

# **HPDI32**

**High Performance 32-bit Digital I/O**

**PCI-HPDI32A  
PCI64-HPDI32  
PMC-HPDI32A  
PMC64-HPDI32**

## **Software Development Kit SDK 6.0.0 Reference Manual**

**Manual Revision: March 21, 2008  
Version 6.0.0**

**General Standards Corporation  
8302A Whitesburg Drive  
Huntsville, AL 35802  
Phone: (256) 880-8787  
Fax: (256) 880-8788  
URL: <http://www.generalstandards.com/>  
E-mail: [sales@generalstandards.com](mailto:sales@generalstandards.com)  
E-mail: [support@generalstandards.com](mailto:support@generalstandards.com)**



## Preface

Copyright ©2008, **General Standards Corporation**

Additional copies of this manual or other literature may be obtained from:

**General Standards Corporation**  
8302A Whitesburg Drive  
Huntsville, Alabama 35802  
Phone: (256) 880-8787  
FAX: (256) 880-8788  
URL: <http://www.generalstandards.com/>  
E-mail: [sales@generalstandards.com](mailto:sales@generalstandards.com)

**General Standards Corporation** makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Although extensive editing and reviews are performed before release to ECO control, **General Standards Corporation** assumes no responsibility for any errors that may exist in this document. No commitment is made to update or keep current the information contained in this document.

**General Standards Corporation** does not assume any liability arising out of the application or use of any product or circuit described herein, nor is any license conveyed under any patent rights or any rights of others.

**General Standards Corporation** assumes no responsibility for any consequences resulting from omissions or errors in this manual or from the use of information contained herein.

**General Standards Corporation** reserves the right to make any changes, without notice, to this product to improve reliability, performance, function, or design.

ALL RIGHTS RESERVED.

The Purchaser of this software may use or modify in source form the subject software, but not to re-market or distribute it to outside agencies or separate internal company divisions. The software, however, may be embedded in the Purchaser's distributed software. In the event the Purchaser's customers require the software source code, then they would have to purchase their own copy of the software.

**General Standards Corporation** makes no warranty of any kind with regard to this software, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose and makes this software available solely on an "as-is" basis. **General Standards Corporation** reserves the right to make changes in this software without reservation and without notification to its users.

The information in this document is subject to change without notice. This document may be copied or reproduced provided it is in support of products from **General Standards Corporation**. For any other use, no part of this document may be copied or reproduced in any form or by any means without prior written consent of **General Standards Corporation**.

GSC is a trademark of **General Standards Corporation**.

PLX and PLX Technology are trademarks of PLX Technology, Inc.

# Table of Contents

<b>1. Introduction.....</b>	<b>9</b>
1.1. Purpose .....	9
1.2. Acronyms .....	9
1.3. Definitions.....	9
1.4. Installation.....	9
1.5. Application Programming Interface .....	10
1.6. Software Overview .....	10
1.6.1. Software Architecture .....	11
1.7. Hardware Overview.....	11
1.8. Code Samples .....	11
1.9. Performance Factors .....	11
1.10. Reference Material.....	12
<b>2. Operation .....</b>	<b>13</b>
2.1. Transmitter Operation .....	13
2.1.1. Data Organization.....	14
2.1.2. Cable Signals - continuous unstructured data stream .....	14
2.1.2.1. Tx Clock .....	15
2.1.2.2. Tx Data .....	15
2.1.2.3. Tx Enabled.....	16
2.1.2.4. Tx Ready.....	16
2.1.2.5. Frame Valid .....	17
2.1.2.6. Line Valid .....	17
2.1.2.7. Status Valid.....	18
2.1.2.8. Rx Ready.....	18
2.1.3. Control Options - continuous unstructured data stream .....	19
2.1.3.1. Enable .....	19
2.1.3.2. Auto Start.....	19
2.1.3.3. Auto Stop .....	19
2.1.3.4. Flow Control .....	20
2.1.3.5. Remote Throttle .....	20
2.1.3.6. Tx Overrun.....	20
2.1.3.7. Tx/Rx Enabled Tri-State .....	20
2.2. Transmitter Setup.....	21
2.3. Transmitter Configuration.....	21
2.4. Receiver Operation .....	22
2.4.1. Data Organization.....	22
2.4.2. Cable Signals - continuous unstructured data stream .....	23
2.4.2.1. Rx Clock .....	23
2.4.2.2. Rx Data .....	23
2.4.2.3. Rx Enabled.....	24
2.4.2.4. Frame Valid .....	24
2.4.2.5. Line Valid .....	25
2.4.2.6. Status Valid.....	25
2.4.2.7. Rx Ready.....	26

2.4.3. Control Options - continuous unstructured data stream .....	26
2.4.3.1. Enable .....	26
2.4.3.2. Rx Overrun .....	27
2.4.3.3. Rx Under Run .....	27
2.4.3.4. Tx/Rx Enabled Tri-State .....	27
<b>2.5. Receiver Setup .....</b>	<b>27</b>
<b>2.6. Receiver Configuration.....</b>	<b>28</b>
<b>2.7. Data Transfer Issues .....</b>	<b>28</b>
2.7.1. Tx vs. Rx Defaults .....	28
2.7.2. I/O Abort Requests .....	28
2.7.3. I/O Data Buffers .....	28
2.7.4. General DMA Parameters .....	29
2.7.5. DMA Based I/O Requests .....	30
2.7.6. PIO Threshold .....	30
2.7.7. I/O Timeout .....	30
2.7.8. I/O Data Transfer Modes .....	31
2.7.8.1. DMA (Manual) .....	31
2.7.8.2. Demand Mode DMA .....	32
2.7.9. FIFO Almost Levels .....	32
2.7.10. Flow Control.....	33
2.7.11. Direct Register Access.....	33
<b>2.8. Event Notification .....</b>	<b>33</b>
2.8.1. Event Callback.....	33
2.8.1.1. Interrupt Notification Callback .....	34
2.8.1.2. I/O Completion Notification Callback .....	34
2.8.2. Event Waiting .....	34
<b>3. Macros.....</b>	<b>35</b>
<b>3.1. API Version Number .....</b>	<b>35</b>
<b>3.2. Common Parameter Assignment Values .....</b>	<b>35</b>
<b>3.3. Discrete Data Type Options .....</b>	<b>36</b>
<b>3.4. I/O Status Fields.....</b>	<b>37</b>
<b>3.5. Maximum Number of Open Handles .....</b>	<b>38</b>
<b>3.6. Parameter Access “Which” Bits .....</b>	<b>38</b>
<b>3.7. Registers.....</b>	<b>39</b>
3.7.1. GSC Registers .....	40
3.7.2. PLX PCI9080 PCI Configuration Registers .....	40
3.7.3. PLX PCI9080 Feature Set Registers.....	41
3.7.4. PLX PCI9656 PCI Configuration Registers .....	43
3.7.5. PLX PCI9656 Feature Set Registers.....	44
<b>3.8. Version Data Selectors.....</b>	<b>46</b>
<b>4. Data Types .....</b>	<b>47</b>
<b>4.1. Discrete Data Types .....</b>	<b>47</b>
<b>4.2. hpdi32_callback_func_t.....</b>	<b>47</b>
<b>4.3. Status Values .....</b>	<b>47</b>

<b>5. Functions.....</b>	<b>49</b>
5.1. hpdi32_api_status().....	49
5.2. hpdi32_board_count() .....	50
5.3. hpdi32_close().....	51
5.4. hpdi32_config().....	51
5.5. hpdi32_gpio_mod() .....	53
5.6. hpdi32_gpio_read().....	54
5.7. hpdi32_gpio_write().....	55
5.8. hpdi32_init() .....	55
5.9. hpdi32_io_wait().....	56
5.10. hpdi32_irq_wait().....	57
5.11. hpdi32_open().....	59
5.12. hpdi32_read() .....	60
5.13. hpdi32_reg_mod().....	61
5.14. hpdi32_reg_read().....	62
5.15. hpdi32_reg_write().....	63
5.16. hpdi32_reset().....	64
5.17. hpdi32_status_text().....	65
5.18. hpdi32_version_get().....	66
5.19. hpdi32_write() .....	67
<b>6. Configuration Parameters.....</b>	<b>70</b>
6.1. Parameter Macros .....	70
6.1.1. Parameter Definitions .....	70
6.1.2. Value Definitions.....	70
6.1.3. Service Definitions .....	70
6.1.3.1. Device Handle: h.....	70
6.1.3.2. Which Bits: w .....	71
6.1.3.3. Set Value: s .....	71
6.1.3.4. Get Value: g .....	71
6.2. Cable Parameters.....	71
6.2.1. Cable Parameter: Clock State .....	71
6.2.2. Cable Parameter: Command Mode.....	72
6.2.3. Cable Parameter: Command State .....	72
6.3. FIFO Parameters .....	73
6.3.1. FIFO Parameter: Almost Level .....	73
6.3.2. FIFO Parameter: Reset .....	74
6.3.3. FIFO Parameter: Size .....	75
6.3.4. FIFO Parameter: Status .....	75
6.3.5. FIFO Parameter: Transfer Size.....	75
6.4. I/O Parameters.....	76
6.4.1. I/O Parameter: Abort.....	77
6.4.2. I/O Parameter: Aborted .....	77

6.4.3. I/O Parameter: Buffer Pointer.....	78
6.4.4. I/O Parameter: Buffer Size .....	78
6.4.5. I/O Parameter: Callback Argument .....	79
6.4.6. I/O Parameter: Callback Function .....	79
6.4.7. I/O Parameter: Data Size .....	80
6.4.8. I/O Parameter: DMA Channel Select .....	80
6.4.9. I/O Parameter: DMA Control Mode.....	81
6.4.10. I/O Parameter: DMA Priority .....	82
6.4.11. I/O Parameter: Mode .....	82
6.4.12. I/O Parameter: Overlap Enable.....	83
6.4.13. I/O Parameter: PIO Threshold .....	83
6.4.14. I/O Parameter: Single Cycle .....	84
6.4.15. I/O Parameter: Status.....	85
6.4.16. I/O Parameter: Timeout.....	85
<b>6.5. Interrupt Parameters.....</b>	<b>86</b>
6.5.1. Interrupt Parameter: Callback Argument.....	86
6.5.2. Interrupt Parameter: Callback Function.....	87
6.5.3. Interrupt Parameter: Enable.....	87
6.5.4. Interrupt Parameter: State.....	88
6.5.5. Interrupt Parameter: Trigger Configuration.....	88
<b>6.6. Miscellaneous Parameters.....</b>	<b>89</b>
6.6.1. Miscellaneous Parameter: Board Jumpers .....	90
6.6.2. Miscellaneous Parameter: Favor Tx .....	90
6.6.3. Miscellaneous Parameter: Features .....	90
6.6.4. Miscellaneous Parameter: GSC Register Mapping.....	91
6.6.5. Miscellaneous Parameter: GSC Register Mapping Pointer .....	92
6.6.6. Miscellaneous Parameter: PLX Register Mapping.....	92
6.6.7. Miscellaneous Parameter: PCI Bus Width.....	93
6.6.8. Miscellaneous Parameter: Strict Arguments.....	93
6.6.9. Miscellaneous Parameter: Strict Configuration .....	93
6.6.10. Miscellaneous Parameter: Tx/Rx Tri-State.....	94
<b>6.7. Receiver Parameters.....</b>	<b>94</b>
6.7.1. Receiver Parameter: Rx Enable .....	95
6.7.2. Receiver Parameter: Rx Overrun.....	95
6.7.3. Receiver Parameter: Row Count .....	95
6.7.4. Receiver Parameter: State.....	96
6.7.5. Receiver Parameter: Status Count .....	96
6.7.6. Receiver Parameter: Rx Under Run .....	96
<b>6.8. Transmitter Parameters.....</b>	<b>97</b>
6.8.1. Transmitter Parameter: Auto Start.....	97
6.8.2. Transmitter Parameter: Auto Stop .....	98
6.8.3. Transmitter Parameter: Tx Clock Divider .....	98
6.8.4. Transmitter Parameter: Tx Enable.....	99
6.8.5. Transmitter Parameter: Flow Control.....	99
6.8.6. Transmitter Parameter: Line Valid Off Count.....	100
6.8.7. Transmitter Parameter: Line Valid On Count.....	100
6.8.8. Transmitter Parameter: Tx Overrun.....	101
6.8.9. Transmitter Parameter: Remote Throttle .....	101
6.8.10. Transmitter Parameter: Remote Throttle State .....	102
6.8.11. Transmitter Parameter: Tx State.....	102
6.8.12. Transmitter Parameter: Status Valid Count .....	102
6.8.13. Transmitter Parameter: Status Valid Mirror .....	103
<b>Document History .....</b>	<b>104</b>

## Table of Figures

Figure 1 A depiction of the HPDI32 Transmitter. ....	14
Figure 2 A simple continuous unstructured data stream cable configuration. ....	15
Figure 3 Tx Data is synchronized with Tx Clock. ....	16
Figure 4 The Tx Enabled signal reflects the transmitter enable state (default configuration). ....	16
Figure 5 The Tx Ready signal reflects the Tx FIFO empty state. ....	17
Figure 6 The Frame Valid signal reflects the data transmission process. ....	17
Figure 7 The Line Valid signal reflects valid transmit data being presented at the cable interface. ....	18
Figure 8 The Status Valid signal reflects valid status data being presented at the cable interface. ....	18
Figure 9 The receiving device can drive the Rx Ready signal to control data flow. ....	19
Figure 10 A depiction of the HPDI32 Receiver. ....	22
Figure 11 A simple continuous unstructured data stream cable configuration. ....	23
Figure 12 Rx Data is synchronized with Rx Clock. ....	24
Figure 13 The Rx Enabled signal reflects the receiver enable state (default configuration). ....	24
Figure 14 The Frame Valid signal reflects the data reception process. ....	25
Figure 15 The Line Valid signal reflects valid transmit data being presented at the cable interface. ....	25
Figure 16 The Status Valid signal reflects valid status data being presented at the cable interface. ....	26
Figure 17 The receiver drives the Tx Ready signal to control data flow. ....	26



# 1. Introduction

This reference manual applies to SDK release version 6.0.0.

## 1.1. Purpose

The purpose of this document is to describe the Application Programming Interface to the HPDI32 Software Development Kit. This software provides the interface between “Application Software” and the HPDI32 board. The interface provided by the SDK is based on the board’s functionality.

## 1.2. Acronyms

The following is a list of commonly occurring acronyms used throughout this document.

Acronyms	Description
API	Application Programming Interface (This is sometimes used synonymously with SDK or API Library.)
DMA	Direct Memory Access
DMDMA	Demand Mode DMA
GPIO	General Purpose Input/Output
GSC	General Standards Corporation
PCI	Peripheral Component Interconnect
PIO	Programmed I/O
PMC	PCI Mezzanine Card
SDK	Software Development Kit (This is sometimes used synonymously with API or API Library.)

## 1.3. Definitions

The following is a list of commonly occurring terms used throughout this document.

Term	Definition
API Buffer	A physically contiguous block of memory allocated via the API.
API Library	This refers to the library implementing the application level HPDI32 interface. (This is sometimes used synonymously with SDK or API.)
Application	This refers to user mode processes.
Application Buffers	These are memory buffers allocated and maintained entirely by the application, and which are used for reading data from and writing data to the HPDI32’s FIFOs.
Device Driver	This refers to the driver executable component of the HPDI32 driver package.
Driver	This refers to the device driver, which runs under control of the operating system.
PLX	This refers to the company PLX Technology, Inc., who is the supplier of the PCI bridge chip used on the HPDI32.
Rx	This is a general reference to the receiver portion of the board. This includes reception of data over the cable, either to the FIFOs or from GPIO, data I/O read operations from the receive FIFO, and any and all associated settings.
Tx	This is a general reference to the transmitter portion of the board. This includes transmission of data over the cable, either from the FIFOs or from GPIO, data I/O write operations to the transmit FIFO, and any and all associated settings.

## 1.4. Installation

Installation instructions for the SDK are provided in separate, operating system specific setup guides.

## 1.5. Application Programming Interface

The SDK API is defined in the four header files listed below. These C language headers are C++ compatible. The only header that need be included by HPDI32 applications is `hpdi32_api.h`. The API consists of macros, data types, function calls and parameter definitions. These are described in other sections of this document. The headers define numerous items in addition to those described in this document. These additional items are provided without documentation. All software components of the API begin with a prefix of HPDI32 or GSC (both appear with upper and lower case letters). The table below indicates where to look for any particular item's definition.

File Name	Description
<code>hpdi32_api.h</code>	This header contains the bulk of the API, including function calls, data types and numerous macros. All items defined here include the prefix "HPDI32" or "hpdi32".
<code>gsc_common.h</code>	This header contains status definitions, a few data type definitions and a variety of macros. All items defined here have a prefix of "GSC" or "gsc".
<code>gsc_pci9080.h</code>	This header contains register definitions for the PCI9080, which is the PCI interface chip used on HPDI32s with 32-bit PCI interfaces. All items defined here have a prefix of "GSC" or "gsc" and include "9080".
<code>gsc_pci9656.h</code>	This header contains register definitions for the PCI9656, which is the PCI interface chip used on HPDI32s with 64-bit PCI interfaces. All items defined here have a prefix of "GSC" or "gsc" and include "9656".

## 1.6. Software Overview

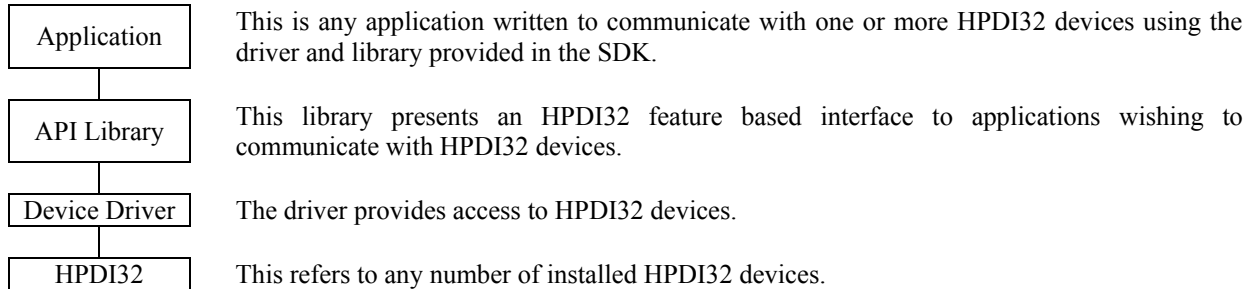
The software interface to the HPDI32 consists of a Device Driver and an API Library; the primary components of the SDK. The Device Driver operates under control of the operating system and must be loaded and running in order to access any installed HPDI32 devices. The interface provided by the API Library is based on the board's functionality and is organized around the HPDI32's set of main hardware features. The general categories are as follows and permit access to and manipulation of virtually every feature available on the board.

- General Access Services (API Status, Version Numbers, Board Count, Open, Close, ...)
- Cable Interface Configuration
- FIFO Configuration
- Data Input and Output Configuration
- General Purpose Input and Output Configuration
- Interrupt Configuration
- Other Miscellaneous Configuration
- Register read and write operations
- Receiver Configuration
- Transmitter Configuration

All HPDI32 features are individually accessible via a generalized configuration service. For each parameter, as appropriate, the API includes a set of support macros. These include setting options (i.e. defaults and acceptable values), quick access retrieval macros, and quick access manipulation macros. All are described later in this document.

### 1.6.1. Software Architecture

An application communicates with an HPDI32 using the driver and library described briefly above. Any number of applications may make simultaneous use of the library and each use is totally independent, unless specifically designed to do otherwise. Each instance provides access to at most 32 different HPDI32 devices. The diagram below describes the components and how they fit together.



**NOTE:** While multiple applications can gain access to the same device, this is discouraged since the driver maintains resources and settings per device rather than per application or device handle.

## 1.7. Hardware Overview

The HPDI32 is a high-performance 32-bit parallel digital I/O interface board. The host side connection is PCI based and is either 32-bit or 64-bit according to the model ordered. The external I/O interface varies per model ordered. The board is capable of transmitting or receiving data at up to 200 Mbytes per second over an external I/O interface, depending on the model ordered. Onboard transmit and receive FIFOs of up to 128k data values each, buffer transfer data between the PCI bus and the cable interface. This allows the HPDI32 to maintain maximum bursts on the cable interface (at least up to the depth of the FIFOs) independent of the PCI bus interface. The onboard FIFOs can also be used to buffer data between the cable interface and the PCI bus to maintain a sustained data throughput for real-time applications.

The HPDI32 offers a half-duplex external I/O interface. The board can either transmit or receive data, but it cannot do both simultaneously. In addition to the 32 synchronous data I/O lines, the external interface includes a set of configurable flow control signals. Some of these can also be configured as discrete I/O. The board accommodates a wide range of applications. This range extends from sending or receiving relatively small blocks of data on demand, to sending or receiving large continuous streams of data for an extended period. Once a data link is established, the data is transferred to/from host memory by simply writing to or reading from the onboard FIFOs. The board has an advanced PCI interface engine, which provides for increased data throughput via DMA.

**NOTE:** PCI form factor boards with a 32-bit PCI interface can be used interchangeably in 64-bit PCI slots, and vice-versa. However, the performance improvements associated with the 64-bit PCI interface can be achieved only when a 64-bit board is used in a 64-bit slot.

## 1.8. Code Samples

All of the code samples in this manual are included in the `hpdi32_ds1` library along with their C source files. The examples given are notably simplistic, but are provided to illustrate use rather than accomplishment of broader tasks.

## 1.9. Performance Factors

The HPDI32 is designed for high performance data transfer. In many instances the form factor, the cable clock rate and the external interface transceivers are dictated by the application. The performance variables that remain are the PCI Bus width and the FIFO sizes. If the application doesn't mandate the PCI bus width, then going with an HPDI32 with a 64-bit bus has the potential for better performance and/or higher bus utilization efficiency. The peak

transfer rates across the PCI bus are 528MB/S for the 64-bit PCI bus and 132MB/S for the 32-bit bus (64-bits @ 66MHz vs. 32-bits @ 33MHz). Actual performance can be drastically different for many reasons. Otherwise, the remaining performance variable is the FIFO size. As FIFO sizes increase, so do throughput rates. In many cases a 32-bit board with larger FIFOs outperforms 64-bit boards with smaller FIFOs. The processor board the HPDI32 is plugged into, and its supporting chip set, also have significant affects on performance.

## 1.10. Reference Material

The following reference material may be of particular benefit in using the HPDI32 and this SDK. The specifications provide the information necessary for an in-depth understanding of the specialized features implemented on this board.

- The applicable *HPDI32 User Manual* from General Standards Corporation.
- The *PCI9080 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc. (for 32-bit PCI interface boards) \*
- The *PCI9656 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc. (for 64-bit PCI interface boards) \*

\* PLX data books are available from PLX at the following location.

PLX Technology Inc.  
870 Maude Avenue  
Sunnyvale, California 94085 USA  
Phone: 1-800-759-3735  
WEB: <http://www.plxtech.com/>

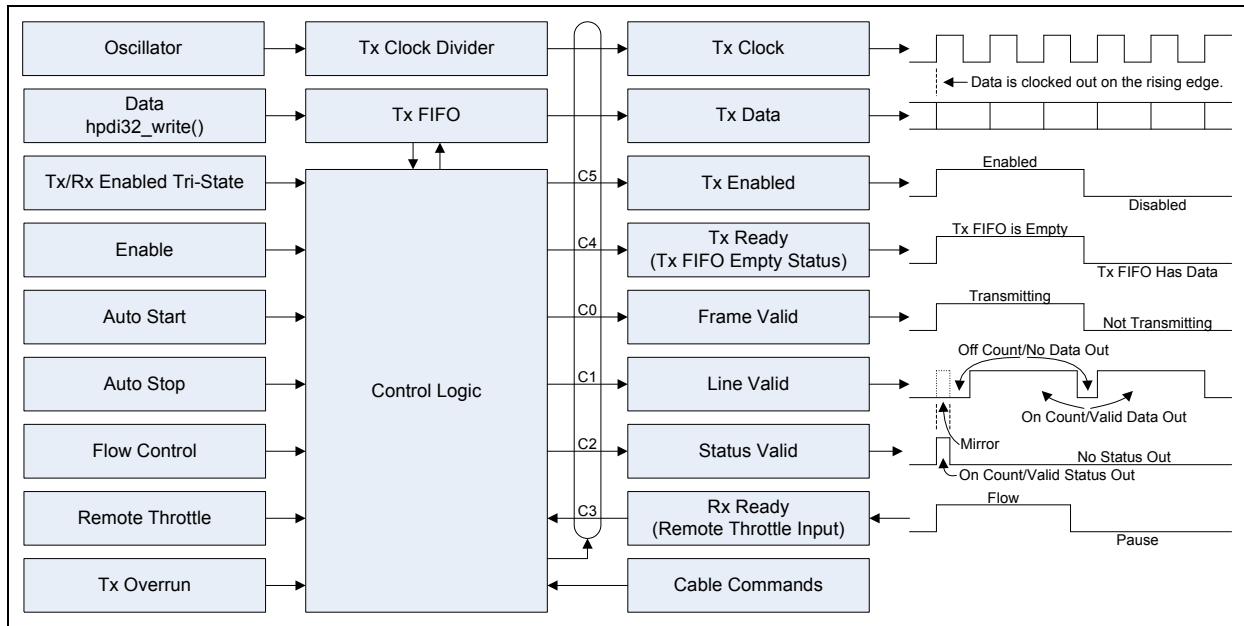
## 2. Operation

The purpose of this section is to provide information on the operation of the HPDI32 and the API. This is not intended to be comprehensive. It is intended to give a basic understanding of the board and the software while addressing some issues relating to their use.

