CRC4 multiframing
E1_RX_CAS_MF	Receive using CAS multiframing
E1_RX_CRC_REFR	Reframe on excessive CRC errors
E1_RX_NO_REFR	Do not reframe after initial sync
E1_RX_FAST_CLK	Receiver is candidate for transmit clock source
E1_TX_DISABLE	Disable transmitter output
E1_TX_HDB3	Transmit using HDB3 line encoding
E1_TX_UNFRAMED	Bypass transmit framing generation
E1_TX_CRC_MF	Transmit using CRC4 multiframing
E1_TX_CAS_MF	Transmit using CAS multiframing
E1_LOOP_SDRAM	Enable SDRAM loopback (processor pre-dual-port)
E1_LOOP_DPRAM	Enable DPRAM loopback (processor post-dual-port)
E1_LOOP_SERIAL	Enable serial loopback (local pre-framer)
E1_LOOP_DIG	Enable digital loopback (local post-framer)
E1_LOOP_PAYLD	Enable payload loopback (remote post-framer)
E1_LOOP_LINE	Enable line loopback (remote pre-framer)

**Framer Mode flags for T1 Operation**

T1_RX_UNFRAMED	Bypass receive framer
T1_RX_B8ZS	Receive using B8ZS line decoding
T1_RX_SF	Receive using standard supeframe mode (a.k.a. D4)
T1_RX_ESF	Receive using extended superframe mode
T1_RX_SLC96F	Receive using SLC96 superframe mode
T1_RX_JPN	Receive using J1 SF or ESF mode
T1_TX_B8ZS	Transmit using B8ZS line encoding
T1_TX_UNFRAMED	Bypass transmit framing generation
T1_TX_SF	Transmit using standard supeframe mode (a.k.a. D4)
T1_TX_ESF	Transmit using extended superframe mode
T1_TX_SLC96F	Transmit using SLC96 superframe mode
T1_TX_JPN	Transmit using J1 SF or ESF mode
T1_LOOP_*	Corresponding E1 loop-back modes

The E1/T1\_LOOP\_SERIAL and E1/T1\_RX\_UNFRAMED flags affect all framers.

**Return Values**

<b>Return</b>	<b>errno</b>	<b>Meaning</b>
NULL	EBUSY	One or more specified timeslots are busy or framer is in use with a different <i>framer_mode</i>
NULL	EINVAL	Invalid Parameter
NULL	ENOMEM	Insufficient Memory
NULL	ENXIO	Framer not available
NULL	ENODEV	Device not available
NULL	ETIMEOUT	Embedded processor did not respond
NULL	EDEADLK	Version mismatch
-1	PORT_CONFIG_ERROR	Framer port not configured for RX or TX as requested.
Other	No Change	Pointer to open channel

### 9.1.68 pci\_open\_chanex

#### Summary

Opens a telecom channel with extended parameters.

#### Function Prototype

```
PCI_CHAN * pci_open_chan(  int      pci_id,      Board number
                          char      framer_id,    Connector or framer ID
                          int      nslots,      Number of timeslots
                          int *     slotlist,    List of timeslots
                          uint32_t  flags,      Open mode flags
                          uint32_t  framer_modes, Desired framer modes
                          chanopen_t * parameters) Extended parameters
```

#### Description

This function performs the same operation and the **pci\_open\_chan** function but provides for additional parameters specified in the **chanopen\_t** structure. Currently there is one parameter, **tx\_start\_delay**.

The **tx\_start\_delay** specifies the number of frames of transmit data that have to be in the buffer before the transmit is started. The default is 8,000 which corresponds to one second. The delay can be made smaller or larger using this method of opening transmit channels. The minimum number of frames is 8, the maximum is 16,000. Below is a code fragment illustrating the use of this function:

```
chanopen_t tx_param;      /* declare the channel parameter structure */
PCI_CHAN *tx_channel;    /* declare the transmit channel handle */

/* Set the start delay in frames.
tx_param.tx_start_delay = 32;

/* Open the transmit channel */
tx_chan = pci_open_chanex(board_nr, framer_id_tx, num_slots,
                          slot_list, CHAN_WRITE, framer_modes, &tx_param);
```

For other parameter usage and return status, see the description of **pci\_open\_chan**.

### 9.1.69 pci\_open\_chan\_rcv\_host\_start

#### Summary

Opens a receive telecom channel but does not start channelization on the specified framer

#### Function Prototype

```

PCI_CHAN * pci_open_chan(  int      pci_id,      Board number
                          char      framer_id,    Connector or framer ID
                          int       nslots,      Number of timeslots
                          int *     slotlist,    List of timeslots
                          uint32_t  flags,       Open mode flags
                          uint32_t  framer_modes) Desired framer modes

```

#### Description

This function opens a receive channel on a DPT device but does not start the receive channelization on the specified framer. The host application must call `pci_start_receive()` to start receive channelization and read data.

Typically, using `pci_open_chan()`, when the first channel on a framer is opened, the channelization is automatically started for that framer. Using this function, and not starting the channelization on the framer, channels on multiple framers may be opened and then channelization can be simultaneously started of all the framers. Starting the channelization in this manner allows there to be a minimal frame offset between data read from channels on different frames.

For parameter usage and return status, see the description of **pci\_open\_chan**.

### 9.1.70 pci\_pad\_chan

#### Summary

Append any required padding to transmit channel buffer.

#### Function Prototype

```
int pci_pad_chan(      PCI_CHAN   *chan)  Pointer to open channel
```

#### Description

For a transmit channel this function writes an appropriate number of the current channel idle bytes so the total number of bytes in the transmit buffer represents a whole number of frames and a multiple of four bytes. The function returns the number of idle bytes written to the transmit buffer, possibly 0.

This function has no meaning for a receive channel and will return 0.

#### Return Values

Return	errno	Meaning
>= 0	unchanged	Success - the number of idle bytes written to the transmit buffer
-1	Various	An error occurred.

#### Discussion

Padding the transmit buffer is necessary when transmitting bursty data. It ensures that all of the data written to the transmit buffer will be transmitted without gaps. This function provides a convenient method of ensuring that the transmit buffer is properly padded.

However, if the application knows what the current idle byte is and that the data transmitted is of a bursty nature, it may be more efficient for the application to add the appropriate padding to each buffer passed to the `pci_write_chan()` function. To ensure proper padding, the number of bytes written for any message should be a multiple of the least common denominator of 4 and the number of slots used by the channel. The table below shows the least common denominators for the various channel sizes. Padding should be appended to messages that do not meet this criteria.

slots	L.C.D.	slots	L.C.D.	slots	L.C.D.	slots	L.C.D.	slots	L.C.D.
1	4	8	8	15	60	22	44	29	116
2	4	9	36	16	16	23	92	30	60
3	12	10	20	17	68	24	24	31	124
4	4	11	44	18	36	25	100	32	32
5	20	12	12	19	76	26	52		
6	12	13	52	20	20	27	108		
7	28	14	28	21	84	28	28		

**9.1.71 pci\_peek8****Summary**

Reads a single 8-bit value from a resource using mmap addressing.

**Function Prototype**

```
uint8_t pci_peek8( PCI_MOD * pci,      Open resource
                  uint32_t  addr)     Address to read
```

**Description**

This function reads a single 8-bit item using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

There is no error indication available from this function. If error detection is important `pci_peek8a()` should be used instead.

**Return Values**

Return	errno	Meaning
8-bit value	No Change	Success
0	No Change	Success
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Source address not within a valid memory region

**9.1.72 pci\_peek8a****Summary**

Reads an array of 8-bit values from a resource using mmap addressing.

**Function Prototype**

```
int pci_peek8a( PCI_MOD * pci,      Open resource
                uint32_t  addr,     Address to read
                uint32_t  nbytes,   Number of bytes to read
                uint8_t *  buf)     Buffer to fill
```

**Description**

This function reads an array of 8-bit items using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Source range not within a valid memory region

### 9.1.73 pci\_peek16

#### Summary

Reads a single 16-bit value from a resource using mmap addressing.

#### Function Prototype

```
uint16_t pci_peek16( PCI_MOD * pci,      Open resource
                    uint32_t  addr)     Address to read
```

#### Description

This function reads a single 16-bit item using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

There is no error indication available from this function. If error detection is important `pci_peek16a()` should be used instead.

#### Return Values

Return	errno	Meaning
16-bit value	No Change	Success
0	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not 16-bit aligned
0	ENXIO	Source address not within a valid memory region

**9.1.74 pci\_peek16a****Summary**

Reads an array of 16-bit values from a resource using mmap addressing.