### 2.1. Transmitter Operation

The transmitter is that portion of the HPDI32 responsible for sending data out over the cable interface. The transmitter consists of numerous hardware features that operate under control of the SDK and the application. The hardware portion includes a clock, data FIFOs, firmware registers, control logic, and cable signal transceivers. The SDK portion consists of function calls, parameter identifiers, and parameter values. Together these components permit applications to feed data to the transmitter, and give applications control over how the transmitter controls data flow out the cable interface. An overall depiction is given in Figure 1. Some general guidelines for using the transmitter are as follows. Each of these steps is further explained in subsequent paragraphs.

1. Identify the basic nature of the data's organization; continuous stream or a sequence of frames.
2. Identify the cable signals needed, how each will be used, and how each will operate.
3. Configure the cable signals according to how each will be used.
4. Configure the cable signal parameters so that each signal has the desired operating characteristics.
5. Identify how overall data flow will be started and stopped; remote, local (automatic and/or manual).
6. Configure the device according to how overall data flow will be started and stopped.
7. Configure the I/O write parameters.
8. Enable the transmitter.
9. Write data to the device.
10. As appropriate, perform any manual steps to start or stop data flow.



**Figure 1** A depiction of the HPDI32 Transmitter.

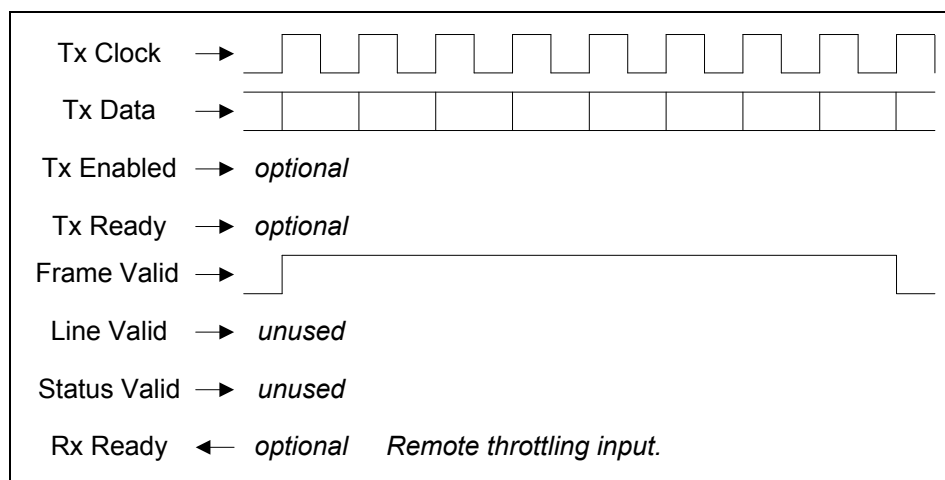
### 2.1.1. Data Organization

The HPDI32 transmitter supports two basic data organization schemes; a structured stream of frames divided into lines, and an unstructured continuous data stream. The structured format divides the overall data stream into a series of data frames, with each frame further divided into a series of data lines. Each line may be preceded by a fixed time delay in which no data is transmitted. In the unstructured format, data appears on the cable when it is available for transmission without delay. By far, most HPDI32 applications have employed an unstructured data stream. For this reason the cable signal descriptions that follow assume the use of an unstructured data stream.

### 2.1.2. Cable Signals - continuous unstructured data stream

For continuous unstructured data streams, some cable signals are required and some can be ignored or used for GPIO. The Tx Clock and Tx Data signals are always required. The Frame Valid signal is needed while the Line Valid and Status valid signals can be ignored or used for GPIO. If the remote device will be controlling data flow, then the Rx Ready signal must be used as the Remote Throttle input. Otherwise the Rx Ready signal can be ignored or used as GPIO. The Tx Enabled signal can be used to indicate when the transmitter is enabled, if desired, or it can be ignored or used as GPIO. Also, the Tx Ready signal can be used to indicate when the transmitter has data, if desired, or it too can be ignored or used as GPIO.

The simplest configuration usable for a continuous unstructured data stream is illustrated in Figure 2. This configuration uses the Tx Clock, Tx Data and Frame Valid signals, while all of the other transmitter signals are unused. Even in this simplest configuration the unused signals must be configured, though they are configured so that they are unused by the transmitter. The easiest way to do this is to configure the unused signals as general purpose inputs. Signal configuration is described below.



**Figure 2** A simple continuous unstructured data stream cable configuration.

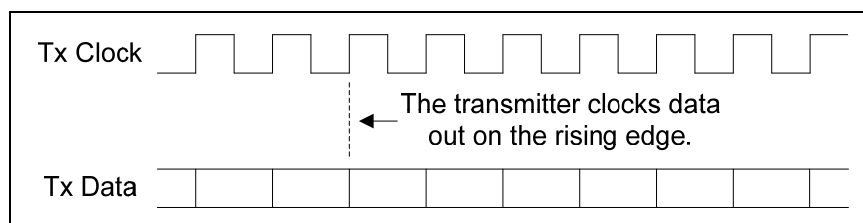
#### 2.1.2.1. Tx Clock

The Tx Clock output signal is the clock that synchronizes the transmitter logic and which clocks data out the cable interface. This clock is derived from the on-board oscillator, which is fed through the Tx Clock Divider. If the divider is zero, then the Tx Clock frequency equals the on-board oscillator frequency. Otherwise the Tx Clock frequency is governed by the formula  $F_{\text{TxC}} = F_{\text{Osc}} / (\text{Div} * 2)$ . In the formula,  $F_{\text{TxC}}$  is the Tx Clock frequency,  $F_{\text{Osc}}$  is the on-board oscillator frequency, and  $\text{Div}$  is the Tx Clock Divider value. The Tx Clock signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled the signal is not driven by the HPDI32. For enabling and disabling the transmitter, refer to “Enable” on page 19.

The Tx Clock Divider is configurable. For details on setting the divider refer to “Transmitter Parameter: Tx Clock Divider” on page 98. The divider can most easily be set using the utility macro `HPDI32_TX_CLOCK_DIVIDER_SET(h,s)`. In the macro, `h` is the device handle obtained from `hpdi32_open()` (page 59). Also, `s` is the divider value to apply and is limited to the range zero to `0xFFFF`. A return value of `GSC_SUCCESS` indicates that the operation was successful. Using the above formula with a 20MHz on-board oscillator, a divider value of two will produce a Tx Clock frequency of 5MHz. Likewise, a divider of ten will result in a 1MHz Tx Clock.

#### 2.1.2.2. Tx Data

The Tx Data output signals are synchronized with the Tx Clock to transmit 32-bits of parallel data. The transmitter clocks out the data on Tx Clock’s rising edge. See Figure 3. The transmitter hardware has a 32-bit data path, including the FIFOs and the cable transceivers. When the source data is less than 32-bits wide, it is aligned with the D0 bit and passed through the transmitter as full 32-bit data words. When the source data is 8-bits wide it appears on cable signals D0 through D7. The upper 24 data signals can be ignored, though they are driven by the transmitter. When the source data is 16-bits wide it appears on cable signals D0 through D15. The upper 16 data signals can be ignored, though they are driven by the transmitter. The Tx Data signals are driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled the signals are not driven by the HPDI32. For enabling and disabling the transmitter, refer to “Enable” on page 19.

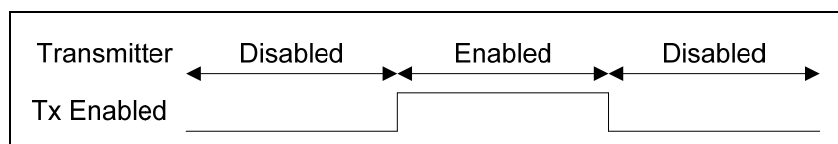


**Figure 3** Tx Data is synchronized with Tx Clock.

The cable data size is configurable. For details refer to “I/O Parameter: Data Size” on page 80. The data size can most easily be set via the utility macros `HPDI32_IO_DATA_SIZE__TX_32/16/8(h)` to specify the data size as 32, 16 or eight bits, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

### 2.1.2.3. Tx Enabled

The Tx Enabled output signal reflects the enabled state of the transmitter. This signal is not required for Flow Control of continuous unstructured data streams so applications may instead configure it as GPIO so that it is ignored by the transmitter. As a Flow Control signal, Tx Enabled is driven high when the transmitter is enabled and is driven low when disabled (see the note below for alternation operation). The signal changes state as the transmitter is enabled or disabled and is not synchronized with Tx Clock. Refer to Figure 4. For enabling and disabling the transmitter, refer to “Enable” on page 19.



**Figure 4** The Tx Enabled signal reflects the transmitter enable state (default configuration).

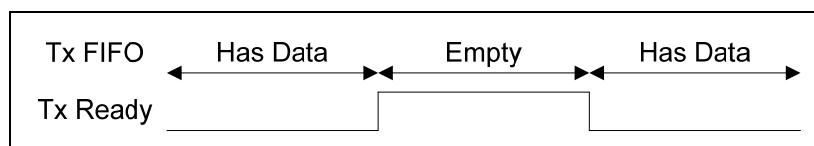
**NOTE:** An alternative option configures Tx Enabled so that it is tri-stated when the transmitter is disabled. Refer to “Tx/Rx Enabled Tri-State” on page 20.

The Tx Enabled signal refers to the Cable Command 5 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__TE_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Tx Enabled), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

### 2.1.2.4. Tx Ready

The Tx Ready output signal reflects the availability of data from the transmitter. This signal is not required for Flow Control of continuous unstructured data streams so applications may instead configure it as GPIO so that it is ignored by the transmitter. As a Flow Control signal, Tx Ready is driven high when the Tx FIFO is empty and is driven low when the Tx FIFO has data. Refer to Figure 5. The signal state changes are not synchronized with Tx Clock. The Tx Ready signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled the signal is not driven by the HPDI32. For enabling and disabling the transmitter, refer to “Enable” on page 19.



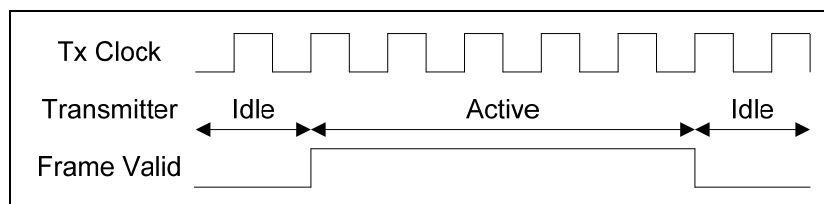


**Figure 5** The Tx Ready signal reflects the Tx FIFO empty state.

The Tx Ready signal refers to the Cable Command 4 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__TR_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Tx Ready), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

#### 2.1.2.5. Frame Valid

The Frame Valid output signal reflects the activity of the data transmission process. When the Line Valid and Status Valid signals are not used for Flow Control, then Frame Valid effectively reflects valid transmit data being presented at the cable interface. The Frame Valid signal is required for Flow Control of continuous unstructured data streams so applications must not configure it as GPIO. As a Flow Control signal, Frame Valid is driven high when the transmission process is active and is driven low when the transmission process is idle. Refer to Figure 6. The signal is synchronized with Tx Clock and changes state on the clock’s rising edge. The Frame Valid signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled the signal is not driven by the HPDI32. For enabling and disabling the transmitter, refer to “Enable” on page 19.

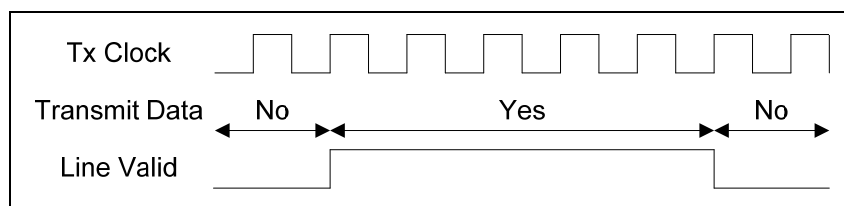


**Figure 6** The Frame Valid signal reflects the data transmission process.

The Frame Valid signal refers to the Cable Command 0 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__FV_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Frame Valid), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

#### 2.1.2.6. Line Valid

The Line Valid signal output reflects valid transmit data being presented at the cable interface. This signal is not required for Flow Control of continuous unstructured data streams so applications should configure it as GPIO so that it is ignored by the transmitter. As a Flow Control signal, Line Valid is driven high when valid transmit data is presented at the cable interface and is driven low otherwise (see below for additional information). Refer to Figure 7. The signal is synchronized with Tx Clock and changes state on the clock’s rising edge. The Line Valid signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled the signal is not driven by the HPDI32. For enabling and disabling the transmitter, refer to “Enable” on page 19.

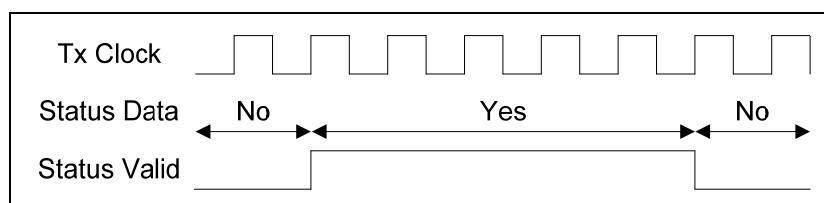


**Figure 7** The Line Valid signal reflects valid transmit data being presented at the cable interface.

The Line Valid signal refers to the Cable Command 1 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__LV_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Line Valid), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input. When configured for Flow Control, the Line Valid timing must be configured. For details refer to “Transmitter Parameter: Line Valid On Count” (page 100) and “Transmitter Parameter: Line Valid Off Count” (page 100). The operation of Line Valid is also affected by the configuration of the Status Valid signal (see the next subsection).

#### 2.1.2.7. Status Valid

The Status Valid output signal reflects valid status data being presented at the cable interface. This signal is not required for Flow Control of continuous unstructured data streams so applications should configure it as GPIO so that it is ignored by the transmitter. As a Flow Control signal, Status Valid is driven high when valid status data is presented at the cable interface and is driven low otherwise (see below for additional information). Refer to Figure 8. The signal is synchronized with Tx Clock and changes state on the clock’s rising edge. The Status Valid signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled the signal is not driven by the HPDI32. For enabling and disabling the transmitter, refer to “Enable” on page 19.



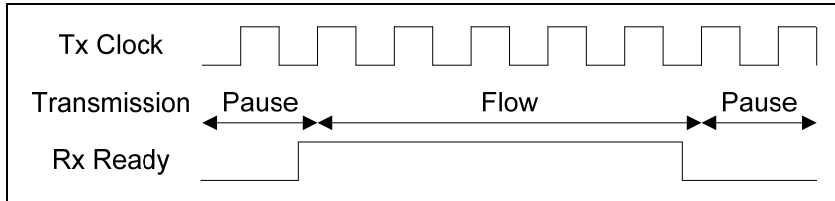
**Figure 8** The Status Valid signal reflects valid status data being presented at the cable interface.

The Status Valid signal refers to the Cable Command 2 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__SV_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Status Valid), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input. When configured for Flow Control, the Status Valid signal must be configured. For details refer to “Transmitter Parameter: Status Valid Count” (page 102) and “Transmitter Parameter: Status Valid Mirror” (page 103). The configuration of the Status Valid signal has an affect on the Line Valid signal (see the previous subsection).

#### 2.1.2.8. Rx Ready

The Rx Ready input signal may be used by receiving devices to pause the flow of data from the HPDI32 transmitter. This signal is optional for Flow Control of continuous unstructured data streams so applications may instead configure it as GPIO so that it is ignored by the transmitter. As a Flow Control signal, Rx Ready is driven high to permit data flow and is driven low to pause data flow (see below for additional information). Refer to Figure 9. The

signal is synchronized with Tx Clock such that state changes are clocked in on the clock's rising edge. The Rx Ready input signal is driven by the receiving device, and not by the HPDI32 transmitter.



**Figure 9** The receiving device can drive the Rx Ready signal to control data flow.

The Rx Ready signal refers to the Cable Command 3 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__RR_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Rx Ready), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input. When configured for Flow Control, the transmitter must be configured to utilize Rx Ready. Otherwise, the transmitter will ignore the Rx Ready input. For additional details refer to “Remote Throttle” on page 20.

### 2.1.3. Control Options - continuous unstructured data stream

The following transmitter control options are discussed from the perspective of sending data via a continuous, unstructured data stream.

#### 2.1.3.1. Enable

This option is used to enable and disable the transmitter. When enabled, the transmitter is able to send data out over the cable interface, and will do so according to related control options. That is, the transmitter will send data when directed to do so. The related control options are discussed below. When disabled, the transmitter is unable to transmit data over the cable interface. If data is being transmitted at the time the transmitter becomes disabled, then data transmission will stop. The transmitter can most easily be enabled and disabled via the utility macros `HPDI32_TX_ENABLE__YES(h)` and `HPDI32_TX_ENABLE__NO(h)`, respectively (see “Transmitter Parameter: Tx Enable” on page 99). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.1.3.2. Auto Start

This control option is used to tell the API to automatically begin data transmission over the cable interface when data is written to the HPDI32. If this option is enabled and the transmitter is enabled (see “Enable” on page 19), then the API will automatically initiate data transmission as data is being written (see `hpdi32_write()` on page 67). The Auto Start feature uses the “Flow Control” *enable* option (page 20) to initiate data transmission. If Auto Start is disabled, then data flow must be controlled either manually via the “Flow Control” option (page 20) or remotely via the “Remote Throttle” option (page 20). Auto Start can most easily be enabled and disabled via the utility macros `HPDI32_TX_AUTO_START__YES(h)` and `HPDI32_TX_AUTO_START__NO(h)`, respectively (see “Transmitter Parameter: Auto Start” on page 97). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.1.3.3. Auto Stop

This control option is available on current firmware versions, and should always be disabled. This option is presented here for completeness sake only. When the Auto Stop feature is available in firmware, disabling it could interfere with proper data flow. This option can most easily be enabled and disabled via the utility macros

`HPDI32_TX_AUTO_STOP__YES(h)` and `HPDI32_TX_AUTO_STOP__NO(h)`, respectively (see “Transmitter Parameter: Auto Stop” on page 98). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.1.3.4. Flow Control

This option is used for local, manual control to permit or pause data flow over the cable interface. If the transmitter is enabled (see “Enable” on page 19) and data is in the Tx FIFO (see `hpdi32_write()` on page 67), then data transmission over the cable interface will begin when this option is enabled. Data transmission will pause when this option is disabled. This option can most easily be used to start and stop data flow via the utility macros `HPDI32_TX_FLOW_CONTROL__START(h)` and `HPDI32_TX_FLOW_CONTROL__STOP(h)`, respectively (see “Transmitter Parameter: Flow Control” on page 99). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

**NOTE:** This option operates in parallel with the “Remote Throttle” option (page 20). These two features should generally not be used at the same time.

#### 2.1.3.5. Remote Throttle

This option configures the transmitter to use or not use the “Rx Ready” cable signal (page 18). If this option is enabled, and the “Rx Ready” signal is configured for Flow Control, then the transmitter will use that signal to either permit or pause data transmission over the cable interface. This makes it possible for the receiving device to pause data transfer as needed. When properly configured, the receiving device must drive the “Rx Ready” signal appropriately to affect the flow of data. This option can most easily be enabled and disabled via the utility macros `HPDI32_TX_REMOTE_THROTTLE__ENABLE(h)` and `HPDI32_TX_REMOTE_THROTTLE__DISABLE(h)`, respectively (see “Transmitter Parameter: Remote Throttle” on page 101). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

**NOTE:** For the remote throttling feature to function properly this option must be enabled and the “Rx Ready” signal (page 18) must be configured for Flow Control. Otherwise, the remote throttling feature will not operate properly.

**NOTE:** This option operates in parallel with the “Flow Control” option (page 20). These two features should generally not be used at the same time.

#### 2.1.3.6. Tx Overrun

This control option is available via the API, though it is rarely needed or used. This option is presented here for completeness sake only. This option is used to report cases where data has been written to the Tx FIFO when it was already full. This circumstance can occur only when applications write directly to the Tx FIFO or when applications use non-Demand Mode DMA (page 82) with the Manual DMA Control Mode option (page 81). Otherwise, the API prevents the Tx FIFO from being overfilled. This option can both report the overflow condition and clear the condition. This option can most easily be used to query for an overflow via the utility macro `HPDI32_TX_OVERRUN__GET(h,g)` (see “Transmitter Parameter: Tx Overrun” on page 101). If the value returned for `g` equals `HPDI32_TX_OVERRUN__YES`, then an overflow has occurred. An overflow can most easily be cleared via the utility macro `HPDI32_TX_OVERRUN__CLEAR(h)`. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59), and `g` is the value reported for a query. A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.1.3.7. Tx/Rx Enabled Tri-State

This option controls how the “Tx Enabled” signal (page 16) is driven when the transmitter is disabled. Ordinarily, the signal is driven all the time, even when the transmitter is disabled. With this control option however, the signal

can be tri-stated when the transmitter is disabled. The signal's state when the transmitter is disabled can most easily be tri-stated or driven low via the utility macros `HPDI32_MISC_TX_RX_TRI_STATE__YES(h)` and `HPDI32_MISC_TX_RX_TRI_STATE__NO(h)`, respectively (see "Miscellaneous Parameter: Tx/Rx Tri-State" on page 94). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

**NOTE:** This option affects both the "Tx Enabled" signal (page 16) and the "Rx Enabled" signal (page 24).

## 2.2. Transmitter Setup

The below outlines the basic steps needed to setup the HPDI32 for transmission to a receiving device. Follow these simple steps to help establish communications between the HPDI32 as a transmission device and a remote data reception device.

1. Configure the HPDI32 for data transmission as outlined in the following subsection. This includes enabling the transmitter.
2. Configure the remote device as needed for data reception operations.
3. The remote device should now be ready to receive data.
4. Initiate data transmission from the HPDI32 as appropriate.

## 2.3. Transmitter Configuration

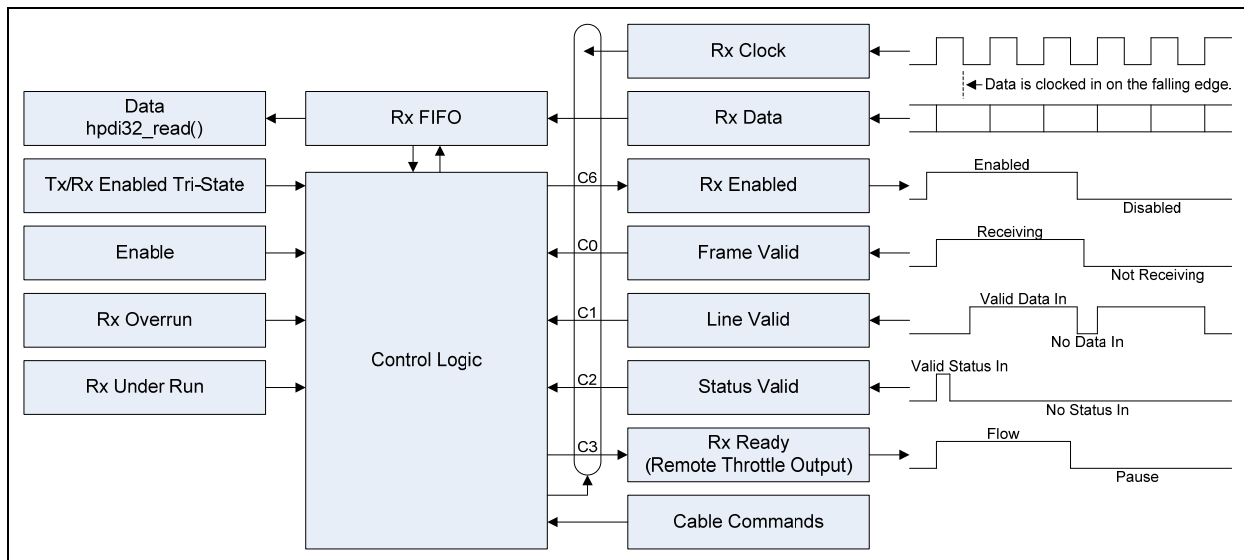
The below guidelines give an overview of the programming steps needed to configure the HPDI32 transmitter to send data out over the cable interface.