**Function Prototype**

```
int pci_peek16a( PCI_MOD * pci,      Open resource
                uint32_t  addr,      Address to read
                uint32_t  nshorts,   Number of 16-bit words to read
                uint16_t * buf)      Buffer to fill
```

**Description**

This function reads an array of 16-bit items using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Source range not within a valid memory region

### 9.1.75 pci\_peek32

#### Summary

Reads a single 32-bit value from a resource using mmap addressing.

#### Function Prototype

```
uint32_t pci_peek32( PCI_MOD * pci,      Open resource
                    uint32_t  addr)     Address to read
```

#### Description

This function reads a single 32-bit item using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

There is no error indication available from this function. If error detection is important `pci_peek32a()` should be used instead.

#### Return Values

Return	errno	Meaning
32-bit value	No Change	Success
0	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not 32-bit aligned
0	ENXIO	Source address not within a valid memory region

**9.1.76 pci\_peek32a****Summary**

Reads an array of 32-bit values from a resource using mmap addressing.

**Function Prototype**

```
int pci_peek32a( PCI_MOD * pci,      Open resource
                uint32_t  addr,     Address to read
                uint32_t  nlongs,   Number of 32-bit words to read
                uint32_t * buf)     Buffer to fill
```

**Description**

This function reads an array of 32-bit items using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 32-bit boundary
0	ENXIO	Source range not within a valid memory region

**9.1.77 pci\_poke8****Summary**

Writes a single 8-bit value to a resource using mmap addressing.

**Function Prototype**

```
int pci_poke8( PCI_MOD * pci,      Open resource
               uint32_t  addr,     Address to write
               uint8_t   val)      Value to write
```

**Description**

This function writes a single 8-bit item using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Source address not within a valid memory region

**9.1.78 pci\_poke8a****Summary**

Writes an array of 8-bit values to a resource using mmap addressing.

**Function Prototype**

```
int pci_poke8a( PCI_MOD * pci,      Open resource
                uint32_t  addr,     Address to write
                uint32_t  nbytes,   Number of bytes to write
                uint8_t *  buf)     Data to write
```

**Description**

This function writes an array of 8-bit items using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Source range not within a valid memory region

**9.1.79 pci\_poke16****Summary**

Writes a single 16-bit value to a resource using mmap addressing.

**Function Prototype**

```
int pci_poke16( PCI_MOD * pci,      Open resource
                uint32_t  addr,     Address to write
                uint16_t  val)      Value to write
```

**Description**

This function writes a single 16-bit item using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Source address not within a valid memory region

**9.1.80 pci\_poke16a****Summary**

Writes an array of 16-bit values to a resource using mmap addressing.

**Function Prototype**

```
int pci_poke16a( PCI_MOD * pci,      Open resource
                uint32_t  addr,      Address to write
                uint32_t  nshorts,    Number of 16-bit words to write
                uint16_t * buf)       Data to write
```

**Description**

This function writes an array of 16-bit items using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Source range not within a valid memory region

**9.1.81 pci\_poke32****Summary**

Writes a single 32-bit value to a resource using mmap addressing.

**Function Prototype**

```
int pci_poke32( PCI_MOD * pci,      Open resource
                uint32_t  addr,     Address to write
                uint32_t  val)      Value to write
```

**Description**

This function writes a single 32-bit item using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Source address not within a valid memory region

**9.1.82 pci\_poke32a****Summary**

Writes an array of 32-bit values to a resource using mmap addressing.

**Function Prototype**

```
int pci_poke32a( PCI_MOD * pci,      Open resource
                uint32_t  addr,      Address to write
                uint32_t  nlongs,    Number of 32-bit words to write
                uint32_t * buf)      Data to write
```

**Description**

This function writes an array of 32-bit items using mmap addressing. It is primarily intended for use within the library and by CAC diagnostics and utilities.

**Return Values**

Return	errno	Meaning
1	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 32-bit boundary
0	ENXIO	Source range not within a valid memory region

### 9.1.83 pci\_read\_chan

#### Summary

Reads a receive channel.

#### Function Prototype

```
int pci_read_chan (   PCI_CHAN    *chan,    Pointer to open receive channel
                    uint8_t *   buf,      Buffer to receive data
                    size_t     nbytes)   Number of bytes to read
```

#### Description

This function reads from a receive channel. If the channel is using blocking I/O (see `pci_chan_blocking()`) the function will not return until *nbytes* of data has been read or an error occurs. If the channel is using non-blocking I/O the function will return with only the data that is immediately available, if any.

#### Return Values

Return	errno	Meaning
-1	EBADF	Invalid <i>chan</i>
-1	EBUSY	I/O already in progress on <i>chan</i>
-1	ENOLINK	Framer out of sync
-1	E_OVERFLOW	Buffer overflow
-1	ETIMEDOUT	Timed out waiting for DMA or for data to become available
non-negative	No Change	Amount of data read

**9.1.84 pci\_reg\_rmw****Summary**

Atomically modifies a register.

**Function Prototype**

```
int pci_reg_rmw(  PCI_MOD * pci,      Pointer to open resource
                 uint32_t  addr,     Register (mmap) address
                 uint32_t  clr,     Bits to clear
                 uint32_t  set,     Bits to set
                 uint32_t * old_val) Receives old value if not NULL
```

**Description**

This function atomically modifies the bits within a register. Locking mechanisms are provided to synchronize access to critical registers between all processes on the host and the embedded software. The function operates by obtaining the lock, reading the current value of the register, setting any bits present in *set*, clearing any bits present in *clr*, writing back the updated value, and releasing the lock. If *old\_val* is not NULL it receives the old register value.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid parameter
0	ENXIO	Source range not within a valid memory region
1	No Change	Success

### 9.1.85 `pcireset`

#### Summary

Resets a board.

#### Function Prototype

```
int pcireset( PCI_MOD * pci)  Pointer to open board
```

#### Description

This function puts a DPT board into reset. This is used primarily for diagnostics but may also be useful for error recovery. Before most resources (particularly channelization resources) may be accessed the reset must be released with `pcirun()`.

#### Return Values

Return	errno	Meaning
Non-zero	EINVAL	Invalid parameter
0	No Change	<i>pci</i> was in reset
1	No Change	<i>pci</i> was not in reset

**9.1.86 pcirun****Summary**

Releases board reset.

**Function Prototype**

```
int pcirun( PCI_MOD * pci)  Pointer to open board
```

**Description**

This function releases the reset signal on a DPT board. This is necessary before most resources (particularly channelization resources) may be accessed.

The processor requires some time to complete initialization. This is typically on the order of a second, but may be as long as five seconds or more. A future version of the API will have a mechanism to determine when the processor finishes booting.

This function is called by the initialization script at system boot and should not be necessary in most applications.

**Return Values**

<b>Return</b>	<b>errno</b>	<b>Meaning</b>
Non-zero	EINVAL	Invalid parameter
0	No Change	<i>pci</i> was in reset
1	No Change	<i>pci</i> was not in reset

**9.1.87 pci\_run\_chan****Summary**

Starts, restarts or stops the embedded processor channelization application..

**Function Prototype**

```
int pci_run_chan(  PCI_MOD * pci,  Pointer to open board
                  int      run,    0 stops channelization
                               1 starts or restarts channelization
                  int      wait,   0 does not wait for embedded processor to
                               complete
                               1 waits for embedded processor to complete
```

**Description**

If the board is not currently in a reset state then channelization is halted and the board is put into a reset state. If the run parameter is not zero then the reset state is cleared allowing the embedded processor to restart its application.

The processor requires some time to complete initialization. This is typically on the order of a second, but may be as long as five seconds or more. A future version of the API will have a mechanism to determine when the processor finishes booting.

**Return Values**

Return	errno	Meaning
-1	EINVAL	Invalid parameter
0	No Change	success
1	error code	failure

**9.1.88 pci\_set\_dpbuffer****Summary**

Sets the number of frames to buffer in the board's Dual Port RAM.

**Function Prototype**

```
int pci_set_dpbuffer (    int          pci_id,    Board number
                        int          size)       number frames to buffer
```

**Description**

The function sets the input and output Dual Port RAM buffer size. The size parameter specifies the number of frames to buffer in each direction. The value of *size* must be a multiple of 4. The minimum value is 8 and the maximum is 252. 252 is the default.

Smaller buffer sizes can minimize overall latency at the expense of system efficiency and possible overflow and underflows. This function can only be called when there are no open channels to the DPT4.

**Return Values**

<b>Return</b>	<b>Error Number</b>	<b>Meaning</b>
-1	EINVAL	Invalid Parameter
8..252	No Change	Previous value of the Dual Port Ram Buffer Size

### 9.1.89 pci\_sleep

#### Summary

Platform independent, thread safe sleep function.