1. Return the API and the device to a known state by calling `hpdi32_init()` (page 55). This places the API and the HPDI32 in the same state it was in when first opened.
2. Configure the Miscellaneous Parameters, which can be done using the many `HPDI32_MISC_XXX()` macros (page 89).
3. Configure the Cable Parameters, which can be done using the many `HPDI32_CABLE_XXX()` macros (page 71).
4. Configure the FIFO Parameters, which can be done using the many `HPDI32_FIFO_XXX()` macros (page 73).
5. Configure the I/O Parameters, which can be done using the many `HPDI32_IO_XXX()` macros (page 76).
6. Configure the Transmitter Parameters, which can be done using the many `HPDI32_TX_XXX()` macros (page 97). Enabling the transmitter is generally a very last step.
7. Configure the Interrupt Parameters, which can be done using the many `HPDI32_IRQ_XXX()` macros (page 86).
8. Write the desired data to the device. Refer to `hpdi32_write()` on page 67.

## 2.4. Receiver Operation

The receiver is that portion of the HPDI32 responsible for receiving data in from the cable interface. The receiver consists of numerous hardware features that operate under control of the SDK and the application. The hardware portion includes data FIFOs, firmware registers, control logic, and cable signal transceivers. The SDK portion consists of function calls, parameter identifiers, and parameter values. Together these components permit applications to retrieve data from the receiver as it is captured over the cable interface. An overall depiction is given in Figure 10. Some general guidelines for using the receiver are as follows. Each of these steps is further explained in subsequent paragraphs.

1. Identify the basic nature of the data's organization; continuous stream or a sequence of frames.
2. Identify the cable signals needed, how each will be used, and how each will operate.
3. Configure the cable signals according to how each will be used.
4. Configure the cable signal parameters so that each signal has the desired operating characteristics.
5. Identify if and how overall data flow will be permitted or paused.
6. Configure the device according to how overall data flow will be permitted or paused.
7. Configure the I/O read parameters.
8. Enable the receiver.
9. Read data from the device.



**Figure 10** A depiction of the HPDI32 Receiver.

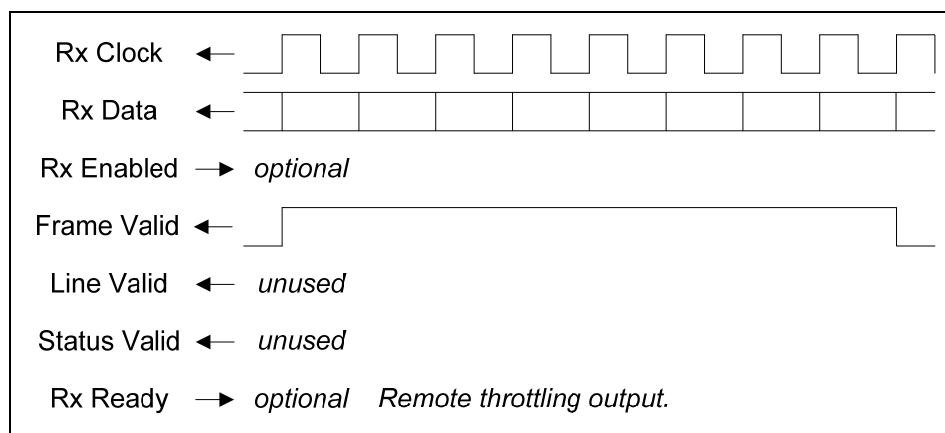
### 2.4.1. Data Organization

The HPDI32 receiver supports two basic data organization schemes; a structured stream of frames divided into lines, and an unstructured continuous data stream. The structured format divides the overall data stream into a series of data frames, with each frame further divided into a series of data lines. In the unstructured format, data is captured without regard to such boundaries. By far, most HPDI32 applications have employed an unstructured data stream. For this reason the cable signal descriptions that follow assume the use of an unstructured data stream.

### 2.4.2. Cable Signals - continuous unstructured data stream

For continuous unstructured data streams, some cable signals are required and some can be ignored or used for GPIO. The Rx Clock and Rx Data signals are always required. The Frame Valid signal is needed while the Line Valid and Status valid signals can be ignored or used for GPIO. If the remote device can be paused, then the Rx Ready signal may be used as the Remote Throttle output. Otherwise the Rx Ready signal can be ignored or used as GPIO.

The simplest configuration usable for a continuous unstructured data stream is illustrated in Figure 11. This configuration uses the Rx Clock, Rx Data and Frame Valid signals, while all of the other receiver signals are unused. Even in this simplest configuration the unused signals must be configured, though they are configured so that they are unused by the receiver. The easiest way to do this is to configure the unused signals as general purpose inputs. Signal configuration is described below.



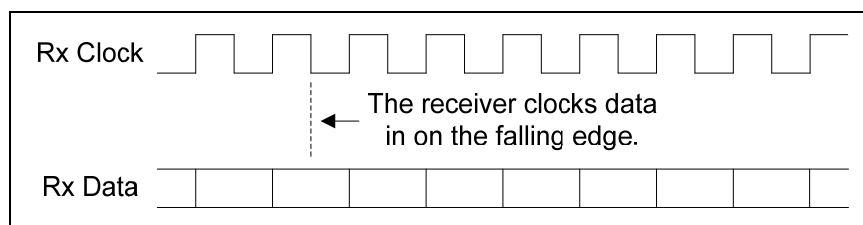
**Figure 11** A simple continuous unstructured data stream cable configuration.

#### 2.4.2.1. Rx Clock

The Rx Clock input signal is the clock that synchronizes the receiver logic and which clocks data in from the cable interface. The Rx Clock must be provided by the remote transmitting device. The input is ignored when the receiver is disabled and must be driver when the receiver is enabled. For enabling and disabling the receiver, refer to “Enable” on page 26.

#### 2.4.2.2. Rx Data

The Rx Data input signals are synchronized with the Rx Clock to record 32-bits of parallel data. The receiver clocks in the data on Rx Clock’s falling edge. The transmitting device clocks out the data on the clock’s rising edge. See Figure 12. The receiver hardware has a 32-bit data path, including the FIFOs and the cable transceivers. When the source data is less than 32-bits wide, it is aligned with the D0 bit and passed through the receiver as full 32-bit data words. When the source data is 8-bits wide it appears on cable signals D0 through D7. The upper 24 data signals are recorded, but can be ignored. When the source data is 16-bits wide it appears on cable signals D0 through D15. The upper 16 data signals are recorded, but can be ignored. The Rx Data signals are ignored when the receiver is disabled and must be driver when the receiver is enabled. For enabling and disabling the receiver, refer to “Enable” on page 26.

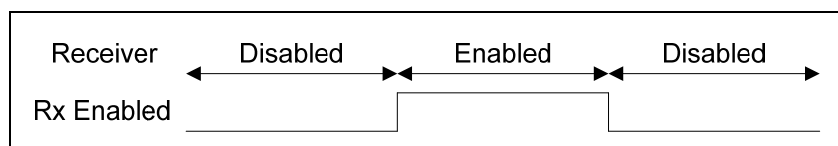


**Figure 12** Rx Data is synchronized with Rx Clock.

The cable data size is configurable. For details refer to “I/O Parameter: Data Size” on page 80. The data size can most easily be set via the utility macros `HPDI32_IO_DATA_SIZE__RX_32/16/8(h)` to specify the data size as 32, 16 or eight bits, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.4.2.3. Rx Enabled

The Rx Enabled output signal reflects the enabled state of the receiver. This signal is not required for Flow Control of continuous unstructured data streams so applications may instead configure it as GPIO so that it is ignored by the receiver. As a Flow Control signal, Rx Enabled is driven high when the receiver is enabled and is driven low when disabled (see the note below for alternation operation). The signal changes state as the receiver is enabled or disabled and is not synchronized with Rx Clock. Refer to Figure 13. For enabling and disabling the receiver, refer to “Enable” on page 26.



**Figure 13** The Rx Enabled signal reflects the receiver enable state (default configuration).

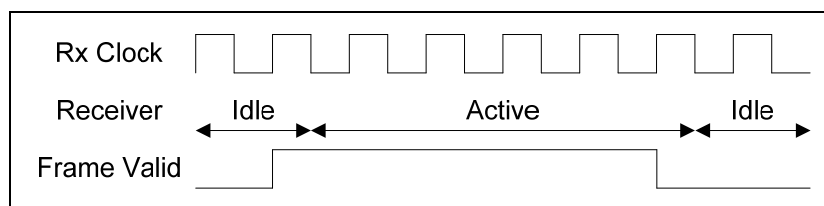
**NOTE:** An alternative option configures Rx Enabled so that it is tri-stated when the receiver is disabled. Refer to “Tx/Rx Enabled Tri-State” on page 27.

The Rx Enabled signal refers to the Cable Command 6 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__RE_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Rx Enabled), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

#### 2.4.2.4. Frame Valid

The Frame Valid input signal reflects the activity of the data reception process. When the Line Valid and Status Valid signals are not used for Flow Control, then Frame Valid effectively reflects valid receive data being available at the cable interface. The Frame Valid signal is required for Flow Control of continuous unstructured data streams so applications must not configure it as GPIO. As a Flow Control signal, Frame Valid is driven high when the reception process is active and is driven low when the reception process is idle. Refer to Figure 14. The signal is synchronized with Rx Clock. Frame Valid should change state on the clock’s rising edge as it is clocked in on the clock’s falling edge. The Frame Valid signal is ignored when the receiver is disabled and must be driver when the receiver is enabled. For enabling and disabling the receiver, refer to “Enable” on page 26.



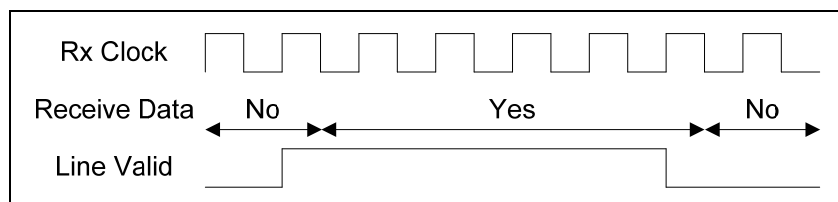


**Figure 14** The Frame Valid signal reflects the data reception process.

The Frame Valid signal refers to the Cable Command 0 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__FV_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Frame Valid), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

#### 2.4.2.5. Line Valid

The Line Valid input signal reflects valid receive data being presented at the cable interface. This signal is not required for Flow Control of continuous unstructured data streams so applications should configure it as GPIO so that it is ignored by the receiver. As a Flow Control signal, Line Valid is driven high when valid transmit data is presented at the cable interface and is driven low otherwise (see below for additional information). Refer to Figure 15. The signal is synchronized with Rx Clock. Line Valid should change state on the clock’s rising edge as it is clocked in on the clock’s falling edge. The Line Valid signal is ignored when the receiver is disabled and must be driver when the receiver is enabled. For enabling and disabling the receiver, refer to “Enable” on page 26.

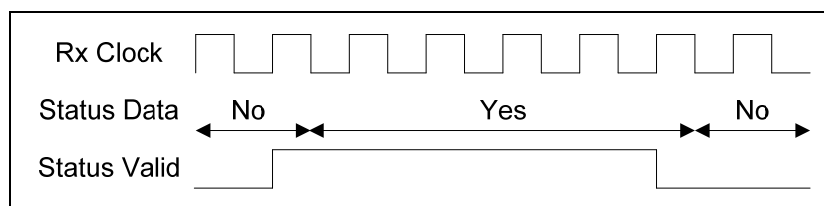


**Figure 15** The Line Valid signal reflects valid transmit data being presented at the cable interface.

The Line Valid signal refers to the Cable Command 1 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__LV_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Line Valid), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

#### 2.4.2.6. Status Valid

The Status Valid input signal reflects valid status data being presented at the cable interface. This signal is not required for Flow Control of continuous unstructured data streams so applications should configure it as GPIO so that it is ignored by the receiver. As a Flow Control signal, Status Valid is driven high when valid status data is presented at the cable interface and is driven low otherwise (see below for additional information). Refer to Figure 16. The signal is synchronized with Rx Clock and is clocked in on the clock’s falling edge. The Status Valid signal is ignored when the receiver is disabled and must be driver when the receiver is enabled. For enabling and disabling the receiver, refer to “Enable” on page 26.

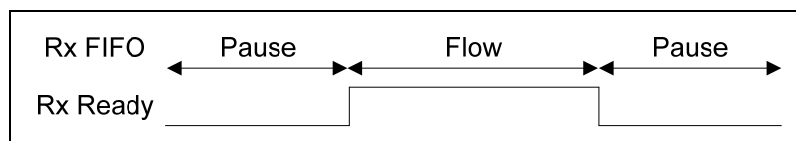


**Figure 16** The Status Valid signal reflects valid status data being presented at the cable interface.

The Status Valid signal refers to the Cable Command 2 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__SV_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Status Valid), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

#### 2.4.2.7. Rx Ready

The Rx Ready output signal may be used by the receiver to pause the flow of data from the remote transmitting device. This signal is optional for Flow Control of continuous unstructured data streams so applications may instead configure it as GPIO so that it is ignored by the receiver. As a Flow Control signal, Rx Ready is driven high to permit data flow and is driven low to pause data flow. Refer to Figure 17. The signal reflects the Rx FIFO Almost Full Status. The signal is not synchronized with Rx Clock and changes state as the FIFO fill level changes. The Rx Ready signal is driven when the receiver is enabled and is tri-stated when the receiver is disabled. For enabling and disabling the receiver, refer to “Enable” on page 26.



**Figure 17** The receiver drives the Tx Ready signal to control data flow.

The Rx Ready signal refers to the Cable Command 3 signal when configured to operate in its Flow Control mode. For details on setting the mode refer to “Cable Parameter: Command Mode” on page 72. The mode can most easily be set via the utility macros `HPDI32_CABLE_COMMAND_MODE__RR_FC/IN/LOW/HI(h)` to set the mode to Flow Control (Rx Ready), a general purpose input, a general purpose output driven low, or a general purpose output driven high, respectively. In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful. To configure the signal so that it can be ignored altogether, configure it as a general purpose input.

### 2.4.3. Control Options - continuous unstructured data stream

The following receiver control options are discussed from the perspective of receiving data via a continuous, unstructured data stream.

#### 2.4.3.1. Enable

This option is used to enable and disable the receiver. When enabled, the receiver is able to capture data from the cable interface, and will do so according to related control options. That is, the receiver will record data when directed to do so. The related control options are discussed below. When disabled, the receiver is unable to receive data over the cable interface. If data is being received at the time the receiver becomes disabled, then data recording will stop. The receiver can most easily be enabled and disabled via the utility macros `HPDI32_RX_ENABLE__YES(h)` and `HPDI32_RX_ENABLE__NO(h)`, respectively (see “Receiver Parameter:

Rx Enable” on page 95). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.4.3.2. Rx Overrun

This control option is used to capture instances where data is recorded into the Rx FIFO when it is already full. This can occur only when the receiver is recording data faster than it is being read out by the host. This option can both report the overflow condition and clear the condition. This option can most easily be used to query for an overflow via the utility macro `HPDI32_RX_OVERRUN__GET(h,g)` (see “Receiver Parameter: Rx Overrun” on page 95). If the value returned for `g` equals `HPDI32_RX_OVERRUN_YES`, then an overflow has occurred. An overflow can most easily be cleared via the utility macro `HPDI32_RX_OVERRUN__CLEAR(h)` (see page 95). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59), and `g` is the value reported for a query. A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.4.3.3. Rx Under Run

This control option is available via the API, though it is rarely needed or used. This option is presented here for completeness sake only. This option is used to report cases where data has been read from the Rx FIFO when it was empty. This circumstance can occur only when applications read directly from the Rx FIFO or when applications use non-Demand Mode DMA (page 82) with the Manual DMA Control Mode option (page 81). Otherwise, the API prevents the Rx FIFO from being read when empty. This option can both report the underflow condition and clear the condition. This option can most easily be used to query for an underflow via the utility macro `HPDI32_RX_UNDER_RUN__GET(h,g)` (see “Receiver Parameter: Rx Under Run” on page 96). If the value returned for `g` equals `HPDI32_RX_UNDER_RUN_YES`, then an underflow has occurred. An underflow can most easily be cleared via the utility macro `HPDI32_RX_UNDER_RUN__CLEAR(h)` (see page 96). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59), and `g` is the value reported for a query. A return value of `GSC_SUCCESS` indicates that the operation was successful.

#### 2.4.3.4. Tx/Rx Enabled Tri-State

This option controls how the “Rx Enabled” signal (page 24) is driven when the receiver is disabled. Ordinarily, the signal is driven all the time, even when the receiver is disabled. With this control option however, the signal can be tri-stated when the receiver is disabled. The signal’s state when the receiver is disabled can most easily be tri-stated or driven low via the utility macros `HPDI32_MISC_TX_RX_TRI_STATE__YES(h)` and `HPDI32_MISC_TX_RX_TRI_STATE__NO(h)`, respectively (see “Miscellaneous Parameter: Tx/Rx Tri-State” on page 94). In the macros, `h` is the device handle obtained from `hpdi32_open()` (page 59). A return value of `GSC_SUCCESS` indicates that the operation was successful.

**NOTE:** This option affects both the “Rx Enabled” signal (page 24) and the “Tx Enabled” signal (page 16).

## 2.5. Receiver Setup

The below outlines the basic steps needed to setup the HPDI32 for reception from a transmitting device. Follow these simple steps to help establish communications between the HPDI32 as a reception device and a remote data transmission device.

1. Configure the remote device as needed for data transmission operations. This includes driving all appropriate signals that go to the HPDI32.
2. Configure the HPDI32 for data reception as outlined in the following subsection. This includes enabling the receiver.

3. The HPDI32 is now ready to receive data, so the HPDI32 application should prepare itself for reception of data.
4. Initiate data transmission from the remote device.

## 2.6. Receiver Configuration

The below guidelines give an overview of the programming steps needed to configure the HPDI32 receiver to capture data from the cable interface.

1. Return the API and the device to a known state by calling `hpdi32_init()` (page 55). This places the API and the HPDI32 in the same state it was in when first opened.
2. Configure the Miscellaneous Parameters, which can be done using the many `HPDI32_MISC_XXX()` macros (page 89).
3. Configure the Cable Parameters, which can be done using the many `HPDI32_CABLE_XXX()` macros (page 71).
4. Configure the FIFO Parameters, which can be done using the many `HPDI32_FIFO_XXX()` macros (page 73).
5. Configure the I/O Parameters, which can be done using the many `HPDI32_IO_XXX()` macros (page 76).
6. Configure the Receiver Parameters, which can be done using the many `HPDI32_RX_XXX()` macros (page 94). Enabling the transmitter is generally a very last step.
7. Configure the Interrupt Parameters, which can be done using the many `HPDI32_IRQ_XXX()` macros (page 86).
8. Read data from the device. Refer to `hpdi32_read()` on page 60.

## 2.7. Data Transfer Issues

### 2.7.1. Tx vs. Rx Defaults

There are numerous configurable parameters governing data transfer. When a device is first opened, all are in their default state and permit optimal data transmission, once the transmitter is enabled. While some parameters default to favor data transmission they are few in number and can easily be configured to favor data reception. These can be found in `hpdi32_api.h` by looking for those macros ending in `RX_DEFAULT` and `TX_DEFAULT`.

### 2.7.2. I/O Abort Requests

The API includes the feature of aborting I/O operations. One issue with requesting an abort is that overlapped I/O operations occur in the background with threads which may have a priority greater than that of the requesting thread. This means that the I/O operation, because of its higher priority, may complete before the API is able to register the abort request.

### 2.7.3. I/O Data Buffers

The API Library supports the use of Application Buffers (application allocated buffers) and API Buffers (the API's internally allocated buffers) for I/O operations. Each has pluses and minuses and both can be used by `hpdi32_read()` (page 60) and `hpdi32_write()` (page 67) interchangeably. Application Buffers are under application control and are usually obtained by `malloc()` or similar services. This permits an application to have

any number of buffers of most any desired size, and they can even exceed the size of physical memory. The drawback they have though is that allocations only appear to be contiguous, when, in fact, they are actually scattered throughout physical memory. In addition, they can be paged out to the hard disk as needed by the OS. A result of this is additional overhead when performing DMA based I/O. API Buffers, on the other hand, avoid this inefficiency because they occupy physically contiguous and immovable memory regions, and therefore don't require the overhead during DMA requests. The disadvantage though, is that these may be smaller than desired and the API supports only two. (While each of the two API Buffers is associated with a particular I/O data direction, both can be used interchangeably at will.) In addition, Application Buffers used for DMA based I/O must reside in memory that is both readable and writable. This usually means that I/O buffers declared as `const` or `static const` cannot be used. DMA based I/O requests will fail if the Application Buffers do not have read/write access.

While use of API Buffers may generally give better performance, overall performance will be application dependent. API users are free to use whichever type desired and can switch from one to the other as needed. Within each I/O direction though, there is a small performance penalty when switching from one type to another. There is no penalty however when switching between the Rx API Buffer and the Tx API Buffer, as the Rx/Tx association is part of the interface and not the implementation. The API Buffers are ideally suited for applications wishing to implement a ping-pong or ring-buffer type I/O buffering mechanism.

API Buffers are accessed via the I/O Buffer Size and I/O Buffer Pointer parameters (see "I/O Parameter: Buffer Size" on page 78 and "I/O Parameter: Buffer Pointer" on page 78, respectively). Each buffer size starts out at zero (0) and the pointer as NULL. Application must first use the I/O Buffer Size parameter in order to make an allocation request. Since the resources for these memory regions are much more limited than for `malloc()` type requests, the size of the allocation obtained may be smaller than asked for. After a size request use the I/O Buffer Pointer parameter to get a pointer to the memory obtained. Each attempt to alter the size demands that the application update its pointer. Failure to do so is likely to produce a protection fault. When finished, setting the size to zero (0) frees the buffer.

**NOTE:** Using Application Buffers for DMA based I/O requests imposes system overhead on the call because the memory pages must be prepared for access by the DMA engine. This overhead may result in data transmission pauses (because the Tx FIFO runs empty) or Rx FIFO Overruns (because the Rx FIFO fills before data retrieval gets underway). The amount of overhead imposed can be reduced by reducing the size of the I/O requests, which results in fewer memory pages being processed at any one time. The overhead can be eliminated by using API Buffers, since they are ready for DMA engine use when allocated.

#### 2.7.4. General DMA Parameters

The API has two parameters that affect DMA operations. They are I/O DMA Channel Select and I/O DMA Priority (see "I/O Parameter: DMA Channel Select" on page 80 and "I/O Parameter: DMA Priority" on page 82, respectively). Both operate independently and are described below.

DMA channel selection (page 80) is a process the API follows to assign a DMA channel to an I/O operation. (All HPDI32s have two DMA channels, but both channels don't always have the same capabilities.) If the selection parameter is set to Static, then the API will select a channel the first time it is needed and retain it until directed otherwise. This way, the first read (or write) request will take the overhead hit to acquire the channel, and not again until called for. If set to Dynamic, then the API will select a channel at the beginning of an I/O request and release it as soon as the request completes. The results is an overhead hit at the beginning of each I/O request to acquire the channel and an addition overhead hit afterwards to release it. This parameter should always be set to Static unless the application will be performing simultaneous\* Demand Mode DMA reads and writes on an HPDI32 whose firmware supports only a single DMA channel. Otherwise the parameter should be set to Dynamic. If the Miscellaneous Features parameter reports that the DMA Channel 1 feature is supported, then the HPDI32 can perform bi-directional Demand Mode DMA\*.

The I/O DMA Priority parameter (page 82) is a factor only when performing simultaneous\* reads and writes using either form of DMA. Under these circumstances, if the I/O DMA Priorities are the same for both I/O directions,

either enabled or disabled, then a rotating priority scheme is adopted. Since the HPDI32 cannot perform bidirectional data transfer over the cable interface, the setting of this parameter should not have a noticeable affect on overall performance.