#### Function Prototype

```
int pci_sleep( unsigned int seconds) Time to sleep
```

#### Description

This function suspends the calling thread's execution for *seconds* seconds. Receipt of a signal can cause the function to return prematurely, in which case the return value indicates the number of seconds remaining to sleep.

#### Return Values

Return	errno	Meaning
Non-negative	No Change	Remaining sleep time

### 9.1.90 pci\_start\_receive

#### Summary

Enables the receive channelization hardware logic.

#### Function Prototype

```
int  pci_start_receive(  int          pciId,          Board number
                        uint32_t    frameIdMask,    Bit mask of framer Id's
                        uint32_t    flags)          Open mode flags
```

#### Description

This function is used to enable the DPT4 receive channelization hardware. This function allows the receive channelization hardware logic of the framers indicated in the *framerIdMask* to be started at the same time. Starting the receive channelization hardware at the same times minimizes the frame offset between channels. This function and `pci_open_rcv_host_start` allow a user application to receive data from multiple framers with a minimal frame offset between channels on different framers. A typical application for these functions would be for receiving data from 2 or more framers where the incoming data on the different framers are synchronize together.

The *pciId* parameter gives the index of the desired board (the numeric portion of the board name).

The *framerIdMask* parameter is a bit mask where bit 0 corresponds to framer 0, bit 1 corresponds to framer 1 .... If *flags* contains `CHAN_NOMAP` the framers in the *framerIdMask* are used directly. Otherwise they are interpreted as a connector ID using `pci_chan_log2phys()`.

The *flags* parameter is the same *flags* parameter used in the `pci_open_chan()`. Only the `CHAN_NOMAP` flag is tested to determine the interpretation of the *framerIdMask*.

#### Return Values

Return	errno	Meaning
FALSE	EINVAL	Invalid parameter
TRUE	No Change	Success

**9.1.91 pci\_up\_a8b****Summary**

Reads an array of 8-bit values from a resource.

**Function Prototype**

```
int pci_up_a8b(  PCI_MOD * pci,      Pointer to open resource
                uint32_t  addr,      Source address
                uint32_t  nbytes,    Number of bytes to transfer
                uint8_t *  data)     Pointer to buffer to fill
```

**Description**

This function reads an array of 8-bit items from a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer is from the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer is from the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Source range does not fit within a valid memory region
1	No Change	Success

**9.1.92 pci\_up\_a16b****Summary**

Reads an array of 16-bit values from a resource.

**Function Prototype**

```
int pci_up_a16b( PCI_MOD * pci,      Pointer to open resource
                uint32_t  addr,      Source address
                uint32_t  nshorts,   Number of 16-bit words to transfer
                uint16_t * data)     Pointer to buffer to fill
```

**Description**

This function reads an array of 16-bit items from a resource. If *addr* is greater than or equal to *pci->config.mod\_par\_min* and less than *pci->config.mod\_par\_max* the transfer is from the resource's memory. If *addr* is greater than or equal to *pci->config.gmem\_par\_min* and less than *pci->config.gmem\_par\_max* the transfer is from the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Source range does not fit within a valid memory region
1	No Change	Success

**9.1.93 pci\_up\_a32b****Summary**

Reads an array of 32-bit values from a resource.

**Function Prototype**

```
int pci_up_a32b( PCI_MOD * pci,      Pointer to open resource
                uint32_t  addr,      Source address
                uint32_t  nlongs,    Number of 32-bit words to transfer
                uint32_t * data)     Pointer to buffer to fill
```

**Description**

This function reads an array of 32-bit items from a resource. If *addr* is greater than or equal to *pci->config.mod\_par\_min* and less than *pci->config.mod\_par\_max* the transfer is from the resource's memory. If *addr* is greater than or equal to *pci->config.gmem\_par\_min* and less than *pci->config.gmem\_par\_max* the transfer is from the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 32-bit boundary
0	ENXIO	Source range does not fit within a valid memory region
1	No Change	Success

**9.1.94 pci\_up\_cfg8****Summary**

Reads an 8-bit value from PCI config space using big-endian addressing.

**Function Prototype**

```
int pci_up_cfg8(  PCI_MOD * pci,      Pointer to open resource
                 uint32_t  off,      Offset in config space
                 uint32_t * datap)   Buffer to fill
```

**Description**

This function reads a single 8-bit value from the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses big-endian addressing compatible with CAC expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions (`pci_up_cfg8le` to read 8-bit data).

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

**9.1.95 pci\_up\_cfg8le****Summary**

Reads an 8-bit value from PCI config space using little-endian addressing.

**Function Prototype**

```
int pci_up_cfg8le(  PCI_MOD *   pci,      Pointer to open resource
                  uint32_t   off,      Offset in config space
                  uint32_t *  datap)   Buffer to fill
```

**Description**

This function reads a single 8-bit value from the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses little-endian addressing compatible with standard PCI interfaces including the host and expansion bridges on the baseboard and the bridges on certain expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions.

**Return Values**

<b>Return</b>	<b>errno</b>	<b>Meaning</b>
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

### 9.1.96 pci\_up\_cfg16

#### Summary

Reads a 16-bit value from PCI config space using big-endian addressing.

#### Function Prototype

```
int pci_up_cfg16(  PCI_MOD * pci,      Pointer to open resource
                  uint32_t off,      Offset in config space
                  uint32_t * datap)  Buffer to fill
```

#### Description

This function reads a single 16-bit value from the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF. Misaligned offsets are silently coerced to 16-bit alignment.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses big-endian addressing compatible with CAC expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions (`pci_up_cfg16le` to read 16-bit data).

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

### 9.1.97 pci\_up\_cfg16le

#### Summary

Reads a 16-bit value from PCI config space using little-endian addressing.

#### Function Prototype

```
int pci_up_cfg16(  PCI_MOD *   pci,      Pointer to open resource
                  uint32_t   off,      Offset in config space
                  uint32_t *  datap)    Pointer to data to write
```

#### Description

This function reads a single 16-bit value from the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF. Misaligned offsets are silently coerced to 16-bit alignment.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function uses little-endian addressing compatible with standard PCI interfaces including the host and expansion bridges on the baseboard and the bridges on certain expansion modules. The host and expansion bridges' config space should always be accessed through the little-endian config space functions. The data is in host byte order.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

**9.1.98 pci\_up\_cfg32****Summary**

Reads a 32-bit value from PCI config space.

**Function Prototype**

```
int pci_up_cfg32(  PCI_MOD *   pci,      Pointer to open resource
                  uint32_t   off,      Offset in config space
                  uint32_t *  datap)   Pointer to data to write
```

**Description**

This function reads a single 32-bit value from the PCI config space registers for a board or module. If called for a module, that module's config space is accessed at offsets from 0 to 0xFF and the expansion bridge's config space is accessed at offsets from 0x100 to 0x1FF. If called for a board the host bridge config space is accessed at offsets from 0 to 0xFF, the expansion bridge's config space at offsets from 0x100 to 0x1FF, Module A's config space at offsets from 0x200-0x2FF, and Module B's config space at offsets from 0x300-0x3FF. Misaligned offsets are silently coerced to 32-bit alignment.

Changing config space settings can cause the module, board, or system to stop responding. These functions are primarily intended for use by other library code and by CAC provided diagnostics and utilities.

This function is endian-neutral and may be used to access any device's config space. The data is in host byte order.

**Return Values**

Return	errno	Meaning
0	EINVAL	Invalid argument
0	ENXIO	Addressed device not accessible
Non-zero	No Change	Success

### 9.1.99 pci\_up\_i8b

#### Summary

Reads a single 8-bit value from a DPT resource.

#### Function Prototype

```
uint8_t pci_up_i8b( PCI_MOD * pci,      Pointer to open resource
                   uint32_t addr)      Source address
```

#### Description

This function reads a single 8-bit item from a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer is from the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer is from the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

There is no error indication available from this function. If error detection is important *pci\_up\_a8b()* should be used instead.

#### Return Values

Return	errno	Meaning
8-bit value	No Change	Success
0	No Change	Success
0	EINVAL	Invalid <i>pci</i>
0	ENXIO	Source address not within a valid memory region

**9.1.100 pci\_up\_i16b****Summary**

Reads a single 16-bit value from a DPT resource.