\* Here, simultaneous and bi-directional refer to data transfer over the PCI bus, not the external cable interface.

### 2.7.5. DMA Based I/O Requests

The two DMA engines on the HPDI32 are each limited to transfers of 8,388,607 bytes. That is one byte shy of 8-megabytes. For 32-bit samples this translates to a transfer limit of 8,388,604 bytes, or 2,097,151 samples. For 16-bit samples this translates to a transfer limit of 8,388,606 bytes, or 4,194,303 samples. The API breaks all DMA requests into smaller requests based on these limits. So, if an application made a request for 8MB using 32-bit samples, the API would break that into one request for (8M - 4) bytes and another request for four bytes. Application should therefore consider making DMA requests smaller than, or a multiple of the size limit for the particular sample size in use.

**NOTE:** The DMA engine limitations do not restrict the size of the I/O requests that applications may make of the API. These limitations apply only to the API's processing of such requests and are noted here only to assist applications in achieving the highest possible efficiencies. The API's I/O request size limitation is based on the macro `GSC_IO_STATUS_COUNT_MASK`, which limits requests to approximately 256MB.

### 2.7.6. PIO Threshold

Both forms of DMA based I/O require a certain amount of overhead for setup, maintenance and shutdown. For large requests this is a small price to pay for dramatic performance gains. For smaller requests however DMA could actually be slower than using the PIO mode. To help maximize performance, particularly in cases when DMA requests may vary in size, the I/O PIO Threshold parameter handles an automatic switchover to PIO. The switchover has no performance penalty and operates by using PIO mode when I/O requests are at or below the configured level. See "I/O Parameter: PIO Threshold" on page 83.

### 2.7.7. I/O Timeout

In general the timeout settings should be made so that they expire only when something has gone wrong (see exception below). This is not critical with PIO, but it is with DMA. With PIO transfers, a timeout has no consequence except to cause the API to transfer no additional data. In this case no data is lost and an exact accounting of the amount of data transferred is accurately maintained. With DMA transfers, a timeout results in the DMA engine aborting the transfer midstream. For the HPDI32 this means that the amount of data that was successfully transferred in that request is unknown. With Non-Demand Mode DMA, whether Automatic or Manual, since they tend to complete very quickly, there is little chance of a timeout. For example, a 128K sample request should complete in as little as 5µs, making it unlikely that a timeout will occur midstream. With Demand Mode DMA the chances of a timeout during the transfer are much more likely. This is because transfers can last for very, very long periods. No matter which DMA form is used, if a timeout is encountered, the amount of data transferred in that request will be unknown. See "I/O Parameter: Timeout" on page 85.

The exception to the above guidelines is with a timeout of zero (0). A timeout of zero tells the API to transfer what is available right now and return. This is trivial for PIO since it simply returns when no additional data can be transferred. However, DMA transfers occur in the background and individual, smaller transfers occur based upon the FIFO fill level. This may result in inefficient use of DMA, but it does observe the zero timeout exception. Otherwise most DMA transfers would always timeout since the timeout check occurs just as the DMA is started.

**NOTE:** Applications should avoid setting the timeout limit to zero (0) when using any form of DMA. Doing so may result in inefficient use of DMA and it may be noticeable slower than expected.

### 2.7.8. I/O Data Transfer Modes

The API Library offers three data transfer modes. Each has its pros and cons, which are described briefly below. For additional information refer to “I/O Parameter: Mode” on page 82.

Mode	Description
PIO	This mode uses repetitive register accesses to perform transfers and is capable of transfer rates over 20MB/s.
	Pros: It is the most reliable mode offered. It is well suited for any size I/O request. This mode can be used with 8-bit, 16-bit and 32-bit data. This mode should never return a failure status for valid requests.
	Cons: It is very inefficient.
DMA (Automatic)	This mode uses non-Demand Mode DMA, which transfers data without regard to the FIFO's content. This mode also has the I/O DMA Control Mode parameter set to Automatic. While the actual transfers are performed blindly, the API guarantees data integrity by examining the FIFOs and breaking the request into smaller, appropriately sized chunks. See “I/O Parameter: DMA Control Mode” on page 81.
	Pros: This is the DMA option least likely to encounter an I/O timeout. It is well suited for any size I/O request. See note below. This mode can be used with 8-bit, 16-bit and 32-bit data.
	Cons: It uses DMA inefficiently due to making multiple smaller transfers. If an I/O timeout is encountered, the amount of data may be more than the amount reported. See note below. This mode could return a failure status, depending on system or HPDI32 resources.
DMA (Manual)	This mode uses non-Demand Mode DMA, which transfers data without regard to the FIFO's content. This mode also has the I/O DMA Control Mode parameter set to Manual. Because the data is transferred blindly, the application is responsible for maintaining data integrity by making requests that won't overrun or under run the respective FIFOs. See “I/O Parameter: DMA Control Mode” on page 81.
	Pros: This is less likely to encounter an I/O timeout than Demand Mode DMA. It is best suited for I/O requests not exceeding the size of the respective FIFO. See note below. This mode can be used with 8-bit, 16-bit and 32-bit data.
	Cons: This requires the most effort on the part of the application. If there is an I/O timeout, the amount of data transferred is unknown. See note below. This mode could return a failure status, depending on system or HPDI32 resources.
Demand Mode DMA	This uses the DMA engine's Demand Mode DMA option, which performs the transfer according to the respective FIFO's fill level.
	Pros: This is the most efficient mode offered, especially for very large transfers.
	Cons: If there is an I/O timeout, the amount of data transferred is unknown. See note below. This mode could return a failure status, depending on system or HPDI32 resources.

**NOTE:** If an I/O timeout period expires while the DMA engine is performing a transfer, the transfer is aborted and the amount of data transferred will be unknown.

#### 2.7.8.1. DMA (Manual)

This refers to the DMA I/O mode with the Manual DMA Control Mode setting. Refer to “I/O Parameter: Mode” on page 82 and “I/O Parameter: DMA Control Mode” on page 81.

For write operations, maximum efficiency can generally be achieved when the below conditions are met. The general purpose of these conditions is to make it possible to maintain continuous data transmission over a given time period in the most efficient manner possible.

1. Use the transmit FIFO's Almost Full status as a stimulus to queue additional data for subsequent write operations. The amount of data that needs to be queued is generally a function of the data transfer rate, the period of time over which the rate is to be maintained, the amount of data to be transmitted in the allotted period, the amount of time needed to make the data available for queuing, and application, driver and system overhead. Since the Almost Full status doesn't affect data transfer into the Tx FIFO, the fill level can be set strictly according to application needs.
2. Use the transmit FIFO's Almost Empty status as a stimulus to perform a write operation. The amount of data submitted in each request should be the size of the transmit FIFO minus the Almost Empty value. Since the Almost Empty status doesn't affect Tx FIFO data transfer, the fill level can be set strictly according to application needs. It is desirable though to set the Almost Empty status level as low as possible to assist in overall system efficiency. In contrast however, the status level should be set high enough to prevent the FIFO from becoming empty before the write operation begins, thus preventing a lapse in data transmission due to an empty FIFO.

For read operations, maximum efficiency can generally be achieved when the following conditions are met. The general purpose of these conditions is to make it possible to maintain continuous data reception over a given time period in the most efficient manner possible.

1. Use the receive FIFO's Almost Full status as a stimulus to perform a read operation. The amount of data requested in each request should be the size of the receive FIFO minus the Almost Full value. Since the Almost Full status doesn't affect Rx FIFO data transfer, the fill level can be set strictly according to application needs. It is desirable though to set the Almost Full status level as low as possible to assist in overall system efficiency. In contrast however, the status level should be set high enough to prevent the FIFO from becoming full before the read operation begins, thus preventing loss of data or a halt to data reception due to a full FIFO.

#### 2.7.8.2. Demand Mode DMA

This mode is intended for data transfers that exceed the size of the respective FIFO and uses the FIFO fill levels to control data movement during transfers between the host and the HPDI32. While the FIFOs can hold up to 128K data values, Demand Mode DMA reads and writes may typically entail requests for millions of data values in a single call. For write operations, the data transfer rate into the Tx FIFO peaks while the FIFO is not Almost Full. While the FIFO is Almost Full the transfer rate slows slightly (to maintain reliability). No data transfer occurs while the FIFO is Full. For read operations, the data transfer rate out of the Rx FIFO peaks while the FIFO is not Almost Empty. While the FIFO is Almost Empty the transfer rate slows slightly (to maintain reliability). No data transfer occurs while the FIFO is Empty. Refer to "I/O Parameter: Mode" on page 82.

#### 2.7.9. FIFO Almost Levels

The FIFO Almost Levels and the FIFO status bits they drive basically have two uses; event notification and data flow control. For event notification the levels should be configured as close to the empty or full condition being monitored as possible. In general, with variable or large sized I/O requests, performance increases as the setting levels are reduced. This is because it affords fewer transfers and larger transfer sizes (things run more efficiently). This however is highly application dependent. For data flow control, things are less variable. For data transmission using Demand Mode DMA, data movement into the Tx FIFO slows slightly when the Almost Full level is reached. This helps insure data integrity near the Tx FIFO Full state. For data reception, data movement out of the Rx FIFO slows when the fill level hits the Rx FIFO Almost Empty state. This helps insure data integrity near the Rx FIFO Empty state. In addition, the Rx FIFO Almost Full state drives the cable's Rx Ready signal. This gives the remote device time to halt data transfer before an Rx FIFO Overrun occurs. Refer to "FIFO Parameter: Almost Level" on page 73.



### 2.7.10. Flow Control

For transmit operations, flow control defaults to fully automatic local control. This is achieved by having the Auto Start parameter enabled (page 19), the Auto Stop parameter disabled (page 19), and the Remote Throttle parameter disabled (page 20). With this setup applications can send data out the board essentially by just enabling the transmitter then calling `hpdi32_write()` (page 67). If however, the Auto Start parameter is disabled, then applications must use the Flow Control parameter (page 20) to forcibly start and stop the flow of data over the cable. Applications must also factor this into the I/O Timeout parameter setting and must supply data at a rate sufficient to prevent the Tx FIFO from running empty.

For remote control of transmission data flow, the Remote Throttle parameter must be enabled (page 20). Also, the Rx Ready signal (page 18) must be configured for Flow Control, its default. When this is done the remote device drives the Rx Ready signal to permit or inhibit data transfer. With this setup applications accommodate transmission by enabling the transmitter then calling `hpdi32_write()` (page 67). Here to, applications must factor this configuration into the I/O Timeout parameter setting and must supply data at a rate sufficient to prevent the Tx FIFO from running empty.

Local control of receive data flow is handled automatically by the HPDI32. Applications essentially need only enable the receiver then call `hpdi32_read()` (page 60) to retrieve collected data. Application responsibility here must be to retrieve data at a rate sufficient to prevent the Rx FIFO from running either Full or Almost Full. The result of the Rx FIFO becoming Full is the probable loss of data due to an Rx FIFO Overrun condition. The result of the Rx FIFO becoming Almost Full is the halt to data flow since the HPDI32 applies this to the cable's Rx Ready signal, thus directing the remote device to stop supplying data. If the application cannot read data fast enough, then either data flow will pause or data will be lost.

The HPDI32 does not have a cable signal dedicated to local control of receive data flow. To implement this type control, an application must configure one of the dual function cable signals as GPIO output. Software can then manipulate that output as appropriate to command the remote device to commence or cease data flow. As in the above scenarios, applications must account for this operation when setting the I/O Timeout parameter.

### 2.7.11. Direct Register Access

While direct access to the HPDI32 firmware register can contribute to a performance gain, there is virtually no gain to an application using this feature for I/O purposes, even for Non-Demand Mode DMA under Manual operation. The reason is because the API uses direct register access at all times, when possible. This is done automatically for performance reasons and occurs unless the application disables the Miscellaneous GSC Register Mapping parameter. If this is done, then direct access is available to neither the API nor the application. Refer to "Miscellaneous Parameter: GSC Register Mapping" on page 91 and "Miscellaneous Parameter: GSC Register Mapping Pointer" on page 92.

## 2.8. Event Notification

The API Library supports event notification for two sources or types of events. They include Interrupt Notification and I/O Completion Notification and operate independently. Notification for both sources includes both a callback mechanism and a wait mechanism. All are described below. Interrupt Notification is driven by interrupts generated by the HPDI32 from any of the interrupt sources identified in the Interrupt Control Register (`HPDI32_ICR`). I/O Completion Notification is associated with completed I/O requests. This applies to both blocking and overlapped I/O, and occurs no matter the outcome of the I/O request (i.e. successful transfer or not).

### 2.8.1. Event Callback

Using the callback mechanism each notification source can be assigned a callback function. Each source can have a single callback with an application specific value passed as an argument. Callbacks can be assigned to any source and in any combination desired. If a given callback is associated with multiple sources, then multiple callbacks will be made as the different events occur. So, for example, if a single callback is assigned to two different interrupts,

then the callback function will be called separately for each interrupt, as often as each occurs. Since each source is associated with its own callback context, a thread context, such callbacks must support multithreaded operation. Applications are free to reconfigure callbacks during a callback context, but the callback for a given event must return before subsequent callback notification can occur for that same event. The prototype required for all callbacks is the data type `hpdi32_callback_func_t` (page 47). The three arguments to the callback are each U32 data types. Application must cast the values given to their respective types, which are described below. Refer to “Interrupt Parameter: Callback Function” on page 87 and “Interrupt Parameter: Callback Argument” on page 86.

### 2.8.1.1. Interrupt Notification Callback

The callback function arguments are described in the following table. The values received during the callback must be cast according to the data types specified.

Argument	Cast	Description
arg1	void*	This is the device handle received from <code>hpdi32_open()</code> (page 59).
arg2	U32	This is the specific “which” bit for the interrupt that produced the callback. Refer to the <code>HPDI32_WHICH_IRQ_XXX</code> macros (page 38).
arg3	U32	This is an application specific argument. This is the Interrupt Callback Argument parameter.

### 2.8.1.2. I/O Completion Notification Callback

The callback function arguments are described in the following table. The values received during the callback must be cast according to the data types specified.

Argument	Cast	Description
arg1	void*	This is the device handle received from <code>hpdi32_open()</code> (page 59)
arg2	U32	This is the applicable I/O status data. Refer to the <code>GSC_IO_STATUS_XXX</code> macros (page 38).
arg3	U32	This is an application specific argument. This is the I/O Callback Argument parameter.

## 2.8.2. Event Waiting

The waiting mechanism operates by blocking the calling thread until any one of a number of referenced events occurs. The calling thread is resumed when the first of the referenced events occurs, or when a timeout limit expires, whichever occurs first. The time limit is passed as an argument to the wait service. Threads can wait on any number or combinations of interrupts, or either or both I/O directions, but the two sources cannot be combined. Also, any number of threads can wait on identical or different events. All are resumed when a referenced event occurs. Refer to “`hpdi32_io_wait()`” on page 56 and “`hpdi32_irq_wait()`” on page 57.

### 3. Macros

The HPDI32 API includes the following macros. The headers also contain various other utility type macros, which are provided without documentation. Parameter support macros are not presented in this subsection. These macros are described in section 6 beginning on page 70.

#### 3.1. API Version Number

This macro defines the version number of the API's executable interface. It does not refer to the SDK version number, the API Library version number or the Device Driver version number. Applications pass this value to the function `hpdi32_api_status()` (page 49), which is used to verify that the application and the library are compatible.

Macros	Description
HPDI32_API_VERSION	This is the API's overall version number.

#### 3.2. Common Parameter Assignment Values

The below macros define universal values understood by all parameters to have special meanings, as given below. Any time a parameter assignment request is being carried out, use of these macros as the assignment value will produce the results given here.

Macros	Description
GSC_DEFAULT	Set the parameter to its default state/value. This is equivalent to using the explicitly defined default macro for the respective parameter.
GSC_NO_CHANGE	Do not change the parameter's state/value. Since parameter access follows a set-then-get model, this value can be used to achieve a get only operation.

#### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_io_tx_timeout_reset(void* handle, int verbose)
{
    unsigned long    get;
    U32              status;

    // Reset the Tx I/O timeout period to its default.
    status = hpdi32_config(handle,
                          HPDI32_IO_TIMEOUT,
                          HPDI32_WHICH_TX,
                          GSC_DEFAULT,
                          &get);

    if (!verbose)
    {
    }
    else if (status == GSC_SUCCESS)
    {
        printf("hpdi32_config() failure: %ld\n", (long) status);
    }
    else
    {
    }
}
```

```

    {
        printf("Tx Timeout: %lu seconds\n", (long) get);
    }

    return(status);
}

```

### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_io_tx_timeout_get(
    void*          handle,
    unsigned long* timeout_secs,
    int            verbose)
{
    U32 status;

    // Retrieve the Tx I/O timeout period without changing it.
    status = hpdi32_config(handle,
                           HPDI32_IO_TIMEOUT,
                           HPDI32_WHICH_TX,
                           GSC_NO_CHANGE,
                           timeout_secs);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_config() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Tx Timeout: %lu seconds\n", (long) timeout_secs[0]);
    }

    return(status);
}

```

## 3.3. Discrete Data Type Options

The below macros are defined by application code as needed to disable declarations for and size validation for the data types S8, U8, S16, U16, S32 and U32. The API declares these data types by default, but applications can disable this as needed.

Macros	Description
GSC_DATA_TYPES_CHECK	If the API declares the data types and the application defines this macro, then the data type sizes will be validated during the application's build process. This macro should only be defined if the compiler in use supports the <code>sizeof()</code> macro during preprocessing.
GSC_DATA_TYPES_NOT_NEEDED	Applications should define this macro before including <code>hpdi32_api.h</code> to disable the declarations for these data types.

### 3.4. I/O Status Fields

This set of macros applies to the 32-bit value reported when requesting the status of an I/O operation. The value reported includes a direction bit, a status field and a count field. The completion status of the operation is obtained by looking only at the GSC\_IO\_STATUS\_MASK bits from the I/O status value. All other bits refer to other than the completion status. The accompanying sample code illustrates how the I/O status could be utilized.

Fields	Description
GSC_IO_STATUS_COUNT_MASK	This macro applies to the count field, which covers the lower set of status bits. The count is zero while the operation is in progress and, once ended, indicates the number of bytes successfully transferred. This macro also identifies the maximum number of bytes that can be transferred in a single I/O request. The count is only guaranteed to be accurate when an operation completes with all data being successfully transferred.
GSC_IO_STATUS_MASK	This macro applies to the I/O completion status field. Apply this mask to the I/O status value (bitwise AND) to get the completion status. Supported completion status values are given in the below table.
GSC_IO_STATUS_TX	If this bit is set then the operation was from a write request to the device. If not set, then the operation was a read request from the device.

The following defines the I/O completion status options. These values are obtained by performing a bitwise AND of the overall status with the I/O completion status mask above.

Macros	Description
GSC_IO_STATUS_ABORTED	This indicates that the operation ended due to an abort request. This arises either from an application's explicit abort request, or from a reset or initialization request. The count field may be inaccurate when this status is reported.
GSC_IO_STATUS_ACTIVE	This indicates that the operation is still in progress. If the status is other than this value, then the I/O operation is no longer in progress.
GSC_IO_STATUS_ERROR	This indicates that the operation ended due to an error condition, which can arise for any number of reasons. The count field may be inaccurate when this status is reported.
GSC_IO_STATUS_SUCCESS	This indicates that the operation completed successfully. The count field is accurate when this status is reported.
GSC_IO_STATUS_TIMEOUT	This indicates that the operation ended because the timeout period lapsed. The count field may be inaccurate when this status is reported.

#### Example

```
#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

long hpdi32_dsl_io_status_evaluate(U32 io_status)
{
    long    bytes;
    U32     status = io_status & GSC_IO_STATUS_MASK;

    if (status == GSC_IO_STATUS_ACTIVE)
    {
        // The operation is still active.
        bytes = 0;
    }
    else if (status == GSC_IO_STATUS_SUCCESS)
    {

```

```

        // No operation has been requested.
        bytes = (long) (io_status & GSC_IO_STATUS_COUNT_MASK);
    }
    else if (status == GSC_IO_STATUS_TIMEOUT)
    {
        // The timeout period lapsed.
        // The count may not be accurate.
        bytes = -1;
    }
    else if (status == GSC_IO_STATUS_ERROR)
    {
        // There was an error.
        // The count may not be accurate.
        bytes = -1;
    }
    else if (status == GSC_IO_STATUS_ABORTED)
    {
        // The operation was aborted.
        // The count may not be accurate.
        bytes = -1;
    }
    else
    {
        // Unknown status.
        bytes = -1;
    }
}

return(bytes);
}

```

### 3.5. Maximum Number of Open Handles

This macro defines the maximum number of device handles that can be opened at any one time. All open handles are unique even if they refer to the same device, though handles are reused once closed.

Macros	Description
GSC_PROCESS_OPEN_MAX	This defines the maximum number of open handles.

### 3.6. Parameter Access “Which” Bits

The table below lists the set of selection bits that may be set when a configuration parameter is modified or accessed. They are referred to as “which” bits in that they specify the objects which the parameter is to access. When appropriate, bits within the same category may be bitwise or’d in order to apply the action to multiple objects. For retrieval purposes, only the data for the last object successfully accessed is retrieved. The bits’ use is explained along with the parameters that each is associated with, and appears in subsequent portions of this document.

**NOTE:** The interrupt related “which” bits include both general and specific definitions for those cable signals which may have dual functionality. These are for reference and usability purposes only and do not refer to different interrupts. In addition, use of any particular definition will not alter which functionality is active at any particular time.

**NOTE:** Some of the “which” bit macros end with an underscore (“\_”). This is added to convey to users that the respective cable signals are dual function; data Flow Control and GPIO. The respective cable signals are also represented by additional macros representing the specific functionalities.

Macros	Description
HPDI32_WHICH_AE	This specifies the Almost Empty level for the FIFOs. *
HPDI32_WHICH_AF	This specifies the Almost Full level for the FIFOs. *
HPDI32_WHICH_COMMAND_0_	This specifies the Cable Command 0, which may be either Frame Valid or GPIO 6. *
HPDI32_WHICH_COMMAND_1_	This specifies the Cable Command 1, which may be either Line Valid or GPIO 0. *
HPDI32_WHICH_COMMAND_2_	This specifies the Cable Command 2, which may be either Status Valid or GPIO 1. *
HPDI32_WHICH_COMMAND_3_	This specifies the Cable Command 3, which may be either Rx Ready or GPIO 2. *
HPDI32_WHICH_COMMAND_4_	This specifies the Cable Command 4, which may be either Tx Ready or GPIO 3. *
HPDI32_WHICH_COMMAND_5_	This specifies the Cable Command 5, which may be either Tx Enabled or GPIO 4. *
HPDI32_WHICH_COMMAND_6_	This specifies the Cable Command 6, which may be either Rx Enabled or GPIO 5. *
HPDI32_WHICH_IRQ_C0A_	This specifies the Cable Command 0 interrupt that defaults to triggering when the signal is active. This refers either to the Frame Valid Begin or GPIO 6 High.
HPDI32_WHICH_IRQ_C0I_	This specifies the Cable Command 0 interrupt that defaults to triggering when the signal is inactive. This refers either to the Frame Valid End or GPIO 6 Low.
HPDI32_WHICH_IRQ_C1_	This specifies the Cable Command 1 interrupt, which refers either to Line Valid or GPIO 0.
HPDI32_WHICH_IRQ_C2_	This specifies the Cable Command 2 interrupt, which refers either to Status Valid or GPIO 1.
HPDI32_WHICH_IRQ_C3_	This specifies the Cable Command 2 interrupt, which refers either to Tx Ready or GPIO 2.
HPDI32_WHICH_IRQ_C4_	This specifies the Cable Command 2 interrupt, which refers either to Rx Ready or GPIO 3.
HPDI32_WHICH_IRQ_C5_	This specifies the Cable Command 2 interrupt, which refers either to Tx Enabled or GPIO 4.
HPDI32_WHICH_IRQ_C6_	This specifies the Cable Command 2 interrupt, which refers either to Rx Enabled or GPIO 5.
HPDI32_WHICH_IRQ_RX_AE	This specifies the Rx FIFO Almost Empty interrupt.
HPDI32_WHICH_IRQ_RX_AF	This specifies the Rx FIFO Almost Full interrupt.
HPDI32_WHICH_IRQ_RX_E	This specifies the Rx FIFO Empty interrupt.
HPDI32_WHICH_IRQ_RX_F	This specifies the Rx FIFO Full interrupt.
HPDI32_WHICH_IRQ_TX_AE	This specifies the Tx FIFO Almost Empty interrupt.
HPDI32_WHICH_IRQ_TX_AF	This specifies the Tx FIFO Almost Full interrupt.
HPDI32_WHICH_IRQ_TX_E	This specifies the Tx FIFO Empty interrupt.
HPDI32_WHICH_IRQ_TX_F	This specifies the Tx FIFO Full interrupt.
HPDI32_WHICH_RX	This specifies that the receiver is to be accessed, such as the Rx FIFO. *
HPDI32_WHICH_TX	This specifies that the transmitter is to be accessed, such as the Tx FIFO. *

\* Other macros are also defined that include other logical combinations or representations of some bits. Additional macros may also be defined using alternate representations of the same source. For example, Cable Command 0 is also referred to as Frame Valid and GPIO 6.