**Function Prototype**

```
uint16_t pci_up_i16b(  PCI_MOD * pci,      Pointer to open resource
                      uint32_t  addr)    Source address
```

**Description**

This function reads a single 16-bit item from a resource. If *addr* is greater than or equal to *pci→config.mod\_par\_min* and less than *pci→config.mod\_par\_max* the transfer is from the resource's memory. If *addr* is greater than or equal to *pci→config.gmem\_par\_min* and less than *pci→config.gmem\_par\_max* the transfer is from the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

There is no error indication available from this function. If error detection is important *pci\_up\_a16b()* should be used instead.

**Return Values**

Return	errno	Meaning
8-bit value	No Change	Success
0	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 16-bit boundary
0	ENXIO	Source address not within a valid memory region

**9.1.101 pci\_up\_i32b****Summary**

Reads a single 32-bit value from a DPT resource.

**Function Prototype**

```
uint32_t pci_up_i32b( PCI_MOD * pci,      Pointer to open resource
                    uint32_t  addr)     Source address
```

**Description**

This function reads a single 32-bit item from a resource. If *addr* is greater than or equal to *pci*→*config.mod\_par\_min* and less than *pci*→*config.mod\_par\_max* the transfer is from the resource's memory. If *addr* is greater than or equal to *pci*→*config.gmem\_par\_min* and less than *pci*→*config.gmem\_par\_max* the transfer is from the board's global memory. Note that some modules may have holes in module memory. Attempting to access addresses within these holes returns an error.

There is no error indication available from this function. If error detection is important *pci\_up\_a32b()* should be used instead.

**Return Values**

Return	errno	Meaning
8-bit value	No Change	Success
0	No Change	Success
0	EINVAL	Invalid <i>pci</i> or <i>addr</i> not on a 32-bit boundary
0	ENXIO	Source address not within a valid memory region

### 9.1.102 pci\_usleep

#### Summary

Platform independent, thread safe fine-grained sleep function.

#### Function Prototype

```
int pci_sleep( useconds_t useconds)    Time to sleep
```

#### Description

This function suspends the calling thread's execution for *useconds* microseconds. Receipt of a signal can cause the function to return prematurely, in which case the return value indicates the number of seconds remaining to sleep.

Note that the granularity of the sleep time is limited by the characteristics of the system clock. If *useconds* is zero the function will return immediately. Otherwise it will block for at least *useconds* microseconds, possibly much longer. In practice the minimum sleep time is on the order of milliseconds, often tens of milliseconds.

#### Return Values

Return	errno	Meaning
-1	EINTR	A signal was caught before the sleep time expired
-1	EINVAL	<i>useconds</i> is less than 0 or greater than 1,000,000
0	No Change	Blocked for specified time

### 9.1.103 pci\_version\_checking

#### Summary

Sets the version checking options.

#### Function Prototype

```
int    pci_version_checking (
        int    board_number,
        int    require_match
        int    report_mismatch
```

#### Description

The version checking options required match and report mismatches options are set. The required match option applies to all boards. If it is non zero pciopen and pci\_open\_chan will fail if a version mismatch occurs. The report mismatches option is kept on a per board basis for each program. If it is non zero when a mismatch is detected a line of output indicating the mismatch will be written to stderr. The report mismatches option is kept on a per board basis so that once a report has been printed it can be turned off for that board. The two options are treated independently. Reports will still be generated if a mismatch is detected but the require match option is zero.

#### Return Values

Return	errno	Meaning
0	EINVAL	Invalid Argument
1	No Change	Success

**9.1.104 pci\_write\_chan****Summary**

Writes a transmit channel.

**Function Prototype**

```
int pci_write_chan ( PCI_CHAN * chan,      Pointer to open transmit channel
                   uint8_t *  buf,       Data to send
                   size_t     nbytes)    Number of bytes to write
```

**Description**

This function writes to a transmit channel. If the channel is using blocking I/O (see `pci_chan_blocking()`) the function will not return until *nbytes* of data has been written or an error occurs. If the channel is using non-blocking I/O the function will only write data into the immediately available buffer space, if any.

**Return Values**

<b>Return</b>	<b>errno</b>	<b>Meaning</b>
-1	EBADF	Invalid <i>chan</i>
-1	EBUSY	I/O already in progress on <i>chan</i> or channel loop source set for <i>chan</i>
-1	E_OVERFLOW	Buffer underflow occurred
-1	ETIMEDOUT	Timed out waiting for buffer space to become available
non-negative	No Change	Amount of data written

## 10 Expansion and TDM API Functions

This section describes the API library functions for the smPCI expansion interface and TDM subsystems for use with smPCI expansion modules on DPT boards equipped with the expansion interface.

### 10.1 smPCI Module Functions

Functions for transferring data between the host and smPCI modules are described in section 9. These functions include:

- pci\_dl\_i8b(),
- pci\_dl\_i16b(),
- pci\_dl\_i32b(),
- pci\_dl\_a8b(),
- pci\_dl\_a16b(),
- pci\_dl\_a32b(),
- pci\_up\_i8b(),
- pci\_up\_i16b(),
- pci\_up\_i32b(),
- pci\_up\_a8b(),
- pci\_up\_a16b() and
- pci\_up\_a32b().

This section describes other generic functions for the DPT expansion interface. API functions for specific smPCI modules are described in the manuals for each module type.

### 10.1.1 pci\_get\_module\_type

#### Summary

Determine the type of smPCI module installed.

#### Function Prototype

```
int pci_get_module_type(  PCI_MOD *   pci,           Open DPT board resource
                        int          mod_num,      Module location
                        uint32_t *   type)        Pointer to storage for type
```

#### Description

This function determines the type of smPCI module installed on one of the DPT expansion module locations. The *pci* argument is a PCI\_MOD pointer obtained from the pciopen() function. The *mod\_num* argument specifies the module location, 0 for module A or 1 for module B. The *type* argument is a pointer to a variable where the function will store the module type.

The value stored in *type* will be one of the smPCI module type macros defined in <dputil.h>, such as PCI\_DM5420, PCI\_DM12C549, etc.

#### Return Values

Return	errno	Meaning
1	unchanged	The module type was determined and stored in <i>type</i>
0	various	No module or unknown module installed on the specified location
-1	EINVAL	Invalid PCI_MOD or module number, <i>type</i> unchanged
-1	various	Error occurred accessing hardware, <i>type</i> unchanged

## 10.1.2 pci\_load\_code, pci\_verify\_code, pci\_read\_code

### Summary

Download, verify and/or read executable code for smPCI processor modules.

### Function Prototypes

```
int pci_load_code(    PCI_MOD * pci,      Open smPCI module resource
                    char *   codename,  Name of code file
                    int *   codetype)  Type of code file

int pci_verify_code( PCI_MOD * pci,      Open smPCI module resource
                    char *   codename,  Name of code file
                    int *   codetype)  Type of code file

int pci_read_code(   PCI_MOD * pci,      Open smPCI module resource
                    char *   codename,  Name of code file
                    int *   codetype)  Type of code file
```

### Description

These functions handle executable code files for smPCI processor modules. All three functions read and parse the specified code file and save symbol table information in the PCI\_MOD structure. The pci\_load\_code() function reads the code and downloads it to the processor. The pci\_verify\_code() function downloads the code and verifies the download. The pci\_read\_code() function does not download the code and is provided to allow an application to access symbol table information for a program that is already running on a processor.

The *codename* argument specifies the name of the code file on the host file system. The *codetype* argument specifies the type of code file using one of the following macros for supported code file types:

MM_COFF	COFF file
MM_ELF	ELF file
MM_SRECORDS	S3 file

### Return Values

Return	errno	Meaning
1	unchanged	The code was successfully read and downloaded, if requested.
0	EINVAL	Invalid PCI_MOD or code type
0	ENOEXEC	The code file is not of the specified type, COFF file header does not match processor type or an error occurred parsing the file contents
0	EIO	Error occurred downloading code or verifying the download
0	various	Error occurred opening the code file

## 10.2 TDM Control Functions

This section describes functions for controlling the Expansion TDM subsystem. The DPT's TDM subsystem provides 4 serial time-division multiplexed busses allowing smPCI modules on the DPT4's expansion interface to read and write telecom data.

Control operations on the TDM subsystem are accomplished through function calls operating on the TDM resource of the DPT board. Access to the TDM resource is provided using the `pci_tdmopen()` and `pci_tdmclose()` functions.

The TDM subsystem runs at a fixed frame rate of 8 kHz (125 microseconds per frame) with 128 time slots of 8 bits. Note that the TDM time slots are numbered from 1 to 128. TDM multi-frame pulses are generated at a rate set by the `pci_tdm_multiframe()` function. The TDM clock source is set using the `pci_tdm_clock_cfg()` or the `pci_hw_fast_clk()` function. These functions allow any of the incoming framers to be chosen as the source for the TDM clock, including a mode in which the fastest incoming framer clock is chosen.

Once the TDM timing has been configured the TDM subsystem is started using the `pci_tdm_run()` function. The `pci_tdm_stop()` function may be used to halt the TDM subsystem.

Data is transferred to and from the TDM busses based on a TDM connection map, which specifies connections to and from each TDM bus during each TDM time slot. The connections are controlled using the map modification functions. The `pci_tdm_src_add()` and `pci_tdm_src_del()` functions are used for devices writing data to TDM busses. The `pci_tdm_dst_add()` and `pci_tdm_dst_del()` functions are used for devices reading data from TDM busses. A device can be both a source and a destination for TDM busses.

The map modification functions make changes to the software copy of the TDM connection map. After a group of connections have been specified the map is updated on the hardware using the `pci_tdm_updatemap()` or `pci_tdm_start()` functions. The `pci_tdm_clrmap()` function may be used to clear both the software and hardware TDM connection maps.

To transmit telecom data from the TDM subsystem use the `pci_framer_tx_source()` function, described in section 9. The `pci_framer_set_mode()` function, also described in section 9, is used to configure the framers when using the TDM subsystem. Incoming telecom data is always available for TDM usage. Channelization and TDM telecom transmission cannot be used at the same time. TDM connections must still be specified for each incoming and outgoing telecom time slot to be connected to or from the TDM busses. For these connections the telecom interfaces, E1/T1 framers and H.100 interfaces, have a fixed mapping between the telecom time slots and TDM time slots. Time slots for the four telecom framers are multiplexed across the TDM frame and the H.100 slots have a direct, 1-to-1 mapping, as shown below.

Framer and H.100 slots to TDM time slot mapping		
E1/T1 Framer Slots	H.100 (A or B) Slots	TDM Time Slot
Framer 0 slot 0	slot 0	1
Framer 1 slot 0	slot 1	2
Framer 2 slot 0	slot 2	3
Framer 3 slot 0	slot 3	4
Framer 0 slot 1	slot 4	5
Framer 1 slot 1	slot 5	6
Framer 2 slot 1	slot 6	7
Framer 3 slot 1	slot 7	8
...	...	...
Framer 0 slot 31	slot 124	125
Framer 1 slot 31	slot 125	126
Framer 2 slot 31	slot 126	127
Framer 3 slot 31	slot 127	128

Only the framer and H.100 time slots required need be added to the TDM connection map. Note that when sending and receiving the same framer or H.100 slot, different TDM busses must be used for the `pci_tdm_src_add()` and `pci_tdm_dst_add()` functions, Unless the goal is to retransmit the received time slot.

A framer status byte is also available as a TDM data source. The status value is updated each frame and consists of the following data:

Status Bit	Indication	Meaning
7	Framer 3 Multiframe	1 indicates the first frame of multi-frame
6	Framer 2 Multiframe	
5	Framer 1 Multiframe	
4	Framer 0 Multiframe	
3	Framer 3 Frame Slip	1 indicates a slipped frame. (a repeated frame when TDM clock is derived from the fastest framer receive clock)
2	Framer 2 Frame Slip	
1	Framer 1 Frame Slip	
0	Framer 0 Frame Slip	

The framer status byte is always available but should only be mapped onto TDM time slots 5 through 127 to ensure that it accurately reflects the status for the current TDM frame.

### 10.2.1 pci\_tdmclose

#### Summary

Closes an opened TDM resource.

#### Function Prototype

```
int pci_tdmclose(      PCI_TDM * tdm)  Pointer to open TDM resource structure
```

#### Description

This function closes the TDM resource and deallocates the memory allocated for the PCI\_TDM structure which was allocated using pci\_tdmopen().

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EBADF	The PCI_TDM structure passed in is NULL

## 10.2.2 pci\_tdmopen

### Summary

Opens access to the TDM subsystem resource.

### Function Prototype

```
PCI_TDM * pci_tdmopen( char * tdmname) name of DPT board
```

### Description

This function opens the TDM resource device on the named DPT board. When successful it allocates a PCI\_TDM structure and returns a pointer to it.

The application uses the PCI\_TDM pointer to refer to the TDM resource when calling other TDM functions.

### Return Values

Return	errno	Meaning
NULL	Various	Invalid arguments or error occurred
Other	No Change	Pointer to open TDM resource

### 10.2.3 pci\_tdm\_clock\_cfg

#### Summary

Configures the TDM subsystem clock source.

#### Function Prototype

```
int pci_tdm_clock_cfg( PCI_TDM * tdm,          Pointer to open TDM resource
                    uint32_t framerMode,      Framer Configuration –
                    uint32_t clockSource)     E1_MODE or T1_MODE
                                             TDM clock source
```

#### Description

This function configures the source for the TDM subsystem clock. There are three potential TDM clock sources: TDM\_RX\_FAST\_CLK, TDM\_LOCAL\_CLK, TDM\_DB\_CLK or TDM\_H100\_CLK.

TDM\_LOCAL\_CLK selects the local oscillator on the board. TDM\_H100\_CLK selects the clock derived from the H.100 interface and is only supported on DPT4 boards that have the H.100 interface populated. TDM\_DB\_CLK selects the clock signal generated on one of the expansion modules. This mode is only supported on DPT5 or DPT6 boards and only one of the expansion models should be driving a clock at any one time.

If the source is the TDM\_RX\_FAST\_CLK, then the framers to be included in the receiver fast clock detection must also be provided using the macros, TDM\_RX\_FRAMER0, TDM\_RX\_FRAMER1, TDM\_RX\_FRAMER2 and TDM\_RX\_FRAMER3 (using bit-wise OR). These macros specify the framer number and not the connector number, if the DPT board is configured with connectors other than RJ45 connects (RJ45 connector numbers map directly to framer numbers), then the user application should call pci\_comet\_log2phys() with the is\_tx parameter set to 0 to translate the connector number to a framer number.

For example, to choose the fastest of all 4 framers, specify:

```
TDM_RX_FAST_CLK | TDM_RX_FRAMER0 | TDM_RX_FRAMER1
| TDM_RX_FRAMER2 | TDM_RX_FRAMER3
```

To include just one of the framers, Framer 0 for example, using it's receive clock exclusively, specify:

```
TDM_RX_FAST_CLK | TDM_RX_FRAMER0
```

Note that when a framer has no incoming signal it's receive clock is replaced by the local oscillator on the DPT board. If a framer without an incoming signal is included in the fast clock selection and the local oscillator is running faster than the other receive clocks, the TDM clock will be derived from the local oscillator. It may be desirable to check for incoming framer signals when choosing framers for fast clock selection. This may done using the pci\_comet\_read() function to examine the **LOSV** bit in the **CDRC Interrupt Status** register (bit 0 of register 0x12) for each framer. The **LOSV** bit will be 0 for framers with incoming signal.

This function and the `pci_hw_fast_clk()` and `pci_hw_tdmdb_clk()` functions configure the TDM clock source. Only one of these functions should be included in a user application. Note that this function does not configure the transmit clock for the framers.

#### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer or invalid parameter
0	various	Error occurred configuring hardware
1	No Change	Success

## 10.2.4 pci\_tdm\_clrmap

### Summary

Clears the TDM connection map.

### Function Prototype

```
int pci_tdm_clrmap( PCI_TDM * tdm)    Pointer to open TDM resource structure
```

### Description

This function clears the TDM connection map on the DPT board referenced by the PCI\_TDM pointer. Both the software copy of the map (stored in the PCI\_TDM structure) and the TDM map RAM on the board are cleared. Clearing the map sets all bits in the map to 1.

This function should be used whenever an application needs to initialize the TDM connectivity on the board.

### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer
0	various	Error occurred configuring hardware
1	No Change	Success

## 10.2.5 pci\_tdm\_dst\_add

### Summary

Add a data destination to the TDM connection map.

### Function Prototype

```
int pci_tdm_dst_add( PCI_TDM * tdm,      Pointer to open TDM
                    int slot,          Time slot for connection
                    int bus,           Bus for connection
                    int dev_type,      Device type to connect
                    int dev_num)      Device number
```

### Description

This function adds a data destination connection to the TDM map on the DPT board referenced by the PCI\_TDM pointer. Changes are made to the software copy of the map. Use pci\_tdm\_run() or pci\_tdm\_updatemap() when a group of changes are to be made current.

### Arguments

*slot* specifies the TDM slot number for the connection. Slot numbers start at 1 and go up to 128.