### 3.7. Registers

The following tables give the complete set of HPDI32 registers. The tables are divided by register categories. There are PCI registers which differ slightly between the 32-bit and the 64-bit boards, PLX feature set registers which also

differ slightly between the 32-bit and the 64-bit boards, and there are GSC firmware based registers. The PCI registers and the PLX registers are provided by the PCI interface chips used on the HPDI32. Applications have read access to all registers, but write access only to the GSC firmware registers.

### 3.7.1. GSC Registers

The following table gives the complete set of GSC specific HPDI32 registers. For detailed definitions of these registers refer to the applicable *HPDI32 User Manual*.

Macros	Description
HPDI32_BCR	Board Control Register (BCR)
HPDI32_BSR	Board Status Register (BSR)
HPDI32_FDR	FIFO Data Register (FDR)
HPDI32_FRR	Firmware Revision Register (FRR)
HPDI32_FSR	Feature Set Register (FSR)
HPDI32_ICR	Interrupt Control Register (ICR)
HPDI32_IELR	Interrupt Edge/Level Register (IELR)
HPDI32_IHLR	Interrupt High/Low Register (IHLR)
HPDI32_ISR	Interrupt Status Register (ISR)
HPDI32_RAR	Rx Almost Register (RAR)
HPDI32_RFSR	Rx FIFO Size Register (RFSR)
HPDI32_RFWR	Rx FIFO Words Register (RFWR)
HPDI32_RLCR	Rx Line Counter Register (RLCR)
HPDI32_RSCR	Rx Status Counter Register (RSCR)
HPDI32_TAR	Tx Almost Register (TAR)
HPDI32_TCDR	Tx Clock Divider Register (TCDR)
HPDI32_TFSR	Tx FIFO Size Register (TFSR)
HPDI32_TFWR	Tx FIFO Words Register (TFWR)
HPDI32_TLILCR	Tx Line Invalid Length Count Register (TLILCR)
HPDI32_TLVLCR	Tx Line Valid Length Count Register (TLVLCR)
HPDI32_TSVLCR	Tx Status Valid Length Count Register (TSVLCR)

### 3.7.2. PLX PCI9080 PCI Configuration Registers

The following table gives the set of PCI Configuration Registers available on 32-bit boards. These registers are present on 64-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9080 Data Book*.

Macros	Description
GSC_PCI_9080_BAR0	PCI Base Address Register for Memory Accesses to Local, Runtime, and DMA Registers (PCIBAR0)
GSC_PCI_9080_BAR1	PCI Base Address Register for I/O Accesses to Local, Runtime, and DMA Registers (PCIBAR1)
GSC_PCI_9080_BAR2	PCI Base Address Register for Memory Accesses to Local Address Space 0 (PCIBAR2)
GSC_PCI_9080_BAR3	PCI Base Address Register for Memory Accesses to Local Address Space 1 (PCIBAR3)
GSC_PCI_9080_BAR4	Unused Base Address Register (PCIBAR4)
GSC_PCI_9080_BAR5	Unused Base Address Register (PCIBAR5)
GSC_PCI_9080_BISTR	PCI Built-In Self Test Register (PCIBISTR)
GSC_PCI_9080_CCR	PCI Class Code Register (PCICCR)
GSC_PCI_9080_CIS	PCI Cardbus CIS Pointer Register (PCICIS)
GSC_PCI_9080_CLSR	PCI Cache Line Size Register (PCICLSR)
GSC_PCI_9080_CR	PCI Command Register (PCICR)



GSC_PCI_9080_DIDR	PCI Device ID Register (PCIDIDR)
GSC_PCI_9080_ERBAR	PCI Expansion ROM Base Address (PCIERBAR)
GSC_PCI_9080_HTR	PCI Header Type Register (PCIHTR)
GSC_PCI_9080_ILR	PCI Interrupt Line Register (PCIILR)
GSC_PCI_9080_IPR	PCI Interrupt Pin Register (PCIIPR)
GSC_PCI_9080_LTR	PCI Latency Timer Register (PCILTR)
GSC_PCI_9080_MGR	PCI Min Gnt Register (PCIMGR)
GSC_PCI_9080_MLR	PCI Max Lat Register (PCIMLR)
GSC_PCI_9080_REV	PCI Revision ID Register (PCIREV)
GSC_PCI_9080_SID	PCI Subsystem ID Register (PCISID)
GSC_PCI_9080_SR	PCI Status Register (PCISR)
GSC_PCI_9080_SVID	PCI Subsystem Vendor ID Register (PCISVID)
GSC_PCI_9080_VIDR	PCI Vendor ID Register (PCIVIDR)

**NOTE:** The following table gives register identification information for 32-bit HPDI32 boards. There are some DIO24 variations that identify themselves as members of the HPDI32 product family. To distinguish these DIO24 variations from HPDI32s, refer to the Firmware Revision Registers for the respective DIO24.

Register	Value	Description
GSC_PCI_9080_VIDR	0x10B5	The PCI interface chip as a PLX device.
GSC_PCI_9080_DIDR	0x9080	The PCI interface chip as a PLX PCI9080.
GSC_PCI_9080_SVID	0x10B5	The below register value has been assigned by PLX.
GSC_PCI_9080_SID	0x2400	The device is an HPDI32 family product.

### 3.7.3. PLX PCI9080 Feature Set Registers

The following tables give the set of PLX feature set registers available on 32-bit boards.

#### Local Configuration Registers

The following table gives the set of PLX Local Configuration Registers available on 32-bit boards. These registers are present on 64-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9080 Data Book*.

Macros	Description
GSC_PLX_9080_BIGEND	Big/Little Endian Descriptor Register (BIGEND)
GSC_PLX_9080_DMCFGA	PCI Configuration Address Register for Direct Master to PCI IO/CFG (DMCFG A)
GSC_PLX_9080_DMLBAM	Local Bus Base Address Register for Direct Master to PCI Memory (DMLBAM)
GSC_PLX_9080_DMLBAI	Local Bus Base Address Register for Direct Master to PCI IO/CFG (DMLBAI)
GSC_PLX_9080_DMPBAM	PCI Base Address Register for Direct Master to PCI Memory (DMPBAM)
GSC_PLX_9080_DMRR	Local Range Register for Direct Master to PCI (DMRR)
GSC_PLX_9080_EROMBA	Expansion ROM Local Base Address Register (EROMBA)
GSC_PLX_9080_EROMRR	Expansion ROM Range Register (EROMRR)
GSC_PLX_9080_LAS0BA	Local Address Space 0 Local Base Address Register (LAS0BA)
GSC_PLX_9080_LAS0RR	Local Address Space 0 Range Register for PCI-to-Local Bus (LAS0RR)
GSC_PLX_9080_LAS1BA	Local Address Space 1 Local Base Address Register (LAS1BA)
GSC_PLX_9080_LAS1RR	Local Address Space 1 Range Register for PCI-to-Local Bus (LAS1RR)
GSC_PLX_9080_LBRD0	Local Address Space 0/Expansion ROM Bus Region Descriptor Register (LBRD0)
GSC_PLX_9080_LBRD1	Local Address Space 1 Bus Region Descriptor Register (LBRD1)
GSC_PLX_9080_MARBR	Mode Arbitration Register (MARBR)

## Runtime Registers

The following table gives the set of PLX Runtime Registers available on 32-bit boards. These registers are present on 64-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9080 Data Book*.

Macros	Description
GSC_PLX_9080_CNTRL	Serial EEPROM Control, CPI Command Codes, User I/O, Init Control Register (CNTRL)
GSC_PLX_9080_INTCSR	Interrupt Control/Status Register (INTCSR)
GSC_PLX_9080_L2PDBELL	Local-to-PCI Doorbell Register (L2PDBELL)
GSC_PLX_9080_MBOX0	Mailbox Register 0 (MBOX0)
GSC_PLX_9080_MBOX1	Mailbox Register 1 (MBOX1)
GSC_PLX_9080_MBOX2	Mailbox Register 2 (MBOX2)
GSC_PLX_9080_MBOX3	Mailbox Register 3 (MBOX3)
GSC_PLX_9080_MBOX4	Mailbox Register 4 (MBOX4)
GSC_PLX_9080_MBOX5	Mailbox Register 5 (MBOX5)
GSC_PLX_9080_MBOX6	Mailbox Register 6 (MBOX6)
GSC_PLX_9080_MBOX7	Mailbox Register 7 (MBOX7)
GSC_PLX_9080_P2LDBELL	PCI-to-Local Doorbell Register (P2LDBELL)
GSC_PLX_9080_PCIHIDR	PCI Permanent Configuration ID Register (PCIHIDR)
GSC_PLX_9080_PCIHREV	PCI Permanent Revision ID Register (PCIHREV)

## DMA Registers

The following table gives the set of PLX DMA Registers available on 32-bit boards. These registers are present on 64-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9080 Data Book*.

Macros	Description
GSC_PLX_9080_DMAARB	DMA Arbitration Register (DMAARB)
GSC_PLX_9080_DMACSR0	DMA Channel 0 Command/Status Register (DMACSR0)
GSC_PLX_9080_DMACSR1	DMA Channel 1 Command/Status Register (DMACSR1)
GSC_PLX_9080_DMADPR0	DMA Channel 0 Descriptor Pointer Register (DMADPR0)
GSC_PLX_9080_DMADPR1	DMA Channel 1 Descriptor Pointer Register (DMADPR1)
GSC_PLX_9080_DMALADR0	DMA Channel 0 Local Address Register (DMALADR0)
GSC_PLX_9080_DMALADR1	DMA Channel 1 Local Address Register (DMALADR1)
GSC_PLX_9080_DMAMODE0	DMA Channel 0 Mode Register (DMAMODE0)
GSC_PLX_9080_DMAMODE1	DMA Channel 1 Mode Register (DMAMODE1)
GSC_PLX_9080_DMAPADR0	DMA Channel 0 PCI Address Register (DMAPADR0)
GSC_PLX_9080_DMAPADR1	DMA Channel 1 PCI Address Register (DMAPADR1)
GSC_PLX_9080_DMASIZ0	DMA Channel 0 Transfer Size Register (DMASIZ0)
GSC_PLX_9080_DMASIZ1	DMA Channel 1 Transfer Size Register (DMASIZ1)
GSC_PLX_9080_DMATHR	DMA Threshold Register (DMATHR)

## Message Queue Registers

The following table gives the set of PLX Messaging Queue Registers available on 32-bit boards. These registers are present on 64-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9080 Data Book*.

Macros	Description
GSC_PLX_9080_IFHPR	Inbound Free Head Pointer Register (IFHPR)
GSC_PLX_9080_IFTPR	Inbound Free Tail Pointer Register (IFTPR)

GSC_PLX_9080_IPHPR	Inbound Post Head Pointer Register (IPHPR)
GSC_PLX_9080_IPTPR	Inbound Post Tail Pointer Register (IPTPR)
GSC_PLX_9080_IQP	Inbound Queue Port Register (IQP)
GSC_PLX_9080_MQCR	Messaging Queue Configuration Register (MQCR)
GSC_PLX_9080_OFHPR	Outbound Free Head Pointer Register (OFHPR)
GSC_PLX_9080_OFTPR	Outbound Free Tail Pointer Register (OFTPR)
GSC_PLX_9080_OPHPR	Outbound Post Head Pointer Register (OPHPR)
GSC_PLX_9080_OPLFIM	Outbound Post List FIFO Interrupt Mask Register (OPLFIM)
GSC_PLX_9080_OPLFIS	Outbound Post List FIFO Interrupt Status Register (OPLFIS)
GSC_PLX_9080_OPTPR	Outbound Post Tail Pointer Register (OPTPR)
GSC_PLX_9080_OQP	Outbound Queue Port Register (OQP)
GSC_PLX_9080_QBAR	Queue Base Address Register (QBAR)
GSC_PLX_9080_QSR	Queue Status/Control Register (QSR)

### 3.7.4. PLX PCI9656 PCI Configuration Registers

The following table gives a subset of the PCI Configuration Registers available on 64-bit boards. These registers are present on 32-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

Macros	Description
GSC_PCI_9656_BAR0	PCI Base Address Register for Memory Accesses to Local, Runtime, and DMA Registers (PCIBAR0)
GSC_PCI_9656_BAR1	PCI Base Address Register for I/O Accesses to Local, Runtime, and DMA Registers (PCIBAR1)
GSC_PCI_9656_BAR2	PCI Base Address Register for Memory Accesses to Local Address Space 0 (PCIBAR2)
GSC_PCI_9656_BAR3	PCI Base Address Register for Memory Accesses to Local Address Space 1 (PCIBAR3)
GSC_PCI_9656_BAR4	Unused Base Address Register (PCIBAR4)
GSC_PCI_9656_BAR5	Unused Base Address Register (PCIBAR5)
GSC_PCI_9656_BISTR	PCI Built-In Self Test Register (PCIBISTR)
GSC_PCI_9656_CCR	PCI Class Code Register (PCICCR)
GSC_PCI_9656_CIS	PCI Cardbus CIS Pointer Register (PCICIS)
GSC_PCI_9656_CLSR	PCI Cache Line Size Register (PCICLSR)
GSC_PCI_9656_CR	PCI Command Register (PCICR)
GSC_PCI_9656_DIDR	PCI Device ID Register (PCIDIDR)
GSC_PCI_9656_ERBAR	PCI Expansion ROM Base Address (PCIERBAR)
GSC_PCI_9656_HTR	PCI Header Type Register (PCIHTR)
GSC_PCI_9656_ILR	PCI Interrupt Line Register (PCIILR)
GSC_PCI_9656_IPR	PCI Interrupt Pin Register (PCIIPR)
GSC_PCI_9656_LTR	PCI Latency Timer Register (PCILTR)
GSC_PCI_9656_MGR	PCI Min_Gnt Register (PCIMGR)
GSC_PCI_9656_MLR	PCI Max_Lat Register (PCIMLR)
GSC_PCI_9656_REV	PCI Revision ID Register (PCIREV)
GSC_PCI_9656_SID	PCI Subsystem ID Register (PCISID)
GSC_PCI_9656_SR	PCI Status Register (PCISR)
GSC_PCI_9656_SVID	PCI Subsystem Vendor ID Register (PCISVID)
GSC_PCI_9656_VIDR	PCI Vendor ID Register (PCIVIDR)

**NOTE:** The following table gives register identification information for 64-bit HPDI32 boards.

Register	Value	Description
GSC_PCI_9656_VIDR	0x10B5	The PCI interface chip as a PLX device.

GSC_PCI_9656_DIDR	0x9656	The PCI interface chip as a PLX PCI9656.
GSC_PCI_9656_SVID	0x10B5	The below register value has been assigned by PLX.
GSC_PCI_9656_SID	0x2705	The device is an HPDI32 family product.

The following table gives the remaining set of the PCI Configuration Registers available on 64-bit boards. These registers are not present on 32-bit boards. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

Macros	Description
GSC_PCI_9656_CAP_PTR	New Capability Pointer Register (CAP_PTR)
GSC_PCI_9656_HS_CNTL	Hot Swap Control Registers (HS_CNTL)
GSC_PCI_9656_HS_CSR	Hot Swap Control/Status Register (HS_CSR)
GSC_PCI_9656_HS_NEXT	Hot Swap Next Capability Pointer Register (HS_NEXT)
GSC_PCI_9656_PMC	Power Management Capabilities Register (PMC)
GSC_PCI_9656_PMCAPID	Power Management Capability ID Register (PMCAPID)
GSC_PCI_9656_PMCSR	Power Management Control/Status Register (PMCSR)
GSC_PCI_9656_PMCSR_BSE	PMCSR Bridge Support Expansions Register (PMCSR_BSE)
GSC_PCI_9656_PMDATA	Power Management Data Register (PMDATA)
GSC_PCI_9656_PMNEXT	Power Management Next Capability Pointer Register (PMNEXT)
GSC_PCI_9656_VPD_NEXT	PCI Vital Product Data Next Capability Pointer Register (PVPD_NEXT)
GSC_PCI_9656_VPDAD	PCI Vital Product Data Address Register (PVPDAD)
GSC_PCI_9656_VPDATA	PCI VPD Data Register (PVPDATA)
GSC_PCI_9656_VPDCNTL	PCI Vital Product Data Control Register (PVPDCNTL)

### 3.7.5. PLX PCI9656 Feature Set Registers

The following tables give the set of PLX feature set registers available on 64-bit boards.

#### Local Configuration Registers

The following table gives a subset of the PLX Local Configuration Registers available on 64-bit boards. These registers are present on 32-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

Macros	Description
GSC_PLX_9656_BIGEND	Big/Little Endian Descriptor Register (BIGEND)
GSC_PLX_9656_DMCFGA	PCI Configuration Address Register for Direct Master to PCI IO/CFG (DMCFGA)
GSC_PLX_9656_DMLBAM	Local Bus Base Address Register for Direct Master to PCI Memory (DMLBAM)
GSC_PLX_9656_DMLBAI	Local Bus Base Address Register for Direct Master to PCI IO/CFG (DMLBAI)
GSC_PLX_9656_DMPBAM	PCI Base Address Register for Direct Master to PCI Memory (DMPBAM)
GSC_PLX_9656_DMRR	Local Range Register for Direct Master to PCI (DMRR)
GSC_PLX_9656_EROMBA	Expansion ROM Local Base Address Register (EROMBA)
GSC_PLX_9656_EROMRR	Expansion ROM Range Register (EROMRR)
GSC_PLX_9656_LAS0BA	Local Address Space 0 Local Base Address Register (LAS0BA)
GSC_PLX_9656_LAS0RR	Local Address Space 0 Range Register for PCI-to-Local Bus (LAS0RR)
GSC_PLX_9656_LAS1BA	Local Address Space 1 Local Base Address Register (LAS1BA)
GSC_PLX_9656_LAS1RR	Local Address Space 1 Range Register for PCI-to-Local Bus (LAS1RR)
GSC_PLX_9656_LBRD0	Local Address Space 0/Expansion ROM Bus Region Descriptor Register (LBRD0)
GSC_PLX_9656_LBRD1	Local Address Space 1 Bus Region Descriptor Register (LBRD1)
GSC_PLX_9656_MARBR	Mode Arbitration Register (MARBR)

The following table gives the remaining set of the PLX Local Configuration Registers available on 64-bit boards. These registers are not present on 32-bit boards. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

Macros	Description
GSC_PLX_9656_ARB	PCI Arbiter Control Register (PCIARB)
GSC_PLX_9656_ABTAADR	PCI Abort Address Register (PABTAADR)
GSC_PLX_9656_DMDAC	Direct Master PCI Dual Address Cycle Upper Address Register (DMDAC)
GSC_PLX_9656_LMISC1	Local Miscellaneous Control 1 Register (LMISC1)
GSC_PLX_9656_LMISC2	Local Miscellaneous Control 2 Register (LMISC2)
GSC_PLX_9656_PROT_AREA	Serial EEPROM Write-Protected Address Boundary Register (PROT_AREA)

## Runtime Registers

The following table gives the set of PLX Runtime Registers available on 64-bit boards. These register definitions can be used interchangeably on both 32-bit and 64-bit HPDI32 boards.

Macros	Description
GSC_PLX_9656_CNTRL	Serial EEPROM Control, CPI Command Codes, User I/O, Init Control Register (CNTRL)
GSC_PLX_9656_INTCSR	Interrupt Control/Status Register (INTCSR)
GSC_PLX_9656_L2PDBELL	Local-to-PCI Doorbell Register (L2PDBELL)
GSC_PLX_9656_MBOX0	Mailbox Register 0 (MBOX0)
GSC_PLX_9656_MBOX1	Mailbox Register 1 (MBOX1)
GSC_PLX_9656_MBOX2	Mailbox Register 2 (MBOX2)
GSC_PLX_9656_MBOX3	Mailbox Register 3 (MBOX3)
GSC_PLX_9656_MBOX4	Mailbox Register 4 (MBOX4)
GSC_PLX_9656_MBOX5	Mailbox Register 5 (MBOX5)
GSC_PLX_9656_MBOX6	Mailbox Register 6 (MBOX6)
GSC_PLX_9656_MBOX7	Mailbox Register 7 (MBOX7)
GSC_PLX_9656_P2LDBELL	PCI-to-Local Doorbell Register (P2LDBELL)
GSC_PLX_9656_PCIHIDR	PCI Permanent Configuration ID Register (PCIHIDR)
GSC_PLX_9656_PCIHREV	PCI Permanent Revision ID Register (PCIHREV)

## DMA Registers

The following table gives a subset of the PLX DMA Registers available on 64-bit boards. These registers are present on 32-bit boards as well and the definitions can be used interchangeably on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

Macros	Description
GSC_PLX_9656_DMAARB	DMA Arbitration Register (DMAARB)
GSC_PLX_9656_DMACSR0	DMA Channel 0 Command/Status Register (DMACSR0)
GSC_PLX_9656_DMACSR1	DMA Channel 1 Command/Status Register (DMACSR1)
GSC_PLX_9656_DMADPR0	DMA Channel 0 Descriptor Pointer Register (DMADPR0)
GSC_PLX_9656_DMADPR1	DMA Channel 1 Descriptor Pointer Register (DMADPR1)
GSC_PLX_9656_DMALADR0	DMA Channel 0 Local Address Register (DMALADR0)
GSC_PLX_9656_DMALADR1	DMA Channel 1 Local Address Register (DMALADR1)
GSC_PLX_9656_DMAMODE0	DMA Channel 0 Mode Register (DMAMODE0)
GSC_PLX_9656_DMAMODE1	DMA Channel 1 Mode Register (DMAMODE1)
GSC_PLX_9656_DMAPADR0	DMA Channel 0 PCI Address Register (DMAPADR0)
GSC_PLX_9656_DMAPADR1	DMA Channel 1 PCI Address Register (DMAPADR1)
GSC_PLX_9656_DMASIZ0	DMA Channel 0 Transfer Size Register (DMASIZ0)
GSC_PLX_9656_DMASIZ1	DMA Channel 1 Transfer Size Register (DMASIZ1)
GSC_PLX_9656_DMATHR	DMA Threshold Register (DMATHR)

The following table gives the remaining set of the PLX DMA Registers available on 64-bit boards. These registers are not present on 32-bit boards. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

Macros	Description
GSC_PLX_9656_DMADAC0	DMA Channel 0 PCI Dual Address Cycle Upper Address Register (DMADAC0)
GSC_PLX_9656_DMADAC1	DMA Channel 1 PCI Dual Address Cycle Upper Address Register (DMADAC1)

### Message Queue Registers

The following table gives the set of PLX Messaging Queue Registers available on 64-bit boards. These register definitions can be used interchangeably on both 32-bit and 64-bit HPDI32 boards.

Macros	Description
GSC_PLX_9656_IFHPR	Inbound Free Head Pointer Register (IFHPR)
GSC_PLX_9656_IFTP	Inbound Free Tail Pointer Register (IFTPR)
GSC_PLX_9656_IPHPR	Inbound Post Head Pointer Register (IPHPR)
GSC_PLX_9656_IPTPR	Inbound Post Tail Pointer Register (IPTPR)
GSC_PLX_9656_IQP	Inbound Queue Port Register (IQP)
GSC_PLX_9656_MQCR	Messaging Queue Configuration Register (MQCR)
GSC_PLX_9656_OFHPR	Outbound Free Head Pointer Register (OFHPR)
GSC_PLX_9656_OFTPR	Outbound Free Tail Pointer Register (OFTPR)
GSC_PLX_9656_OPHPR	Outbound Post Head Pointer Register (OPHPR)
GSC_PLX_9656_OPLFIM	Outbound Post List FIFO Interrupt Mask Register (OPLFIM)
GSC_PLX_9656_OPLFIS	Outbound Post List FIFO Interrupt Status Register (OPLFIS)
GSC_PLX_9656_OPTPR	Outbound Post Tail Pointer Register (OPTPR)
GSC_PLX_9656_OQP	Outbound Queue Port Register (OQP)
GSC_PLX_9656_QBAR	Queue Base Address Register (QBAR)
GSC_PLX_9656_QSR	Queue Status/Control Register (QSR)

## 3.8. Version Data Selectors

This set of macros is used when requesting a version number and indicates which version number is desired. The macros are passed as the `id` argument to the `hpdi32_version_get()` function (see page 66). The second table below lists utility macros used to retrieve each of the respective version numbers. In the second table, the argument `h` refers to the handle used to access the device, the `b` refers to an application buffer where the version string is recorded, and the `s` is the size of that buffer.