*bus* specifies which of the four TDM busses the connection should be made to. The bus is designated using one of the macros, TDM\_BUSA, TDM\_BUSB, TDM\_BUSC and TDM\_BUSD. These correspond to integer values 0, 1, 2 and 3.

*dev\_type* specifies the type of device for the connection. The value must be one of the macros for the framers, H.100 interfaces or one of the supported smPCI modules.

TDM_DEVFRAMER	DPT E1 or T1 framers
TDM_DEVH100A	DPT H.100 Bus A
TDM_DEVH100B	DPT H.100 Bus B
TDM_DEVDM5420	smPCI 5420 DSP module

*dev\_num* specifies module location and unit number for smPCI modules.

For TDM\_DEVFRAMER, TDM\_DEVH100A or TDM\_DEVH100B *dev\_num* should be 0.  
For TDM\_DEVDM5420 *dev\_num* specifies the module location using the values 0 for module A and 1 for module B.

### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer or other invalid argument value
0	ENODEV	The requested device type is not installed
1	No Change	Success

## 10.2.6 pci\_tdm\_dst\_del

### Summary

Delete a data destination to the TDM connection map.

### Function Prototype

```
int pci_tdm_dst_del(  PCI_TDM *  tdm,      Pointer to open TDM
                    int          slot,    Time slot for connection
                    int          dev_type, Device type to connect
                    int          dev_num) Device number
```

### Description

This function removes a data destination connection from the TDM map on the DPT board referenced by the PCI\_TDM pointer. Changes are made to the software copy of the map. Use pci\_tdm\_run() or pci\_tdm\_updatemap() when a group of changes are to be made current.

### Arguments

Since a device can only be a TDM destination from one bus during a given time slot, *t*, only the slot number and device information is required.

*slot* specifies the TDM slot number for the connection. Slot numbers start at 1 and go up to 128.

*dev\_type* specifies the type of device for the connection. The value must be one of the macros for the framers, H.100 interfaces or one of the supported smPCI modules.

TDM_DEVFRAMER	DPT E1 or T1 framers
TDM_DEVH100A	DPT H.100 Bus A
TDM_DEVH100B	DPT H.100 Bus B
TDM_DEVDM5420	smPCI 5420 DSP module

*dev\_num* specifies module location and unit number for smPCI modules.

For TDM\_DEVFRAMER, TDM\_DEVH100A or TDM\_DEVH100B *dev\_num* should be 0.  
For TDM\_DEVDM5420 *dev\_num* specifies the module location using the values 0 for module A and 1 for module B.

### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer or other invalid argument value
0	ENODEV	The requested device type is not installed
1	No Change	Success

## 10.2.7 pci\_tdm\_getmap

### Summary

Read and store the current TDM connection map.

### Function Prototype

```
int pci_tdm_getmap( PCI_TDM * tdm,      Pointer to open TDM resource
                   uint8_t * mapbuf,   Buffer to store map data
                   int nslots)        Number of slots in map
```

### Description

This function reads the current TDM connection map from the DPT board referenced by the PCI\_TDM pointer. It stores the map data in the software map buffer in the PCI\_TDM structure.

The *nslots* argument specifies the size of the TDM map to read. The TDM subsystem uses a fixed TDM frame with 128 slots, *nslots* should be specified as 128.

If the application requires a separate copy of the map data, the *mapbuf* argument should point to enough memory to store the data. The map data requires 8 bytes per time slot so the total required space is 1024 bytes. If a separate copy of the map is not required *mapbuf* should be NULL.

Reading the current map allows the application to make changes to an existing TDM map when the application inherits the TDM resource from a DPT board that is already running an application.

### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer
0	various	Error occurred reading data from the hardware
1	No Change	Success

## 10.2.8 pci\_tdm\_multiframe

### Summary

Configures the TDM subsystem clock source.

### Function Prototype

```
int pci_tdm_multiframe(  PCI_TDM *  tdm,          Pointer to open TDM resource
                        int          nframes)     structure
                                                Number of frames per multi-frame
```

### Description

This function configures the number of TDM frames per TDM multi-frame. Valid values for *nframes* are from 0 to 32. A value of 0 disables the TDM multi-frame sync pulses. Normally this should be set based on the type of E1 or T1 multi-frame structure being used.

### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer or Invalid parameter
0	various	Error occurred configuring hardware
1	No Change	Success

## 10.2.9 pci\_tdm\_run

### Summary

Start the TDM subsystem.

### Function Prototype

```
int pci_tdm_run(      PCI_TDM * tdm)  Pointer to open TDM resource structure
```

### Description

This function starts the TDM subsystem on the DPT board referenced by the PCI\_TDM pointer. When TDM framing is derived from incoming E1 or T1 framers, the TDM system will start synchronously with the next frame boundary. If the TDM connection map is not current on the board, it will be updated to match the software copy of the connection map.

### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer
0	various	Error occurred configuring hardware
1	No Change	Success

### 10.2.10 pci\_tdm\_src\_add

#### Summary

Add a data source to the TDM connection map.

#### Function Prototype

```
int pci_tdm_src_add( PCI_TDM * tdm,      Pointer to open TDM
                                resource structure
                                int slot,    Time slot for connection
                                int bus,     Bus for connection
                                int dev_type, Device type to connect
                                int dev_num) Device number
```

#### Description

This function adds a data source connection to the TDM map on the DPT board referenced by the PCI\_TDM pointer. Changes are made to the software copy of the map. Use pci\_tdm\_run() or pci\_tdm\_updatemap() when a group of changes are to be made current.

#### Arguments

*slot* specifies the TDM slot number for the connection. Slot numbers start at 1 and go up to 128.

*bus* specifies which of the four TDM busses the connection should be made to. The bus is designated using one of the macros, TDM\_BUSA, TDM\_BUSB, TDM\_BUSC and TDM\_BUSD. These correspond to integer values 0, 1, 2 and 3.

*dev\_type* specifies the type of device for the connection. The value must be one of the macros for the framer s, H.100 interfaces or one of the supported smPCI modules.

TDM_DEVFRAMER	DPT E1 or T1 framers
TDM_DEVH100A	DPT H.100 Bus A
TDM_DEVH100B	DPT H.100 Bus B
TDM_DEVDM5420	smPCI 5420 DSP module

*dev\_num* specifies options for DPT framer connections or module location and unit number for smPCI modules.

For TDM\_DEVFRAMER *dev\_num* should be 0 to write incoming framer data to the TDM time slot or the macro TDM\_FRAMER\_STATUS to write status information to the TDM time slot.

For TDM\_DEVH100A or TDM\_DEVH100B *dev\_num* should be 0.

For TDM\_DEVDM5420 *dev\_num* specifies the module location using the values 0 for module A and 1 for module B.

#### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer or other invalid argument value
0	ENODEV	The requested device type is not installed
1	No Change	Success

### 10.2.11 pci\_tdm\_src\_del

#### Summary

Delete a data source to the TDM connection map.

#### Function Prototype

```
int pci_tdm_src_del(  PCI_TDM *  tdm,      Pointer to open TDM
                    int         slot,    Time slot for connection
                    int         bus,     Bus for connection
```

#### Description

This function removes a data source connection from the TDM map on the DPT board referenced by the PCI\_TDM pointer. Changes are made to the software copy of the map. Use pci\_tdm\_run() or pci\_tdm\_updatemap() when a group of changes are to be made current.

#### Arguments

Since there can only be one device sourcing a given TDM bus during a given TDM time slot, only the slot number and bus need be specified

*slot* specifies the TDM slot number for the connection. Slot numbers start at 1 and go up to 128.

*bus* specifies which of the four TDM busses the connection should be removed from. The bus is designated using one of the macros, TDM\_BUSA, TDM\_BUSB, TDM\_BUSC and TDM\_BUSD. These correspond to integer values 0, 1, 2 and 3.

#### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer or other invalid argument value
1	No Change	Success

### 10.2.12 pci\_tdm\_stop

#### Summary

Stop the TDM subsystem.

#### Function Prototype

```
int pci_tdm_stop( PCI_TDM * tdm)    Pointer to open TDM resource structure
```

#### Description

This function stops the TDM subsystem on the DPT board referenced by the PCI\_TDM pointer.

#### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer
0	various	Error occurred configuring hardware
1	No Change	Success

### 10.2.13 pci\_tdm\_updatemap

#### Summary

Update the TDM connection map on the board.

#### Function Prototype

```
Int pci_tdm_updatemap( PCI_TDM * tdm)    Pointer to open TDM resource structure
```

#### Description

This function updates the TDM connection map memory on the board to match the software copy. Changes made to the connection map using the functions to add or delete sources and destinations are made to the software copy of the connection map. Only when pci\_tdm\_updatemap() is called do the changes actually written to the hardware connection map memory.