Macros (Values)	Description
GSC_VERSION_LIBRARY	This requests the library's version number.
GSC_VERSION_DRIVER	This requests the driver's version number.

Macro (Services)	Description
HPDI32_VERSION_GET_LIBRARY( <code>h, b, s</code> )	This requests the version number for the API Library.
HPDI32_VERSION_GET_DRIVER( <code>h, b, s</code> )	This requests the version number for the Device Driver.

## 4. Data Types

The interface includes the following data types.

### 4.1. Discrete Data Types

The following discrete data types are defined and used by the API. If an HPDI32 application includes other headers which also define these types, then the API can be directed to omit these definitions. This is done by defining the macro `GSC_DATA_TYPES_NOT_NEEDED` before including the API header. The alternate definitions must however define these types as listed in the below table.

Data Type	Description
S8	This is an 8-bit signed integer.
U8	This is an 8-bit unsigned integer.
S16	This is a 16-bit signed integer.
U16	This is a 16-bit unsigned integer.
S32	This is a 32-bit signed integer.
U32	This is a 32-bit unsigned integer.

### 4.2. `hpdi32_callback_func_t`

This is the data type required for all event notification callback functions. This applies both to Interrupt Notification callbacks and I/O Completion callbacks.

Definition

```
typedef void (*hpdi32_callback_func_t)(U32 arg1, U32 arg2, U32 arg3);
```

Arguments	Description
arg1	This is the device handle cast to a U32 data type.
arg2	For Interrupt Notification this is the HPDI32_WHICH_XXX bit for the respective interrupt. For I/O Completion Notification this is the applicable GSC_IO_STATUS_XXX status components.
arg3	This is any arbitrary application supplied data value.

### 4.3. Status Values

This unnamed enumerated data type lists all possible status values returnable from API service calls. The enumerated values represent common definitions used across all of GSC's PLX based API Libraries and many values will never be encountered when using the HPDI32 API Library. The table below gives brief descriptions for many values and omits those that should never be seen with the API. The most common value encountered is `GSC_SUCCESS` and indicates the request was completed successfully.

Definition

```
typedef enum
{
    ...
};
```

Values	Description
GSC_ABORTED	An I/O operation was aborted due to a user's explicit or implicit request.

GSC_ACCESS_DENIED	The operation failed because access to a device, service or system resource or service was denied.
GSC_DMA_CHANNEL_UNAVAILABLE	An operation failed because a DMA channel was unavailable.
GSC_FAILED	An operation failed in a non-specific manner.
GSC_INIT_FAILURE	API Library initialization failed.
GSC_INSUFFICIENT_RESOURCES	An operation failed because insufficient OS resources were available.
GSC_INVALID_API_HANDLE	An operation failed because the application supplied an invalid device handle. API device handles are API specific resources and are of no meaning to the OS.
GSC_INVALID_DATA	An operation failed because invalid data was provided.
GSC_INVALID_VERSION_API	API Library initialization failed because the API Library version was incompatible. This refers either to the API's version number or the GSC revision level. The version data can still be retrieved when this status is seen.
GSC_INVALID_VERSION_DRIVER	API Library initialization failed because the Device Driver version was incompatible. This refers either to the driver's version number or the GSC revision level. The version data can still be retrieved when this status is seen.
GSC_NULL_PARAM	An operation failed because an argument was NULL.
GSC_SUCCESS	An operation completed successfully.
GSC_THREAD_FAILURE	An operation (hpdi32_open()) failed because a support thread could not be started.
GSC_TOO_MANY_OPEN_HANDLES	An operation (hpdi32_open()) failed because the application attempted too many simultaneous device accesses.
GSC_UNSUPPORTED_FUNCTION	An operation failed because the application requested a service that is unsupported or unimplemented.
GSC_WAIT_TIMEOUT	An operation completed because a timeout period lapsed.
GSC_WAIT_CANCELED	An operation waiting for an event ended prematurely. This usually means the application was terminated while waiting for the event.



## 5. Functions

The HPDI32 API includes the following functions. The SDK interface also includes a number of function style macro definitions. These macros are described in section 6 beginning on page 70.

### 5.1. hpdi32\_api\_status()

This function is the entry point to determine the status of the API Library. This must be the very first call into the API and determines the usability of the API Library and the Device Driver. If the initial status obtained is other than GSC\_SUCCESS, then only a limited portion of the API is functional. If not fully usable, then both values returned may be useful in resolving the situation. Thereafter, the status obtained might vary if the API encounters irregular circumstances.

#### Prototype

```
U32 hpdi32_api_status(U32* stat, U32* arg, U32 api_ver);
```

Argument	Description
stat	The API records the current API status here, which can change during use. The pointer must not be NULL.
arg	The API records auxiliary status information here, which can change during use. This value should be related to the above reported status. The pointer must not be NULL.
api_ver	This must be the version number of the API the application was written for. If this number does not match, then the API is unusable by the application.

Return Value	Description
GSC_SUCCESS	The operation succeeded (the status was retrieved).
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

#### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_api_status(int verbose)
{
    U32 arg;
    U32 stat;
    U32 status;

    status = hpdi32_api_status(&stat, &arg, HPDI32_API_VERSION);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_api_status() failure: %ld\n", (long) status);
    }
    else
    {
        status = stat;
        printf("API Status:\n");
    }
}
```

```

        printf("    Status:    0x%lX\n", (long) stat);
        printf("    Argument: 0x%lX\n", (long) arg);
    }

    return(status);
}

```

## 5.2. hpdi32\_board\_count()

This function is the entry point to determine the number of HPDI32 boards installed in the system and accessible to the API. This service can be called without requiring access to any particular device.

### Prototype

```
U32 hpdi32_board_count(U8* count);
```

Argument	Description
count	The API records the number of board at this location. This pointer must not be NULL.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_board_count(U8* count, int verbose)
{
    U32 status;

    status = hpdi32_board_count(count);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_board_count() failure: %ld\n", (long) status);
    }
    else
    {
        printf("HPDI32 Board Count: %d\n", (int) count[0]);
    }

    return(status);
}

```

### 5.3. hpdi32\_close()

This function is the entry point to close a connection to an open HPDI32 board. The function should only be called after a successful open of the respective device via `hpdi32_open()` and must not be used after being closed. Before returning, the API returns the device to the same state produced when originally opened.

#### Prototype

```
U32 hpdi32_close(void* handle);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

#### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_close(void* handle, int verbose)
{
    U32 status;

    status = hpdi32_close(handle);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_close() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Device Closed: 0x%lX\n", (long) handle);
    }

    return(status);
}
```

### 5.4. hpdi32\_config()

This function is the entry point to accessing an individual parameter where all pertinent data is given as separate arguments. The function should only be called after a successful open of the respective device via `hpdi32_open()`.

#### Prototype

```
U32 hpdi32_config(
    void* handle,
```

```

U32    parm,
U32    which,
U32    set,
U32*   get);

```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
parm	This specifies the parameter to be accessed.
which	This is any number or combination of parameter specific <code>HPDI32_WHICH_XXX</code> bits that specify object(s) the parameter is applied to. Many parameters ignore this argument. When it is used, a value of zero is acceptable, and merely specifies to access none of the corresponding objects.
set	This is the value to apply to the parameter being accessed. The universal value <code>GSC_NO_CHANGE</code> specifies that the parameter not be altered and must be used when the purpose of the access is to get the current setting. Some parameters are read-only, in which case this argument is ignored.
get	After applying any changes to the parameter, its current setting is recorded here. When the “which” argument specifies multiple objects, only the last accessed is recorded here. This argument may be <code>NULL</code> , in which case the current setting is not retrieved.

Return Value	Description
<code>GSC_SUCCESS</code>	The operation succeeded.
Otherwise ...	A <code>GSC_XXX</code> error status reflecting the problem encountered.

### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_io_tx_timeout_set(
    void*    handle,
    U32      timeout_s,
    int      verbose)
{
    unsigned long    get;
    U32              status;

    status = hpdi32_config(handle,
                           HPDI32_IO_TIMEOUT,
                           HPDI32_WHICH_TX,
                           timeout_s,
                           &get);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_config() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Tx Timeout:\n");
    }
}

```

```

        printf("  Set: 0x%lX\n", (long) timeout_s);
        printf("  Get: 0x%lX\n", (long) get);
    }

    return(status);
}

```

## 5.5. hpdi32\_gpio\_mod()

This function is the entry point to performing a read-modify-write on the cable signals configured for General Purpose I/O. Only cable signals configured as GPIO are affected. All non-GPIO cable signals are unaffected. The function should only be called after a successful open of the respective device via `hpdi32_open()`.

### Prototype

```
U32 hpdi32_gpio_mod(void* handle, U8 value, U8 mask);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
value	This is the desired value to apply. Bits which are outside the GPIO range or which correspond to non-GPIO cable signals are ignored.
mask	This specifies the “value” bits to modify. If a bit is set here, then the corresponding “value” bit will be applied. The remaining “value” bits are ignored. Bits which are outside the GPIO range or which correspond to non-GPIO cable signals are ignored.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_gpio_0_mod(void* handle, U8 value, int verbose)
{
    U8 mask    = 0x1;
    U32 status;

    status = hpdi32_gpio_mod(handle, value, mask);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_gpio_mod() failure: %ld\n", (long) status);
    }
    else
    {
        printf("GPIO Modify:\n");
        printf("  Value: 0x%lX\n", (long) value);
        printf("  Mask:  0x%lX\n", (long) mask);
    }
}

```

```

    }

    return(status);
}

```

## 5.6. hpdi32\_gpio\_read()

This function is the entry point to reading the value from the cable signals configured for General Purpose I/O. Only cable signals configured as GPIO return actual values. All non-GPIO cable signals are returned as zero. The function should only be called after a successful open of the respective device via `hpdi32_open()`.

### Prototype

```
U32 hpdi32_gpio_read(void* handle, U8* value);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
value	The value read is recorded here. Only bits which are inside the GPIO range and which correspond to GPIO cable signals return actual values. All others return zero.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_gpio_read(void* handle, U8* value, int verbose)
{
    U32 status;

    status = hpdi32_gpio_read(handle, value);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_gpio_read() failure: %ld\n", (long) status);
    }
    else
    {
        printf("GPIO Read: 0x%lX\n", (long) value[0]);
    }

    return(status);
}

```

## 5.7. hpdi32\_gpio\_write()

This function is the entry point to writing to the cable signals configured for General Purpose I/O. Only cable signals configured as GPIO are affected. All non-GPIO cable signals are unaffected. The function should only be called after a successful open of the respective device via `hpdi32_open()`.

### Prototype

```
U32 hpdi32_gpio_write(void* handle, U8 value);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
value	This is the value to write. Only bits which are inside the GPIO range and which correspond to GPIO cable signals are affected. All others are unaffected.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_gpio_write(void* handle, U8 value, int verbose)
{
    U32 status;

    status = hpdi32_gpio_write(handle, value);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_gpio_write() failure: %ld\n", (long) status);
    }
    else
    {
        printf("GPIO Write: 0x%lX\n", (long) value);
    }

    return(status);
}
```

## 5.8. hpdi32\_init()

This function is the entry point to return a device and all parameters to the state produced when the device was first opened. In doing this, any I/O operations in progress are aborted. This function should only be called after a successful open of the respective device via `hpdi32_open()`.

## Prototype

```
U32 hpdi32_init(void* handle);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

## Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_init(void* handle, int verbose)
{
    U32 status;

    status = hpdi32_init(handle);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_init() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Device Initialized: 0x%lX\n", (long) handle);
    }

    return(status);
}
```

## 5.9. hpdi32\_io\_wait()

This function is the entry point to pause thread execution until an I/O operation completes. The function should only be called after a successful open of the respective device via `hpdi32_open()`. When called, the current thread will block until a specified I/O read or write operation completes. The waiting will occur if no I/O operations are currently active, or if an I/O operation is active in either blocking or overlapped mode. The call will return as soon as the time period expires, or when the first referenced operation completes, whether by an abort request, a failed I/O request, a timeout or successful data transfer. There is no limit to the number of threads that may simultaneously utilize this service or on the combination of operations that may be referenced.

## Prototype

```
U32 hpdi32_io_wait(void* handle, U32 which, U32 timeout_ms);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .



which	This is any bitwise or'd combination of HPDI32_WHICH_TX or HPDI32_WHICH_RX. Set HPDI32_WHICH_TX to wait on a write operation. Set HPDI32_WHICH_RX to wait on a read operation. If neither is set the function returns immediately with GSC_SUCCESS.
timeout_ms	This is the timeout limit in milliseconds. If an I/O operation does not complete within this time period, then the call returns at the end of the period. The timeout period will be at least the amount of time specified, but may be longer depending on the OS.

Return Value	Description
GSC_SUCCESS	Either no I/O operation was referenced or one of the referenced operations completed. No indication is given to indicate which event, if any, caused the call to return.
GSC_WAIT_TIMEOUT	The timeout period expired before completion of a referenced I/O operation.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_io_tx_wait(void* handle, U32 timeout_ms, int verbose)
{
    U32 status;

    status = hpdi32_io_wait(handle, HPDI32_WHICH_TX, timeout_ms);

    if (!verbose)
    {
    }
    else if (status == GSC_SUCCESS)
    {
        printf("Tx Wait: write operation completed.\n");
    }
    else if (status == GSC_WAIT_TIMEOUT)
    {
        printf("Tx Wait: timeout after %ld milliseconds\n",
            (long) timeout_ms);
    }
    else
    {
        printf("hpdi32_io_wait() failure: %ld\n", (long) status);
    }

    return(status);
}
```

## 5.10. hpdi32\_irq\_wait()

This function is the entry point to pause thread execution until an interrupt occurs. The function should only be called after a successful open of the respective device via `hpdi32_open()`. When called, the current thread will block until any one of a specified set of interrupts occurs. The call will return as soon as the time period expires, or when the first referenced interrupt occurs, whichever occurs first. There is no limit to the number of threads that may simultaneously utilize this service or on the combination of interrupts that may be referenced.

## Prototype

```
U32 hpdi32_irq_wait(void* handle, U32 which, U32 timeout_ms);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
which	This is any bitwise or'd combination of <code>HPDI32_WHICH_IRQ_XXX</code> bits. Set the bits according to the interrupt of interest. Unreferenced interrupts will have no impact. If none are set the function returns immediately with <code>GSC_SUCCESS</code> .
timeout_ms	This is the timeout limit in milliseconds. If an interrupt does not occur within this time period, then the call returns at the end of the period. The timeout period will be at least the amount of time specified, but may be longer depending on the OS.

Return Value	Description
<code>GSC_SUCCESS</code>	Either no interrupts were referenced or one of the referenced interrupts occurred. No indication is given to indicate which interrupt, if any, caused the call to return.
<code>GSC_WAIT_TIMEOUT</code>	The timeout period expired before a referenced interrupt occurred.
Otherwise ...	A <code>GSC_XXX</code> error status reflecting the problem encountered.

## Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_irq_fifo_full_wait(
    void*    handle,
    U32      timeout_ms,
    int       verbose)
{
    U32 status;

    status = hpdi32_irq_wait( handle,
                              HPDI32_WHICH_IRQ_TX_F |
                              HPDI32_WHICH_IRQ_RX_F,
                              timeout_ms);

    if (!verbose)
    {
    }
    else if (status == GSC_SUCCESS)
    {
        printf("Tx/Rx FIFO Full Wait: interrupt occurred.\n");
    }
    else if (status == GSC_WAIT_TIMEOUT)
    {
        printf("Tx/Rx FIFO Full Wait: "
               "timeout after %ld milliseconds\n",
               (long) timeout_ms);
    }
    else
    {
        printf("hpdi32_irq_wait() failure: %ld\n", (long) status);
    }
}
```

```

    }

    return(status);
}

```

### 5.11. hpdi32\_open()

This function is the entry point to open a connection to an HPDI32 board. This function must be called before any other device access functions may be called. If successful, the device and all parameters are initialized to default settings. Multiple requests can be made to access the same device, and each can succeed. However, care must be taken when doing this as device access via one handle is likely to interfere with the device state maintained by the other. Additionally, one handle may configure the device in a way that conflicts with the configuration established by the other.

#### Prototype

```
U32 hpdi32_open(U8 index, void** handle);
```

Argument	Description
index	This is the zero based index of the board to access.
handle	If the request succeeds, the API records at this address the handle to be used for subsequent access to the respective device. This pointer must not be NULL. The pointer returned will be NULL if the request fails and non-NULL otherwise.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

#### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_open(U8 index, void** handle, int verbose)
{
    U32 status;

    status = hpdi32_open(index, handle);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_open() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Device Opened:\n");
        printf("  Index: 0x%lX\n", (long) index);
        printf("  Handle: 0x%lX\n", (long) handle);
    }
}

```

```

    return(status);
}

```

## 5.12. hpdi32\_read()

This function is the entry point to reading received data from an HPDI32. The function should only be called after a successful open of the respective device via `hpdi32_open()`. The operation will be carried out according to the current set of receive side I/O Parameters. If the Overlap Enable option is disabled, then the function will block and return either when the requested amount of data has been read or when the timeout period has lapsed, whichever occurs first. If the Overlap Enable option is enabled, the function will return immediately and the operation will be carried out in the background. In this case the application must either use I/O Completion Notification or query the receive side I/O Status parameters to determine when the operation completes and how much data was read. The service reads up to the requested number of bytes, according to the receive side I/O Data Size parameter (only full data values are retrieved). Only a single read operation can be active at a time. If a request is made while a read operation is in progress, then the new request will fail. If overlapped I/O is requested and the function returns an error status, the overlapped operation may not have been initiated. No matter how an I/O operation ends though, even if it could not be started, an I/O completion event will be triggered if at all possible.

**NOTE:** If the I/O Overlapped parameter is enabled (permitting background read processing), then the I/O buffer handed to `hpdi32_read()` must remain available until the operation completes. Failure to do so will likely result either in stack corruption or a general protection fault.

**NOTE:** An Rx Overrun may occur during a read request when using DMA (either Demand Mode or Non-Demand Mode) with Application Buffers. Such overruns can arise because of the overhead required to prepare the memory for DMA use. To reduce this overhead, reduce the size of the I/O request. To eliminate this overhead, use API Buffers for I/O requests. The likelihood of such Rx Overruns can be reduced by using larger Rx FIFOs. The likelihood of such Rx Overruns can also be reduced by reducing the Rx clock rate.

**NOTE:** For those boards without the Single Cycle Disable feature (see the Board Control Register) Demand Mode DMA based reads may produce an Rx FIFO Under Run. If this does occur, then the failure status `GSC_INVALID_DATA` will be returned, reflecting that the read buffer contains invalid data. On boards with 32-bit PCI interfaces this can occur only when the data size is 8-bit or 16-bit. On boards with 64-bit PCI interfaces this can occur with any data size setting.

**NOTE:** Thoroughly examine the various I/O Parameters to determine the settings required for each application.

**NOTE:** For DMA based I/O using Application Buffers, the buffer must be both readable and writable. This usually means that buffers cannot be declared as `const` or `static const`. I/O requests will fail if the buffer does not have read/write access.

**NOTE:** Applications may make I/O requests of any size. However, the maximum amount of data that can be transfer in a single call is approximately 256MB. This upper limit is based on the macro `GSC_IO_STATUS_COUNT_MASK`.

### Prototype

```
U32 hpdi32_read(void* handle, void* buffer, U32 bytes, U32* count);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
buffer	This is where the retrieved data is stored. It must be large enough to store all of the data requested and it must remain accessible by the API until the operation completes. The

	pointer must not be NULL. The buffer can be an application allocated buffer or either of the API Buffers.
bytes	This is the desired number of bytes to retrieve. The API will limit this to the value specified by the macro GSC_IO_STATUS_COUNT_MASK and will round it down to an integral multiple of the I/O Data Size parameter.
count	The API records the number of bytes actually transferred here. The value recorded may be less than the amount requested due to various factors; request limits, timeout, abort or other errors.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
GSC_WAIT_TIMEOUT	The operation timed out before the requested amount of data was received.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_read(
    void*   handle,
    void*   buffer,
    U32     bytes,
    U32*    count,
    int     verbose)
{
    U32 status;

    status = hpdi32_read(handle, buffer, bytes, count);

    if (!verbose)
    {
    }
    else
    {
        printf("I/O Read Operation:\n");
        printf("  Status:    0x%lX\n", (long) status);
        printf("  Requested: 0x%lX\n", (long) bytes);
        printf("  Received:  0x%lX\n", (long) count[0]);
    }

    return(status);
}
```

### 5.13. hpdi32\_reg\_mod()

This function is the entry point to performing a read-modify-write on a register. The function should only be called after a successful open of the respective device via `hpdi32_open()`. Only the HPDI32 firmware registers (those defined inside `hpdi32_api.h`) may be modified. All PCI and PLX registers are read-only.

#### Prototype

```
U32 hpdi32_reg_mod(void* handle, U32 reg, U32 value, U32 mask);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
reg	This is the register to access. PCI and PLX registers are read-only.
value	This is the desired value to apply. Bits not referenced by the mask are ignored.
mask	This specifies the “value” bits to modify. If a bit is set here, then the corresponding “value” bit will be applied. The remaining “value” bits are ignored.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_reg_bcr_mod(
    void*    handle,
    U32      value,
    U32      mask,
    int      verbose)
{
    U32 status;

    status = hpdi32_reg_mod(handle, HPDI32_BCR, value, mask);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_reg_mod() failure: %ld\n", (long) status);
    }
    else
    {
        printf("BCR Modify:\n");
        printf("  Value: 0x%lX\n", (long) value);
        printf("  Mask:  0x%lX\n", (long) mask);
    }

    return(status);
}
```

## 5.14. hpdi32\_reg\_read()

This function is the entry point to reading the value from an HPDI32 register. The function should only be called after a successful open of the respective device via `hpdi32_open()`. All HPDI32 registers may be read.

### Prototype

```
U32 hpdi32_reg_read(void* handle, U32 reg, U32* value);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
reg	This is the register to access.
value	The value read is recorded here. If this is NULL then no action is taken.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

#### Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_reg_bcr_read(void* handle, U32* value, int verbose)
{
    U32 status;

    status = hpdi32_reg_read(handle, HPDI32_BCR, value);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_reg_read() failure: %ld\n", (long) status);
    }
    else
    {
        printf("BCR Read: 0x%lX\n", (long) value[0]);
    }

    return(status);
}
```

### 5.15. hpdi32\_reg\_write()

This function is the entry point to writing to the register. The function should only be called after a successful open of the respective device via `hpdi32_open()`. Only the HPDI32 firmware registers (those defined inside `hpdi32_api.h`) may be modified. All PCI and PLX registers are read-only.

#### Prototype

```
U32 hpdi32_reg_write(void* handle, U32 reg, U32 value);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
reg	This is the register to access.
value	The value read from the register is recorded here.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

**Example**

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_reg_bcr_write(void* handle, U32 value, int verbose)
{
    U32 status;

    status = hpdi32_reg_write(handle, HPDI32_BCR, value);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_reg_write() failure: %ld\n", (long) status);
    }
    else
    {
        printf("BCR Write: 0x%lX\n", (long) value);
    }

    return(status);
}

```

**5.16. hpdi32\_reset()**

This function is the entry point to perform a device hardware reset. The function should only be called after a successful open of the respective device via `hpdi32_open()`. In doing this, any I/O operations in progress are aborted.

**NOTE:** The API performs a variety of actions during this call that are in addition to the hardware reset. This is necessary for proper API operation. If an application initiates a hardware reset by writing to the Board Control Register the results may be data loss or corruption.

**Prototype**

```
U32 hpdi32_reset(void* handle);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

**Example**

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

```



```

U32 hpdi32_dsl_reset(void* handle, int verbose)
{
    U32 status;

    status = hpdi32_reset(handle);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_reset() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Device Reset: 0x%lX\n", (long) handle);
    }

    return(status);
}

```

### 5.17. hpdi32\_status\_text()

This function is the entry point to retrieving a text based description of the status values supported by the SDK.