When the TDM subsystem is not running the new connection map takes effect immediately. When the TDM subsystem is running, the new map takes effect on the next frame boundary.

If the map is already current, the function returns without modifying the hardware.

#### Return Values

Return	Errno	Meaning
0	EINVAL	NULL TDM resource pointer
0	various	Error occurred configuring hardware
1	No Change	Success (even if no update was required)

### 10.3 H.100 Control Functions

This section describes functions for controlling the H.100 interface. The DPT's H.100 interface provides access to an external H.100 bus (see the H.100 spec for more detail). The H.100 bus consists of 32 bi-directional serial TDM buses ("CT" buses) each running at 8.192 Mb/s, with 128 8-bit time slots per 8 KHz frame. The interface is implemented using an OKI Semiconductor ML53812-2 device (see the data sheet for more detail).

The OKI device has two local serial TDM buses, both running with the same timing as the H.100 buses as described above. The OKI device is highly configurable, and allows interconnection between any timeslot of any of the 32 H.100 buses with any timeslot of the local buses.

In the DPT4, the two local buses are referred to as buses A and B, and are connected to the DPT4 TDM subsystem through the expansion FPGA. This FPGA provides the functionality for interconnections between the A & B buses and the E1/T1 framers and Expansion modules via the TDM subsystem buses.

In general, setting up the H.100 interface involves:

- Initializing and setting up various timing modes using the API functions:
  - h100\_init()
  - h100\_master\_pll()
  - h100\_master()
  - h100\_compat()
  - h100\_netref()
  - h100\_slave()
- Setting up connections using the API functions:
  - h100\_route\_write()
  - h100\_route\_source()
  - h100\_route\_lcl()
  - h100\_par\_write()
  - h100\_par\_read()

These settings typically follow setting up the TDM subsystem (as described in the previous section). The following sections describe library functions which are provided for configuration of the H.100 interface. Programmers should also refer to the data sheet for the ML53812-2 H.100 interface device.

### 10.3.1 h100\_init

#### Summary

Initializes the H.100 controller device.

#### Function Prototype

```
int h100_init(          PCI_MOD * pci)  Pointer to open baseboard
```

#### Description

This function resets the device, checks its ID, and writes initialization values to its registers.

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	No change	Failure

### 10.3.2 h100\_master\_pll

#### Summary

Configures the H.100 controller PLL.

#### Function Prototype

```
int h100_master_pll(  PCI_MOD * pci,      Pointer to open baseboard
                    int      mode,      PLL mode
                    int      ref,       PLL reference
                    int      ref_freq)  PLL reference Frequency
```

#### Description

This function sets the mode of the PLL.

#### Arguments

<i>mode</i>	specifies the PLL mode	
	H100_MASTER_PLL_NORMAL	normal
	H100_MASTER_PLL_HOLDOVER	holdover
	H100_MASTER_PLL_FREE_RUN	free run
	H100_MASTER_PLL_AUTO_TO_HOLDOVER	switch on error
	H100_MASTER_PLL_AUTO_TO_FREE_RUN	switch on error
<i>ref</i>	specifies the PLL reference.	
	H100_MASTER_PLL_REFER_NONE	none
	H100_MASTER_PLL_REFER_CT_NETREF_1	H.100
	H100_MASTER_PLL_REFER_L_NETREF_0	local
<i>ref_freq</i>	specifies the PLL reference frequency	
	H100_MASTER_PLL_FREQ_8KHZ	8 KHz
	H100_MASTER_PLL_FREQ_1536KHZ	1536 KHz
	H100_MASTER_PLL_FREQ_1544KHZ	1544 KHz
	H100_MASTER_PLL_FREQ_2048KHZ	2048 KHz

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.3 h100\_master

#### Summary

Configures the H.100 controller bus mastering.

#### Function Prototype

```
int h100_master(      PCI_MOD * pci,      Pointer to open baseboard
                    int mode,          PLL mode
                    int pri_bus,       Primary bus timing source
                    int advance)       Advance PLL timing
```

#### Description

This function sets the mastering mode of the H.100 device.

#### Arguments

<i>mode</i>	specifies the master mode	
	H100_MASTER_PLL_DISABLE	disabled
	H100_MASTER_PLL_PRIMARY	primary is master
	H100_MASTER_PLL_SECONDARY	secondary is master
	H100_MASTER_PLL_SECONDARY_AUTO	switch on error
<i>pri_bus</i>	specifies the primary bus timing source	
	H100_MASTER_PLL_PRIMARY_BUSA	BUS A
	H100_MASTER_PLL_PRIMARY_BUSB	BUS B
<i>advance</i>	advance PLL timing	
	H100_MASTER_ADVANCE_TIMING_DISABLE	none
	H100_MASTER_ADVANCE_TIMING_ENABLE	advanced

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.4 h100\_compat

**Summary**

Configures the H.100 controller compatibility timing.

**Function Prototype**

```
int h100_compat(    PCI_MOD * pci,           Pointer to open baseboard
                  int    sclk_freq,       SCLK Freq
                  int    sc_master,       SCbus Master
                  int    mvip_master,     MVIP Master
                  int    hmvip_master)    H-MVIP Master
```

**Description**

This function sets the timing compatibility mode of the H.100 device.

**Arguments**

- sclk\_freq* specifies the SCLK frequency
  - H100\_MASTER\_PLL\_DISABLE                    disabled
  - H100\_MASTER\_PLL\_PRIMARY                    primary is master
  - H100\_MASTER\_PLL\_SECONDARY                secondary is master
  - H100\_MASTER\_PLL\_SECONDARY\_AUTO        switch on error
  
- sc\_master* enable SCbus Master
  - H100\_SCBUS\_MASTER\_DISABLE                disabled
  - H100\_SCBUS\_MASTER\_ENABLE                enabled
  
- mvip\_master* enable MVIP Master
  - H100\_MVIP90\_MASTER\_DISABLE              disabled
  - H100\_MVIP90\_MASTER\_ENABLE              enabled
  
- hmvip\_master* enable H\_MVIP Master
  - H100\_HMVIP\_MASTER\_DISABLE                disabled
  - H100\_HMVIP\_MASTER\_ENABLE                enabled

**Return Values**

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.5 h100\_netref

#### Summary

Configures the H.100 controller NETREF output.

#### Function Prototype

```
int h100_netref(    PCI_MOD * pci,          Pointer to open baseboard
                  int    oe,             output enable
                  int    source)         timing source
```

#### Description

This function sets the NETREF mode of the H.100 device.

#### Arguments

<i>oe</i>	enables the NETREF output	
	H100_CT_NETREF_1_OUTPUT_DISABLE	disabled
	H100_CT_NETREF_1_OUTPUT_ENABLE	enabled
<i>source</i>	selects the timing source	
	H100_CT_NETREF_1_SOURCE_NONE	free run
	H100_CT_NETREF_1_SOURCE_L_NETREF0	sync to local netref

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.6 h100\_slave

#### Summary

Configures the H.100 controller slave bus timing.

#### Function Prototype

```
int h100_slave(    PCI_MOD * pci,           Pointer to open baseboard
                  int mode,             mode
                  int ct_auto,          auto switch
                  int ct_src,          mode 0 source
                  int advance)         advance PLL timing
```

#### Description

This function controls slave bus timing of the H.100 device.

#### Arguments

<i>mode</i>	timing mode	
	H100_SLAVE_BUS_MODE_CT_BUS	slave to CT bus
	H100_SLAVE_BUS_MODE_SC_BUS	slave to SC bus
	H100_SLAVE_BUS_MODE_MVIP	slave to MVIP bus
	H100_SLAVE_BUS_MODE_LOCAL	slave to local bus
<i>ct_auto</i>	enable auto timing source switch for mode 0	
	H100_SLAVE_BUS_CT_MANUAL_MODE	manual
	H100_SLAVE_BUS_CT_AUTO_MODE	auto
<i>ct_src</i>	selects timing source for mode 0	
	H100_SLAVE_BUS_CT_A_SELECT	Select A
	H100_SLAVE_BUS_CT_B_SELECT	Select B
<i>advance</i>	advance PLL timing	
	H100_SLAVE_BUS_ADVANCE_TIMING_DISABLE	disabled
	H100_SLAVE_BUS_ADVANCE_TIMING_ENABLE	enabled

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.7 h100\_set\_leds

#### Summary

Controls the H.100 general purpose LEDs.