#### Prototype

```
U32 hpdi32_status_text(U32 status, char* text, size_t size);
```

Argument	Description
status	This is the status value whose description is desired.
text	The descriptive text is recorded here.
size	This gives the size of the above buffer.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

#### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_status_text(U32 stat, int verbose)
{
    char    buf[128];
    U32     status;

    status = hpdi32_status_text(stat, buf, sizeof(buf));

    if (!verbose)
    {

```

```

    }
    else if (status != GSC_SUCCESS)
    {
        printf("hpdi32_status_text() failure: 0x%lX\n", (long) status);
    }
    else
    {
        printf("Status: 0x%lX: %s\n", (long) stat, buf);
    }

    return(status);
}

```

## 5.18. hpdi32\_version\_get()

This function is the entry point to retrieving version numbers. Without a valid device handle, only the API Library version number is accessible. Access to the Device Driver's version number requires a valid device handle. The following table lists macros associated with this service.

Macro (Services)	Description
HPDI32_VERSION_GET_DRIVER(h,b,s)	This retrieves the driver version string.
HPDI32_VERSION_GET_LIBRARY(h,b,s)	This retrieves the API Library version string.

### Prototype

```

U32 hpdi32_version_get(
    void*    handle,
    U8       id,
    char*     version,
    size_t    size);

```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> . This is ignored except when accessing the Device Driver's version data.
id	This indicates the version number desired. It should be either <code>GSC_VERSION_LIBRARY</code> or <code>GSC_VERSION_DRIVER</code> .
version	This is a buffer where the version string is recorded.
size	This is the size of the above buffer.

Return Value	Description
GSC_SUCCESS	The operation succeeded.
Otherwise ...	A GSC_XXX error status reflecting the problem encountered.

### Example

```

#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_version_get(void* handle, U8 id, int verbose)
{
    U32    status;
    char    ver[32];

```

```

status = hpdi32_version_get(handle, id, ver, sizeof(ver));

if (!verbose)
{
}
else if (status != GSC_SUCCESS)
{
    printf("hpdi32_version_get() failure: %ld\n", (long) status);
}
else
{
    printf("Version: %s\n", ver);
}

return(status);
}

```

## 5.19. hpdi32\_write()

This function is the entry point to writing transmit data to an HPDI32. The function should only be called after a successful open of the respective device via `hpdi32_open()`. The operation will be carried out according to the current set of transmit side I/O Parameters. If the Overlap Enable option is disabled, then the function will block and return either when the requested amount of data has been written or when the timeout period has lapsed, whichever occurs first. If the Overlap Enable option is enabled, the function will return immediately and the operation will be carried out in the background. In this case the application must either use I/O Completion Notification or query the transmit side I/O Status parameters to determine when the operation completes and how much data was read. The service writes up to the requested number of bytes, according to the transmit side I/O Data Size parameter (only full data values are written). Only a single write operation can be active at a time. If a request is made while a write operation is in progress, then the new request will fail. If overlapped I/O is requested and the function returns an error status, the overlapped operation may not have been initiated. No matter how an I/O operation ends though, even if it could not be started, an I/O completion event will be triggered if at all possible.

**NOTE:** If the I/O Overlapped parameter is enabled (permitting background write processing), then the I/O buffer handed to `hpdi32_write()` must remain available until the operation completes. Failure to do so will likely result either in stack corruption or a general protection fault.

**NOTE:** The Tx FIFO may run empty during a write request, resulting in a data transfer pause, when using DMA (either Demand Mode or Non-Demand Mode) with Application Buffers. Such overruns can arise because of the overhead required to prepare the memory for DMA use. To reduce this overhead, reduce the size of the I/O request. To eliminate this overhead, use API Buffers for I/O requests. The likelihood of such Rx Overruns can be reduced by using larger Rx FIFOs. The likelihood of such Rx Overruns can also be reduced by reducing the Rx clock rate.

**NOTE:** For those boards without the Single Cycle Disable feature (see the Board Control Register) Demand Mode DMA based writes may produce a Tx FIFO Overrun. If this does occur, then the failure status `GSC_INVALID_DATA` will be returned, reflecting that the Tx FIFO image does not reflect the data written to it. On boards with 32-bit PCI interfaces this can occur only when the data size is 8-bit or 16-bit. On boards with 64-bit PCI interfaces this can occur with any data size setting.

**NOTE:** Thoroughly examine the various I/O Parameters to determine the settings required for each application.

**NOTE:** For DMA based I/O using Application Buffers, the buffer must be both readable and writable. This usually means that buffers cannot be declared as `const` or `static const`. I/O requests will fail if the buffer does not have read/write access.

**NOTE:** Applications may make I/O requests of any size. However, the maximum amount of data that can be transfer in a single call is approximately 256MB. This upper limit is based on the macro `GSC_IO_STATUS_COUNT_MASK`.

## Prototype

```
U32 hpdi32_write(
    void*      handle,
    const void* buffer,
    U32        bytes,
    U32*       count);
```

Argument	Description
handle	This is an API device handle obtained via <code>hpdi32_open()</code> .
buffer	This is the source for the data to send. It must remain accessible by the API until the operation completes. The pointer must not be NULL. The buffer can be an application allocated buffer or either of the API Buffers.
bytes	This is the desired number of bytes to write. The API will limit this to the value specified by the macro <code>GSC_IO_STATUS_COUNT_MASK</code> and will round it down to an integral multiple of the I/O Data Size parameter.
count	The API records the number of bytes actually transferred here. The value recorded may be less than the amount requested due to various factors; request limits, timeout, abort or other errors.

Return Value	Description
<code>GSC_SUCCESS</code>	The operation succeeded.
<code>GSC_WAIT_TIMEOUT</code>	The operation timed out before the requested amount of data was written.
Otherwise ...	A <code>GSC_XXX</code> error status reflecting the problem encountered.

## Example

```
#include <stdio.h>

#include "hpdi32_api.h"
#include "hpdi32_dsl.h"

U32 hpdi32_dsl_write(
    void*      handle,
    const void* buffer,
    U32        bytes,
    U32*       count,
    int        verbose)
{
    U32 status;

    status = hpdi32_write(handle, buffer, bytes, count);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
```

```
{
    printf("hpdi32_write() failure: %ld\n", (long) status);
}
else
{
    printf("I/O Write Operation:\n");
    printf("  Status:      0x%lX\n", (long) status);
    printf("  Requested: 0x%lX\n", (long) bytes);
    printf("  Sent:       0x%lX\n", (long) count[0]);
}