#### Function Prototype

```
int h100_set_leds( PCI_MOD * pci,          Pointer to open baseboard
                  int      val)          mode
```

#### Description

This function controls the H.100 device LEDs that are located on the DPT baseboard.

#### Arguments

<i>val</i>	LED value	
	bit 0	LED A on or off
	bit 1	LED B on or off
	bit 2-31	n/a

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.8 h100\_route\_write

**Summary**

Configures the H.100 controller routing memory.

**Function Prototype**

```
int h100_route_write(  PCI_MOD * pci,           Pointer to open baseboard
                      int      tdm_slot,      TDM slot
                      int      ct_slot,       H.100 slot
                      int      ct_bus,       H.100 bus
                      int      dir,         direction
                      int      oe)         output enable
```

**Description**

This function controls sets up a connection by writing to the routing memory of the H.100 device.

**Arguments**

*tdm\_slot* Local bus slot. 0-127 selects Bus A, 128-255 selects Bus B  
*ct\_slot* H.100 bus slot 0-127  
*ct\_bus* H.100 bus 0-31

*dir* direction  
       H100\_TRANSMIT           TDM to H.100  
       H100\_RECEIVE           H.100 to TDM

*oe* output enable  
       H100\_OUTPUT\_DISABLE   disable  
       H100\_OUTPUT\_ENABLE   enable

**Return Values**

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.9 h100\_route\_source

#### Summary

Configures the H.100 controller routing data source.

#### Function Prototype

```
int h100_route_source(  PCI_MOD * pci,          Pointer to open baseboard
                       int      tdm_slot,      TDM slot
                       int      dir,          direction
                       int      source)        source
```

#### Description

This function controls sets up the data source for a connection in the H.100 device.

#### Arguments

*tdm\_slot* Local bus slot ID. 0-127 selects Bus A, 128-255 selects Bus B

*dir* direction

H100_TRANSMIT	TDM to H.100
H100_RECEIVE	H.100 to TDM

*source* data source

H100_SOURCE_SERIAL	Local bus
H100_SOURCE_PARALLEL	Internal memory

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

**10.3.10 h100\_route\_lcl****Summary**

Configures the H.100 controller to route a local timeslot to another local timeslot.

**Function Prototype**

```
int h100_route_lcl(      PCI_MOD * pci,          Pointer to open baseboard
                        int      in_slot,       input TDM slot
                        int      out_slot,      output TDM slot
                        int      oe)           direction
```

**Description**

This function controls sets up a local connection in the H.100 device.

**Arguments**

*in\_slot* Local input bus slot ID. 0-127 selects Bus A, 128-255 selects Bus B  
*out\_slot* Local output bus slot ID. 0-127 selects Bus A, 128-255 selects Bus B

*oe* output enable  
       H100\_OUTPUT\_DISABLE   disable  
       H100\_OUTPUT\_ENABLE    enable

**Return Values**

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.11 h100\_par\_write

#### Summary

Write to the H.100 controller parallel access port.

#### Function Prototype

```
int h100_par_write(    PCI_MOD * pci,          Pointer to open baseboard
                    int    tdm_slot,        TDM slot
                    int    dir,            direction
                    int    data)           value
```

#### Description

This function writes to the parallel access port in the H.100 device. A value written to this port will be inserted into the selected bus if it is set to be sourced by the parallel port (see h100\_route\_source).

#### Arguments

*tdm\_slot* Local input bus slot ID. 0-127 selects Bus A, 128-255 selects Bus B

*dir* direction  
       H100\_TRANSMIT write to H.100  
       H100\_RECEIVE write to TDM

*data* data value (0-255)

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

### 10.3.12 h100\_par\_read

#### Summary

Read the H.100 controller parallel access port.

#### Function Prototype

```
int h100_par_read(      PCI_MOD * pci,          Pointer to open baseboard
                      int      tdm_slot,      TDM slot
                      int      dir,          direction
                      uint8_t * data)        value
```

#### Description

This function reads the parallel access port in the H.100 device. A value read from this port will be taken from the selected bus.

#### Arguments

*tdm\_slot* Local input bus slot ID. 0-127 selects Bus A, 128-255 selects Bus B

*dir* direction  
       H100\_TRANSMIT read from H.100  
       H100\_RECEIVE read from TDM

*data* data value (0-255)

#### Return Values

Return	Errno	Meaning
1	No change	Success
0	EINVAL	Invalid parameter
0	EIO	I/O error

## 11 Special Support Functions

### 11.1 Physical to Virtual Memory Mapping

Two library functions, **enablePhyToVirCache** and **clearPhyToVirCache** are provided to work around a problem discovered with a particular system. The problem causes the system to hang or freeze when PCI-bus memory is mapped or un-mapped for a user application. This appears to be isolated to the particular system running Windows XP with multi-processor mode enabled. It did not occur when tested on other multi-processor systems or with the particular system when running Linux.

The memory mapping and un-mapping is performed for any operation that needs to access the resources on the DPT board. The more times such mappings are made or un-mapped, the greater the chances for the problem to occur. Although the exact cause of the problem remains unknown and there is no absolute solution, it may be mitigated by limiting the number of times memory on the PCI bus is mapped or un-mapped.

These functions enable the user application to cache the mapping of DPT4 resources so that each memory region only has to be mapped once per application. The memory regions are not un-mapped until the application quits.

To enable the caching, the following function call must be made prior to any other function that would access the DPT board. The function returns no value.

```
enablePhyToVirCache(1);
```

If the user application is linked to the DPT4 DLL, the physical to virtual mappings are automatically un-mapped when the user application exits. Otherwise the following function may be called prior to the application exiting:

```
clearPhyToVirCache();
```

The memory map caching is only supported on Windows based systems. The functions exist as no-ops for Solaris and Linux.

### 11.1.1 clearPhyToVirCache

#### Summary

Clears the software cache containing the physical memory to user virtual memory address map.

#### Function Prototype

```
void clearPhyToVirCache( void)
```

#### Description

This function is used to clear the software cache containing the physical memory to user virtual memory address mapping. When the software cache is cleared, the physical to virtual memory mapping are released by the driver. After the function is call, the user application should not call any DPT4 API library functions that access the DPT4 hardware. The function should only be called when the user application is closing and after all handles and channels to the DPT boards have been closed. The function only needs to be called if the user application called the enablePhyToVirCache() and the application is linked against a static version of the DPT4 API library (The windows DPT4 release only contains the DLL version of the API library, the user will need to build the static version of the library if it is required for their application ). If the application is linked to use the DLL DPT4 API library, the function is automatically called when the user application terminates.

NOTE: The caching of the physical address is only supports in the Windows environment. The function is present in the UNIX versions of the DPT4 API library but does not perform any action. It is present to allow applications to be built for both the UNIX and windows operation systems.

#### Arguments

*none*

#### Return Values

(none)

### 11.1.2 enablePhyToVirCache

#### Summary

Controls the enabling and disabling of the software cache containing the physical memory to user virtual memory address map.

#### Function Prototype

```
void enablePhyToVirCache( int enableFlag)
```

#### Description

This function is used to enable the software cache containing the physical memory to user virtual memory address mapping. To enable the software cache, call the function with the enableFlag argument set to non-zero.

If the system the user application is running on requires using the software cache, this function is required to be called prior to any DPT4 API library functions which maps the DPT4 hardware physical address into the user virtual address memory space. The following are some of the functions will map the hardware into the user space: pciopen(), pci\_open\_chan() and pci\_hw\_fast\_clk(). It is recommend that this function be called prior to any of the DPT4 API library functions.

When the user application terminates, clearPhyToVirCache() should be either directly or indirectly called to free the physical memory to user virtual memory address mapping.

NOTE: The caching of the physical address is only supports in the Windows environment. The function is present in the UNIX versions of the DPT4 API library but does not perform any action. It is present to allow applications to be built for both the UNIX and windows operation systems.

#### Arguments

*enableFlag* Controls the enabling and disabling of the software cache, a non-zero enables the cache. A typical user application should not call the function will the enableFlag set to zero to disable the software cache.

#### Return Values

(none)

## 12 Contact Information

The latest version of the DPT software can be found on the web at

[http://www.cacdsp.com/software/dpt\\_downloads/current\\_version](http://www.cacdsp.com/software/dpt_downloads/current_version)

or

[ftp://ftp.cacdsp.com/pub/dpt/current\\_version](ftp://ftp.cacdsp.com/pub/dpt/current_version)

Technical support can be obtained through e-mail to:

<mailto://support@cacdsp.com>

or by telephone:

1-877-284-4804 Toll Free (US Only)

+1-610-692-9526 International

**This page left intentionally blank.**