return(status);
}
```

## 6. Configuration Parameters

The HPDI32 and the API Library include a number of configurable features. These features are referred to by the API as Configuration Parameters. This section describes all of the HPDI32 Configuration Parameters.

### 6.1. Parameter Macros

The Configuration Parameters are grouped according to their functional categories. Within each category each parameter is described (in this section) along with the set of utility macros designed to facilitate configuration of and access to the respective parameters. Parameter macros fall into three groups, which are described in the following paragraphs. All macros are described in the following pages in association with their respective parameters. The parameter categories are as given in the below table.

Parameter Categories	Description
HPDI32_CABLE_XXX	These refer to the Cable Parameters. These pertain to configuration of the cable signals.
HPDI32_FIFO_XXX	These refer to the FIFO Parameters.
HPDI32_IO_XXX	These refer to the Input/Output Parameters. These pertain to data transfer between the host and the HPDI32.
HPDI32_IRQ_XXX	These refer to the Interrupt Parameters.
HPDI32_MISC_XXX	These refer to the Miscellaneous Parameters.
HPDI32_RX_XXX	These refer to the Receiver Parameters.
HPDI32_TX_XXX	These refer to the Transmitter Parameters.

#### 6.1.1. Parameter Definitions

The first group of macros includes the parameter definitions. These are used to identify the specific parameter to be accessed. These macros begin with “HPDI32\_” and are followed immediately by upper case letters identifying the parameter category. For example “HPDI32\_MISC\_” prefaces all Miscellaneous Parameter identifiers. These macros end with upper case letters indicating the name of the specific parameter. For example “HPDI32\_MISC\_STRICT\_ARGUMENTS” identifies the Miscellaneous Strict Arguments parameter.

#### 6.1.2. Value Definitions

The second group of macros identifies predefined values associated with the respective parameters. These macros begin with the Parameter Definition and are followed by a single underscore (“\_”) then upper case letters that reflect the meaning of the respective values. For example the macro “HPDI32\_MISC\_STRICT\_ARGUMENTS\_DISABLE” is the value that represents the parameter’s *disabled* setting.

#### 6.1.3. Service Definitions

The third group of macros performs operations on parameters. These are utility macros that retrieve parameter settings and states or assign parameter values. These macros include the Parameter Definition followed by a double underscore (“\_\_”) then upper case letters that reflect the action to perform. For example “HPDI32\_MISC\_STRICT\_ARGUMENTS\_\_GET( )” retrieves the current setting of the Miscellaneous Strict Arguments parameter. These macros include arguments, which are described as follows.

##### 6.1.3.1. Device Handle: h

In the service macros, the argument *h* refers to the device handle used to access the respective device. This handle is obtained by calling `hpdi32_open( )`. This argument must not be NULL.

### 6.1.3.2. Which Bits: w

In the service macros, the argument *w* refers to any combination of the HPDI32\_WHICH\_XXX bits. Refer to paragraph 3.6 on page 38. With some parameters this argument is unused or is specified inside the macro's replacement text. In those cases the *w* is not included as a macro argument.

### 6.1.3.3. Set Value: s

In the service macros, the argument *s* refers to the value to be applied to the referenced parameter. With some parameters the value can be arbitrarily assigned by the application. With most parameters this argument should be one of the predefined value definitions. The *s* is not included as a macro argument for those cases where either a value is not being applied or the value applied is specified inside the macro replacement text.

### 6.1.3.4. Get Value: g

In the service macros, the argument *g* refers to the address of the variable to receive the parameter's current setting. In cases where the current setting is not being read, this argument has been omitted from the service macro. In all cases, this argument can be NULL, in which case the current value is not retrieved.

## 6.2. Cable Parameters

The purpose of the Cable Parameters is to permit access to and control of the signals available at the HPDI32's external interface connector. All Cable Parameters are put in a default state when the device is opened and are returned to that state via the `hpdi32_init()` and `hpdi32_reset()` services. The configuration of the cable signals is controlled by HPDI32 firmware based registers. Applications are free to manipulate the configuration either via the API's register access services or the Cable Parameter access services. When accessing the Cable Parameters any number or combination of appropriate HPDI32\_WHICH\_XXX identifiers may be used, even none. The following table summarizes the Cable Parameters.

Parameter Macros	Description
HPDI32_CABLE_CLOCK_STATE	This refers to the state of the cable's clock signal.
HPDI32_CABLE_COMMAND_MODE	This refers to the operating mode for various cable control signals.
HPDI32_CABLE_COMMAND_STATE	This refers to the state of the various cable control signals.

**NOTE:** When a Flow Control signal's mode is set to GPIO, then it defaults to a GPIO input. If the mode is retrieved without also being set, then the mode is reported as GPIO, but might be configured as an output.

### 6.2.1. Cable Parameter: Clock State

The purpose of this read-only parameter is to report the state of the Cable Clock signal. The state is considered *active* if the signal is driven by the board itself or is expected to be driven by a remote device. The state is considered *inactive* otherwise. If the transmitter is enabled then the board drives the signal and its state is reported as *active*. If the receiver is enabled the state is reported as *active* since it should be driven by the remote device. The state is otherwise reported as *inactive*. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_CABLE_CLOCK_STATE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_CABLE_CLOCK_STATE_ACTIVE	This value refers to the signal's <i>active</i> state.
HPDI32_CABLE_CLOCK_STATE_INACTIVE	This value refers to the signal's <i>inactive</i> state.

Macro (Services)	Description
HPDI32_CABLE_CLOCK_STATE_GET(h, g)	This retrieves the signal's current state.

### 6.2.2. Cable Parameter: Command Mode

The purpose of this parameter is to control and report the Cable Command Mode for those Cable Command signals which are configurable. In this respect, the signals operate either in a Flow Control mode to control data flow over the cable interface or as General Purpose I/O. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_CABLE_COMMAND_MODE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_CABLE_COMMAND_MODE_DEFAULT	This refers to the signal's default mode. This is Flow Control, which is the hardware's default.
HPDI32_CABLE_COMMAND_MODE_FLOW_CONTROL	This refers to the data Flow Control mode.
HPDI32_CABLE_COMMAND_MODE_GPIO_IN	This refers to the GPIO Input mode.
HPDI32_CABLE_COMMAND_MODE_GPIO_OUT_HI	This refers to the GPIO Output High mode.
HPDI32_CABLE_COMMAND_MODE_GPIO_OUT_LOW	This refers to the GPIO Output Low mode.

Macro (Services)	Description
HPDI32_CABLE_COMMAND_MODE_FC(h, w)	This sets signals to Flow Control mode.
HPDI32_CABLE_COMMAND_MODE_GET(h, w, g)	This retrieves a signal's current mode.
HPDI32_CABLE_COMMAND_MODE_GPIO_HI(h, w)	This sets signals to GPIO Output High mode.
HPDI32_CABLE_COMMAND_MODE_GPIO_IN(h, w)	This sets signals to GPIO Input mode.
HPDI32_CABLE_COMMAND_MODE_GPIO_LOW(h, w)	This sets signals to GPIO Output Low mode.
HPDI32_CABLE_COMMAND_MODE_RESET(h, w)	This resets the current mode to the default.
HPDI32_CABLE_COMMAND_MODE_SET(h, w, s)	This sets the current mode.
HPDI32_CABLE_COMMAND_MODE_XXX_FC(h)	This sets signal XXX to Flow Control mode. *
HPDI32_CABLE_COMMAND_MODE_XXX_GET(h, g)	This retrieves the current mode for signal XXX. *
HPDI32_CABLE_COMMAND_MODE_XXX_IN(h)	This sets signal XXX to GPIO Input mode. *
HPDI32_CABLE_COMMAND_MODE_XXX_HI(h)	This sets signal XXX to GPIO Output High mode. *
HPDI32_CABLE_COMMAND_MODE_XXX_LOW(h)	This sets signal XXX to GPIO Output Low mode. *
HPDI32_CABLE_COMMAND_MODE_XXX_RESET(h)	This resets the current mode for signal XXX to the default. *
HPDI32_CABLE_COMMAND_MODE_XXX_SET(h, s)	This sets the current mode for signal XXX. *

\* The XXX sequence refers to the following individual options: 0, 1, 2, 3, 4, 5 and 6 for Cable Command signals zero to six, GPIO\_0, GPIO\_1, GPIO\_2, GPIO\_3, GPIO\_4, GPIO\_5 and GPIO\_6 for the Cable Command signals configured as GPIO lines zero to six, and for this Cable Command signals configured as Flow Control it includes FV for Frame Valid, LV for Line Valid, SV for Status Valid, RR for Receive Ready, TR for Transmit Ready, RE for Receive Enable, and TE for Transmit Enable.

### 6.2.3. Cable Parameter: Command State

The purpose of this read-only parameter is to report the state of the Cable Command signals. When the signal is in its Flow Control mode, the state is reported as active when the signal is driven, or is expected to be driven. The state is reported as inactive otherwise. When in the signal's GPIO mode, the state is reported as active when the signal is read as high, and is reported as inactive when read as low.

Macro (Parameter)	Description
HPDI32_CABLE_COMMAND_STATE	This is the identifier for this parameter.



Macro (Values)	Description
HPDI32_CABLE_COMMAND_STATE_ACTIVE	This refers to the signal's active or high state.
HPDI32_CABLE_COMMAND_STATE_INACTIVE	This refers to the signal's inactive or low state.

Macro (Services)	Description
HPDI32_CABLE_COMMAND_STATE_GET(h, w, g)	This retrieves a signal's current state.
HPDI32_CABLE_COMMAND_STATE_XXX_GET(h, g)	This retrieves the current state for signal XXX. *

\* The XXX sequence refers to the following individual options: 0, 1, 2, 3, 4, 5 and 6 for Cable Command signals zero to six, GPIO\_0, GPIO\_1, GPIO\_2, GPIO\_3, GPIO\_4, GPIO\_5 and GPIO\_6 for the Cable Command signals configured as GPIO lines zero to six, and for this Cable Command signals configured as Flow Control it includes FV for Frame Valid, LV for Line Valid, SV for Status Valid, RR for Receive Ready, TR for Transmit Ready, RE for Receive Enable, and TE for Transmit Enable.

### 6.3. FIFO Parameters

The purpose of the FIFO Parameters is to permit access to and control of the transmit and receive FIFOs. All FIFO Parameters are put in a default state when the device is opened and are returned to that state via the `hpdi32_init()` and `hpdi32_reset()` services. The configuration of the FIFOs is partly controlled by HPDI32 firmware based registers. Applications are free to manipulate the configuration either via the API's register access services or the FIFO Parameter access services. When using the service `hpdi32_config()`, any number or combination of `HPDI32_WHICH_TX` or `HPDI32_WHICH_RX` may be used, even none. The transmit FIFO will only be accessed if the transmit bit is set and the receive FIFO will only be accessed if the receive bit is set. If neither is set, then no action will be taken. The following table lists the FIFO Parameters.

Parameter Macros	Description
HPDI32_FIFO_ALMOST_LEVEL	This refers to the FIFO Almost Full and Almost Empty status levels.
HPDI32_FIFO_RESET	This refers to resetting the FIFOs.
HPDI32_FIFO_SIZE	This refers to the size of the FIFOs.
HPDI32_FIFO_STATUS	This refers to the FIFO fill level status.
HPDI32_FIFO_TRANSFER_SIZE	This refers to the amount of guaranteed space/data available in the FIFOs.

#### 6.3.1. FIFO Parameter: Almost Level

The purpose of this parameter is to control and report the FIFO Almost Full and Almost Empty status levels. When using the service `hpdi32_config()`, any number or combination of `HPDI32_WHICH_AF` or `HPDI32_WHICH_AE` may be used, even none (in addition to the transmit and receive bits described above). The Almost Full level will only be accessed if the Almost Full bit is set and the Almost Empty level will only be accessed if the Almost Empty bit is set. If neither is set, then no action will be taken. Which ever bits are set, they will be applied to transmit and receive FIFOs, respectively. The following tables describe the macros associated with this parameter.

**NOTE:** The Almost Empty status becomes active when the FIFO contains *ALMOST EMPTY* or fewer samples.

**NOTE:** The Almost Full status becomes active when the FIFO can receive *ALMOST FULL* or fewer additional samples before being full.

**NOTE:** The API automatically resets the referenced FIFOs when either Almost Level is set. This insures that the setting takes affect immediately. A side affect however is that any data in the FIFO is lost.

**NOTE:** Applications should not apply settings to any FIFO Almost Level while an I/O operation is in progress using the respective FIFO. Doing so will result in the loss of any data in the FIFO and will interfere with proper transfer of data through the board.

**NOTE:** The API will automatically limit the FIFO Almost Level parameter values to the size of the respective FIFO, when the FIFO size is known.

Macro (Parameter)	Description
HPDI32_FIFO_ALMOST_LEVEL	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_FIFO_ALMOST_EMPTY_DEFAULT	This is the default Almost Empty level, which may differ from the hardware's default.
HPDI32_FIFO_ALMOST_FULL_DEFAULT	This is the default Almost Full level, which may differ from the hardware's default.
HPDI32_FIFO_ALMOST_LEVEL_MAX	This is the maximum level permissible.

Macro (Services)	Description
HPDI32_FIFO_ALMOST_LEVEL_GET(h, w, g)	This retrieves a parameter's current setting.
HPDI32_FIFO_ALMOST_LEVEL_SET(h, w, s)	This sets a parameter's level.
HPDI32_FIFO_ALMOST_LEVEL_XXX_GET(h, g)	This retrieves the respective FIFO Almost setting. *
HPDI32_FIFO_ALMOST_LEVEL_XXX_SET(h, s)	This sets the respective FIFO Almost setting. *

\* The XXX sequence refers to the following individual options: RX\_AE for the Rx FIFO Almost Empty level, RX\_AF for the Rx FIFO Almost Full level, TX\_AE for the Tx FIFO Almost Empty level and TX\_AF for the Tx FIFO Almost Full level.

### 6.3.2. FIFO Parameter: Reset

The purpose of this parameter is to control the resetting of the respective FIFOs. The following tables describe the macros associated with this parameter.

**NOTE:** Applications should not reset a FIFO while in use by an I/O operation. Doing so will result in the loss of any data in the FIFO and will interfere with proper transfer of data through the board.

Macro (Parameter)	Description
HPDI32_FIFO_RESET	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_FIFO_RESET_DEFAULT	This is the default action, which to do nothing.
HPDI32_FIFO_RESET_NO	This means the FIFO is not to be reset or that it was not reset.
HPDI32_FIFO_RESET_YES	This means the FIFO is to be reset or that it was reset.

Macro (Services)	Description
HPDI32_FIFO_RESET_RESET(h, w)	This resets the respective FIFOs.
HPDI32_FIFO_RESET_SET(h, w, s)	This applies a FIFO reset option.
HPDI32_FIFO_RESET_XXX_RESET(h, s)	This resets the respective FIFO. *
HPDI32_FIFO_RESET_XXX_SET(h, s)	This applies a setting to the respective FIFO. *
HPDI32_FIFO_RESET_XXX_YES(h)	This resets the respective FIFO. *
HPDI32_FIFO_RESET_YES(h, w)	This resets the specified FIFO(s).

\* The XXX sequence refers to the following individual options: RX for the Rx FIFO and TX for the Tx FIFO.

### 6.3.3. FIFO Parameter: Size

The purpose of this read-only parameter is to report the size of the respective FIFOs. The following tables describe the macros associated with this parameter. If the HPDI32 does not support the FIFO Size Registers then the size is reported as zero (0).

Macro (Parameter)	Description
HPDI32_FIFO_SIZE	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_FIFO_SIZE_GET(h, w, g)	This retrieves a FIFO size.
HPDI32_FIFO_SIZE_XXX_GET(h, g)	This retrieves the size of the respective FIFO. *

\* The XXX sequence refers to the following individual options: RX for the Rx FIFO and TX for the Tx FIFO.

### 6.3.4. FIFO Parameter: Status

The purpose of this read-only parameter is to report the fill level status of the respective FIFOs. The following tables describe the macros associated with this parameter. If the FIFO Almost Levels are set to illogical values (overlapping or larger than the FIFO size) then the status returned may be incorrect.

Macro (Parameter)	Description
HPDI32_FIFO_STATUS	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_FIFO_STATUS_ALMOST_EMPTY	The FIFO contains Almost Empty or fewer data values.
HPDI32_FIFO_STATUS_ALMOST_FULL	The FIFO contains Almost Full or fewer data spaces.
HPDI32_FIFO_STATUS_EMPTY	The FIFO is empty.
HPDI32_FIFO_STATUS_FULL	The FIFO is full.
HPDI32_FIFO_STATUS_MEDIAN	The FIFO is between Almost Empty and Almost Full.

Macro (Services)	Description
HPDI32_FIFO_STATUS_GET(h, w, g)	This retrieves a FIFO fill status.
HPDI32_FIFO_STATUS_XXX_GET(h, g)	This retrieves the fill status of the respective FIFO. *

\* The XXX sequence refers to the following individual options: RX for the Rx FIFO and TX for the Tx FIFO.

### 6.3.5. FIFO Parameter: Transfer Size

The purpose of this read-only parameter is to report the number of samples the API guarantees can be transferred to or from the respective FIFO by an I/O request (i.e. a read or write request). The number returned is not an exact number and may be much less than the exact number. Essentially, it is simply the number the API is able to discern by examining the board's features and state and is the number the API guarantees can be transferred to or from the respective FIFO at that moment. The following tables describe the macros associated with this parameter. If the FIFO Almost Levels are set to illogical values (overlapping or larger than the FIFO size) then the number returned may be invalid.

Macro (Parameter)	Description
HPDI32_FIFO_TRANSFER_SIZE	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_FIFO_TRANSFER_SIZE_GET(h, w, g)	This retrieves a FIFO Transfer Size value.
HPDI32_FIFO_TRANSFER_SIZE_XXX_GET(h, g)	This retrieves the Transfer Size value for the respective FIFO. *

\* The XXX sequence refers to the following individual options: RX for the Rx FIFO and TX for the Tx FIFO.

## 6.4. I/O Parameters

The purpose of the I/O Parameters is to permit access to and control of transmit and receive I/O operations. All I/O Parameters are put in a default state when the device is opened and are returned to that state via the `hpdi32_init()` service. The configuration of the I/O Parameters is retained entirely within the API and cannot be altered by any HPDI32 registers. When using the service `hpdi32_config()`, any number or combination of `HPDI32_WHICH_TX` or `HPDI32_WHICH_RX` may be used, even none. The transmit I/O Parameters will be accessed only if the transmit bit is set and the receive I/O Parameters will be accessed only if the receive bit is set. If neither is set, then no action will be taken.

The API can perform I/O transfer operations using either of two types of buffers. First, it can use Application Buffers which are allocated and maintained entirely by the application. These are obtained by `malloc()` type services and, through the processor's memory manager, appear to the application to be contiguous memory. These buffers are, in-fact, scattered throughout physical memory and at times are paged out to the hard disk. The second buffer type, API Buffers, is memory that is physically contiguous and that is also locked in place in memory until released via the API. Application Buffers have the advantage that they can be significantly larger than API Buffers. API Buffers have the advantage that they require less overhead during I/O operations, potentially producing higher throughput rates. For I/O requests, applications can choose at will what type of buffers to use and, if using API Buffers, which API Buffer to use (Tx or Rx). There is a slight performance penalty however, when switching between Application Buffers and API Buffers, and vice-versa. If an application can acquire API Buffers of suitable size, then the best results will generally be achieved by using those exclusively. Additionally, if an application is interested primarily in transmitting data or in receiving data, then the best performance can generally be gained by using the two API Buffers in a ping-pong sequence; one buffer being used for I/O while the other is being processed, then switching over as soon as processing is done.

The API supports blocking and overlapped I/O requests. The default is blocking I/O where the call returns at the conclusion of the operation. Overlapped I/O is selected merely by enabling the I/O Overlap Enable parameter. When this is done I/O requests return immediately, while the operation is carried out in the background. For both methods, there are two ways of determining when and how an operation concludes. The first method is by polling. By using the I/O Status parameter an application can query for the status of an I/O operation. This indicates if the operation is still in progress, if it has ended, and how (timeout, abort, error) and how much data was transferred. This can be done by any thread both for overlapped I/O and blocking I/O (i.e. one thread can check if another is still blocked on an I/O operation). The second method is by event notification, which is available both as a callback and as a wait service. Using the callback service an application can provide a function pointer and an arbitrary argument that is invoked when the I/O completes (Tx and Rx are independently configurable). The callback occurs in a separate thread context and must return before any follow-on callbacks can be made. (The callback receives the device handle, an application's arbitrary value, and the I/O status as arguments.) Using the wait service, any number of threads can block until an I/O operation ends. Each thread can independently wait on Tx and/or Rx. When a wait request is made the thread will block until the first of the referenced operations ends. This occurs whether the I/O operation began before the request was made or began afterwards. Once resumed the I/O Status parameter must be queried to determine the I/O completion status.

The following table lists the I/O Parameters.

Parameter Macros	Description
<code>HPDI32_IO_ABORT</code>	This refers to aborting an I/O request.
<code>HPDI32_IO_ABORTED</code>	This refers to the abort status of an I/O request.
<code>HPDI32_IO_BUFFER_POINTER</code>	This refers to the pointer to an API Buffer.
<code>HPDI32_IO_BUFFER_SIZE</code>	This refers the size of an API Buffer.
<code>HPDI32_IO_CALLBACK_ARG</code>	This refers to an arbitrary, application supplied callback argument value.
<code>HPDI32_IO_CALLBACK_FUNC</code>	This refers to an application supplied I/O completion callback function.
<code>HPDI32_IO_DATA_SIZE</code>	This refers to width of the cable data: 8, 16 or 32-bits.

HPDI32_IO_DMA_CHANNEL_SEL	This refers to when DMA channels are acquired and released.
HPDI32_IO_DMA_CONTROL_MODE	This refers to how non-Demand Mode DMA is handled by the API.
HPDI32_IO_DMA_PRIORITY	This refers to DMA priority for simultaneous reads and writes.
HPDI32_IO_MODE	This refers to the data transfer mode: PIO, DMA, DMDMA.
HPDI32_IO_OVERLAP_ENABLE	This refers to how I/O requests are processed: foreground, background.
HPDI32_IO_PIO_THRESHOLD	This refers to the threshold at which I/O requests are automatically performed using PIO mode data transfers.
HPDI32_IO_SINGLE_CYCLE	This refers to a device's feature support at critical FIFO fill levels.
HPDI32_IO_STATUS	This refers to the current status of an I/O request.
HPDI32_IO_TIMEOUT	This refers to the overall time limit allowed for I/O requests.

#### 6.4.1. I/O Parameter: Abort

The purpose of this parameter is to abort an ongoing I/O operation. This parameter is applicable to active I/O requests only. No action occurs if none are active at the time an abort request is made. There is also no affect on future I/O requests. In the `hpd32_config()` service the Transmit and Receive selections are made using individual HPDI32\_WHICH\_XX bits. (Here “XX” is “TX”, “RX” or any of the predefined combinations.) The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_IO_ABORT	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_ABORT_DEFAULT	This is the default action to take, which is to do nothing.
HPDI32_IO_ABORT_NO	As a “set” option this means do not perform an abort. As a “get” option it means that an abort did not occur.
HPDI32_IO_ABORT_YES	As a “set” option this requests an abort. This value is never returned as a “get” option as the parameter auto clears after an abort request.

Macro (Services)	Description
HPDI32_IO_ABORT__SET(h, w, s)	This applies an option to an I/O operation.
HPDI32_IO_ABORT__XXX_SET(h, s)	This applies an option to an I/O operation. *
HPDI32_IO_ABORT__XXX_YES(h)	This aborts an I/O operation. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.2. I/O Parameter: Aborted

The purpose of this read-only parameter is to determine if an I/O abort has occurred. This is applied against the I/O status that exists at the time, and will reference the status from the last operation concluded, if applicable, or an ongoing operation, if one is active. Only the most recent or current status is available. The status of previous operations is not available. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_IO_ABORTED	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_ABORTED_NO	This means that an I/O operation was not aborted.
HPDI32_IO_ABORTED_YES	This means that an I/O operation was aborted.

Macro (Services)	Description
HPDI32_IO_ABORTED__GET(h, w, g)	This retrieves the status of an operation.
HPDI32_IO_ABORTED__XXX_GET(h, g)	This retrieves the status of an I/O operation. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

### 6.4.3. I/O Parameter: Buffer Pointer

The purpose of this read-only parameter is to retrieve the pointer to a respective API Buffer. If the application has not configured the size of the respective buffer, then the pointer returned will be NULL. The following tables describe the macros associated with this parameter.

**NOTE:** Applications must obtain a fresh pointer each time a change is made to the API Buffer size. Use of a stale pointer may generate a memory protection fault. Refer to the I/O Buffer Size parameter in the next section.

**NOTE:** For DMA based I/O using Application Buffers, the buffer must be both readable and writable. This usually means that buffers cannot be declared as `const` or `static const`. I/O requests will fail if the buffer does not have read/write access.

Macro (Parameter)	Description
HPDI32_IO_BUFFER_POINTER	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_IO_BUFFER_POINTER_GET(h, w, g)	This retrieves an API Buffer pointer.
HPDI32_IO_BUFFER_POINTER_XXX_GET(h, g)	This retrieves an API Buffer pointer. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

### 6.4.4. I/O Parameter: Buffer Size

The purpose of this parameter is to adjust and retrieve the size of the respective API Buffer. The following tables describe the macros associated with this parameter.

**NOTE:** The Buffer Size cannot be changed while the buffer is in use by an I/O operation.

**NOTE:** The API has no control over the amount of memory the OS will grant in response to an API Buffer allocation request. Each of the API Buffers is a contiguous block of memory requested of the OS by the Device Driver. The OS manages these types of resources differently than application memory resources so the size of the API Buffer obtained may be significantly less than requested by the application. Applications must therefore examine this parameter after it is adjusted to guard against memory protection faults.

**NOTE:** A request to increase the API Buffer Size may take several seconds to complete. This is due entirely to OS and is not controllable by the API or the driver.

**NOTE:** Each time the application requests an API Buffer size change, the pointer used to access the buffer is likely to also change. Applications must therefore obtain a fresh pointer following a size change request. Refer to the I/O Buffer Pointer parameter description in the previous section.

Macro (Parameter)	Description
HPDI32_IO_BUFFER_SIZE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_BUFFER_SIZE_DEFAULT	This is the default size, which is zero.

Macro (Services)	Description
HPDI32_IO_BUFFER_SIZE_GET(h, w, g)	This retrieves a current size setting.

HPDI32_IO_BUFFER_SIZE__SET(h,w,s,g)	This requests a size change and retrieves the results.
HPDI32_IO_BUFFER_SIZE__XXX_FREE(h)	This requests that a buffer be freed.
HPDI32_IO_BUFFER_SIZE__XXX_GET(h,g)	This retrieves a buffer's current size.
HPDI32_IO_BUFFER_SIZE__XXX_SET(h,s,g)	This requests a size change and retrieves the results.

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.5. I/O Parameter: Callback Argument

The purpose of this parameter is to modify and report the application provided argument that it receives as “arg2” for an I/O completion callback event. The following tables describe the macros associated with this parameter.

**NOTE:** Applications must remember that the macros GSC\_NO\_CHANGE and GSC\_DEFAULT have special meaning when applying parameter modifications. If the application specific value being supplied for this parameter happens to equal either of these values, then the results will be according to the API's use of these special values rather than the applications intent.

**NOTE:** This parameter can be accessed and altered during the callback, but the callback must return before subsequent callbacks can be made on the same I/O transfer direction.

Macro (Parameter)	Description
HPDI32_IO_CALLBACK_ARG	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_CALLBACK_ARG_DEFAULT	This is the default, which is zero.

Macro (Services)	Description
HPDI32_IO_CALLBACK_ARG__GET(h,w,g)	This retrieves a current setting.
HPDI32_IO_CALLBACK_ARG__RESET(h,w)	This resets a setting.
HPDI32_IO_CALLBACK_ARG__SET(h,w,s)	This requests a setting change.
HPDI32_IO_CALLBACK_ARG__XXX_GET(h,g)	This retrieves a current setting. *
HPDI32_IO_CALLBACK_ARG__XXX_RESET(h)	This resets a setting. *
HPDI32_IO_CALLBACK_ARG__XXX_SET(h,s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.6. I/O Parameter: Callback Function

The purpose of this parameter is to modify and report the application provided function pointer for I/O completion events. The following tables describe the macros associated with this parameter.

**NOTE:** This parameter can be accessed and altered during the callback, but the callback must return before subsequent callbacks can be made on the same I/O transfer direction.

Macro (Parameter)	Description
HPDI32_IO_CALLBACK_FUNC	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_CALLBACK_FUNC_DEFAULT	This is the default, which is NULL.

Macro (Services)	Description
HPDI32_IO_CALLBACK_FUNC__GET(h,w,g)	This retrieves a current function pointer.
HPDI32_IO_CALLBACK_FUNC__RESET(h,w)	This resets a function pointer.
HPDI32_IO_CALLBACK_FUNC__SET(h,w,s)	This requests a function pointer change.

HPDI32_IO_CALLBACK_FUNC__XXX_GET(h,g)	This retrieves a current function pointer. *
HPDI32_IO_CALLBACK_FUNC__XXX_RESET(h)	This resets a function pointer. *
HPDI32_IO_CALLBACK_FUNC__XXX_SET(h,s)	This requests a function pointer change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.7. I/O Parameter: Data Size

The purpose of this parameter is to modify and report the base Data Size for data transfers on the external cable interface. This parameter specifies the size of each sample transferred across the cable in bytes. The following tables describe the macros associated with this parameter.

**NOTE:** Whatever Data Size is used, it is always aligned against the lowest cable data byte.

Macro (Parameter)	Description
HPDI32_IO_DATA_SIZE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_DATA_SIZE_8_BITS	This sets the data size to 8-bits.
HPDI32_IO_DATA_SIZE_16_BITS	This sets the data size to 16-bits.
HPDI32_IO_DATA_SIZE_32_BITS	This sets the data size to 32-bits.
HPDI32_IO_DATA_SIZE_DEFAULT	This is the default, which is 32-bits.

Macro (Services)	Description
HPDI32_IO_DATA_SIZE__GET(h,w,g)	This retrieves a current setting.
HPDI32_IO_DATA_SIZE__RESET(h,w)	This resets a setting.
HPDI32_IO_DATA_SIZE__SET(h,w,s)	This requests a setting change.
HPDI32_IO_DATA_SIZE__XXX_8(h)	This requests a setting of 8-bits. *
HPDI32_IO_DATA_SIZE__XXX_16(h)	This requests a setting of 16-bits. *
HPDI32_IO_DATA_SIZE__XXX_32(h)	This requests a setting of 32-bits. *
HPDI32_IO_DATA_SIZE__XXX_GET(h,g)	This retrieves a current setting. *
HPDI32_IO_DATA_SIZE__XXX_RESET(h)	This resets a setting. *
HPDI32_IO_DATA_SIZE__XXX_SET(h,s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.8. I/O Parameter: DMA Channel Select

The purpose of this parameter is to modify and report the API's processing of DMA channel selection when I/O requests are made and completed. This parameter is applicable only when applications opt to DMA for I/O requests. The following tables describe the macros associated with this parameter.

**NOTE:** The PCI interface chip used on all HPDI32s includes two DMA channels/controllers. If the HPDI32 supports the Feature Set Register and the DMA Channel 1 bit is set (the HPDI32\_FSR\_DMA\_CH1 bit) then the HPDI32 supports DMA on both channels. Otherwise the firmware supports DMA on only the first DMA channel, meaning that the board cannot support simultaneous DMA reads and writes.

Macro (Parameter)	Description
HPDI32_IO_DMA_CHANNEL_SEL	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_DMA_CHANNEL_SEL_DYNAMIC	This selects the <i>dynamic</i> option. With this setting the API acquires a DMA channel (a hardware resource)



	when needed and releases it when not needed, which is as soon as the I/O request completes. This is applicable when the HPDI32 firmware supports DMA on only a single channel and the application transfers data in both directions.
HPDI32_IO_DMA_CHANNEL_SEL_RX_DEFAULT	This is the Rx default, which is <i>dynamic</i> .
HPDI32_IO_DMA_CHANNEL_SEL_STATIC	This selects the <i>static</i> option. With this setting the API acquires a DMA channel (a hardware resource) when needed and keeps it until told to release it (implicitly). This is applicable when the HPDI32 firmware supports DMA on both DMA channels or when the application transfers data in just a single direction. This option is more efficient.
HPDI32_IO_DMA_CHANNEL_SEL_TX_DEFAULT	This is the Tx default, which is <i>static</i> .

Macro (Services)	Description
HPDI32_IO_DMA_CHANNEL_SEL_GET(h, w, g)	This retrieves a current setting.
HPDI32_IO_DMA_CHANNEL_SEL_SET(h, w, s)	This requests a setting change.
HPDI32_IO_DMA_CHANNEL_SEL_XXX_DYNAMIC(h)	This requests a setting of <i>dynamic</i> . *
HPDI32_IO_DMA_CHANNEL_SEL_XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IO_DMA_CHANNEL_SEL_XXX_RESET(h)	This resets a setting. *
HPDI32_IO_DMA_CHANNEL_SEL_XXX_SET(h, s)	This requests a setting change. *
HPDI32_IO_DMA_CHANNEL_SEL_XXX_STATIC(h)	This requests a setting of <i>static</i> . *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.9. I/O Parameter: DMA Control Mode

The purpose of this parameter is to modify and report the API's handling of I/O requests using Non-Demand Mode DMA. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_IO_DMA_CONTROL_MODE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_DMA_CONTROL_MODE_AUTOMATIC	This selects the <i>automatic</i> option. With this setting the API maintains data integrity automatically on behalf of the application. Reads will not return indeterminate data and writes will not loose data. This is done at the expense of performance.
HPDI32_IO_DMA_CONTROL_MODE_DEFAULT	This is the default, which is <i>automatic</i> .
HPDI32_IO_DMA_CONTROL_MODE_MANUAL	This selects the <i>manual</i> option. With this setting, applications are responsible for insuring data integrity by verifying manually before a transfer that the FIFO can accommodate the request. If not, reads may return indeterminate data and writes may loose data. This is because Non-Demand Mode DMA is a blind data transfer mode.

Macro (Services)	Description
HPDI32_IO_DMA_CONTROL_MODE_GET(h, w, g)	This retrieves a current setting.
HPDI32_IO_DMA_CONTROL_MODE_RESET(h, w)	This resets a setting.
HPDI32_IO_DMA_CONTROL_MODE_SET(h, w, s)	This requests a setting change.
HPDI32_IO_DMA_CONTROL_MODE_XXX_AUTO(h)	This requests a setting of <i>automatic</i> . *

HPDI32_IO_DMA_CONTROL_MODE_XXX_GET(h,g)	This retrieves a current setting. *
HPDI32_IO_DMA_CONTROL_MODE_XXX_MANUAL(h)	This requests a setting of <i>manual</i> . *
HPDI32_IO_DMA_CONTROL_MODE_XXX_RESET(h)	This resets a setting. *
HPDI32_IO_DMA_CONTROL_MODE_XXX_SET(h,s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.10. I/O Parameter: DMA Priority

The purpose of this parameter is to modify and report the DMA data transfer priority assigned during I/O requests. This parameter is applicable only when applications opt to use DMA for simultaneous I/O reads and writes. The following tables describe the macros associated with this parameter.

**NOTE:** If I/O is active in both directions (read and write) at the same time, and both have the same priority, then this results in rotating priority. This occurs whether the priority for both is either enabled or disabled.

Macro (Parameter)	Description
HPDI32_IO_DMA_PRIORITY	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_DMA_PRIORITY_DISABLE	This selects the <i>disable</i> option, which permits the other I/O to have priority.
HPDI32_IO_DMA_PRIORITY_ENABLE	This selects the <i>enable</i> option, which requests that this I/O be have priority.
HPDI32_IO_DMA_PRIORITY_RX_DEFAULT	This is the Rx default, which is <i>disable</i> .
HPDI32_IO_DMA_PRIORITY_TX_DEFAULT	This is the Tx default, which is <i>enable</i> .

Macro (Services)	Description
HPDI32_IO_DMA_PRIORITY_GET(h,w,g)	This retrieves a current setting.
HPDI32_IO_DMA_PRIORITY_SET(h,w,s)	This requests a setting change.
HPDI32_IO_DMA_PRIORITY_XXX_DISABLE(h)	This requests a setting of <i>disable</i> . *
HPDI32_IO_DMA_PRIORITY_XXX_ENABLE(h)	This requests a setting of <i>enable</i> . *
HPDI32_IO_DMA_PRIORITY_XXX_GET(h,g)	This retrieves a current setting. *
HPDI32_IO_DMA_PRIORITY_XXX_RESET(h)	This resets a setting. *
HPDI32_IO_DMA_PRIORITY_XXX_SET(h,s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.11. I/O Parameter: Mode

The purpose of this parameter is to modify and report the data transfer mode used by the API during I/O requests. The following tables describe the macros associated with this parameter.

**NOTE:** For DMA based I/O using Application Buffers, the buffer must be both readable and writable. In some cases this means that buffers cannot be declared as `const` or `static const`. I/O requests will fail if the buffer does not have read/write access.

Macro (Parameter)	Description
HPDI32_IO_MODE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_MODE_DEFAULT	This selects the default, which is Demand Mode DMA.
HPDI32_IO_MODE_DMA	This selects the non-Demand Mode DMA, which is a blind transfer option.

	This option is further configurable as <i>automatic</i> or <i>manual</i> via the Non-Demand Mode DMA parameter.
HPDI32_IO_MODE_DMDMA	This selects Demand Mode DMA, which is the most efficient mode.
HPDI32_IO_MODE_PIO	This selects Programmed I/O, which used repetitive register reads and writes.

Macro (Services)	Description
HPDI32_IO_MODE_GET(h, w, g)	This retrieves a current setting.
HPDI32_IO_MODE_RESET(h, w)	This resets a setting.
HPDI32_IO_MODE_SET(h, w, s)	This requests a setting change.
HPDI32_IO_MODE_XXX_DMA(h)	This requests a setting of DMA. *
HPDI32_IO_MODE_XXX_DMDMA(h)	This requests a setting of DMDMA. *
HPDI32_IO_MODE_XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IO_MODE_XXX_PIO(h)	This requests a setting of PIO. *
HPDI32_IO_MODE_XXX_RESET(h)	This resets a setting. *
HPDI32_IO_MODE_XXX_SET(h, s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.12. I/O Parameter: Overlap Enable

The purpose of this parameter is to modify and report on the API's foreground or background processing of I/O requests. When I/O requests are made the API will use this parameter's setting to control how the request is processed. If the option is enabled then processing occurs as overlapped I/O. Otherwise it is performed as blocking I/O. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_IO_OVERLAP_ENABLE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_OVERLAP_ENABLE_DEFAULT	This selects the default, which is <i>no</i> .
HPDI32_IO_OVERLAP_ENABLE_NO	This selects the <i>no</i> option. I/O requests block until the request completes or times out.
HPDI32_IO_OVERLAP_ENABLE_YES	This selects the <i>yes</i> option. I/O requests return immediately while the data transfer occurs in the background.

Macro (Services)	Description
HPDI32_IO_OVERLAP_ENABLE_GET(h, w, g)	This retrieves a current setting.
HPDI32_IO_OVERLAP_ENABLE_RESET(h, w)	This resets a setting.
HPDI32_IO_OVERLAP_ENABLE_SET(h, w, s)	This requests a setting change.
HPDI32_IO_OVERLAP_ENABLE_XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IO_OVERLAP_ENABLE_XXX_NO(h)	This requests a setting of <i>no</i> . *
HPDI32_IO_OVERLAP_ENABLE_XXX_RESET(h)	This resets a setting. *
HPDI32_IO_OVERLAP_ENABLE_XXX_SET(h, s)	This requests a setting change. *
HPDI32_IO_OVERLAP_ENABLE_XXX_YES(h)	This requests a setting of <i>yes</i> . *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.13. I/O Parameter: PIO Threshold

The purpose of this parameter is to modify and report the threshold for I/O requests where the API automatically reverts to PIO mode verses the configured mode. When I/O requests are made the API will compare the requested number of samples to this parameter's value. If the request is at or below this level, then PIO is used rather than the configured mode. This is because PIO is more efficient with smaller sized I/O requests. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_IO_PIO_THRESHOLD	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_PIO_THRESHOLD_DEFAULT	This selects the default, which is 16 samples.
HPDI32_IO_PIO_THRESHOLD_NONE	This sets the threshold to zero, which disables the feature.

Macro (Services)	Description
HPDI32_IO_PIO_THRESHOLD_GET(h, w, g)	This retrieves a current setting.
HPDI32_IO_PIO_THRESHOLD_RESET(h, w)	This resets a setting.
HPDI32_IO_PIO_THRESHOLD_SET(h, w, s)	This requests a setting change.
HPDI32_IO_PIO_THRESHOLD_XXX_GET(h, g)	This retrieves a current T setting. *
HPDI32_IO_PIO_THRESHOLD_XXX_NONE(h, s)	This requests a setting change to zero. *
HPDI32_IO_PIO_THRESHOLD_XXX_RESET(h)	This resets a setting. *
HPDI32_IO_PIO_THRESHOLD_XXX_SET(h, s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.14. I/O Parameter: Single Cycle

The purpose of this parameter is to modify and report a setting that tells the API how the HPDI32 responds during Demand Mode DMA transfers when the Tx FIFO becomes Almost Full or the Rx FIFO becomes Almost Empty. In essence, this parameter tells the API whether the HPDI32 slows or pauses data transfer between the respective FIFO and the DMA engine at the given fill level. When data transfer slows it is because the board reverts to using single cycle accesses to transfer data, meaning the Single Cycle firmware feature is Present. When data transfer pauses it is because the board momentarily halts data transfer, meaning the Single Cycle firmware feature is Absent. This is generally applicable only to HPDI32 boards with either older or custom firmware. On newer boards which include the Single Cycle Disable feature, the API ignores this parameter. This parameter is applicable only for Demand Mode DMA transfers whose data size, in bits, is less than the board's PCI bus size, in bits, and then only on those boards which do not include the Single Cycle Disable feature. This parameter can be ignored under all other circumstances. The following tables describe the macros associated with this parameter.

**NOTE:** There are no known cases where an HPDI32 has different Tx and Rx characteristics. Applications must therefore insure that the Tx and Rx parameters are set the same.

Macro (Parameter)	Description
HPDI32_IO_SINGLE_CYCLE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_SINGLE_CYCLE_ABSENT	Data transfer pauses as the Single Cycle feature is absent.
HPDI32_IO_SINGLE_CYCLE_DEFAULT	This selects the default, which is the Present option.
HPDI32_IO_SINGLE_CYCLE_PRESENT	Data transfer slows as the Single Cycle feature is present.

Macro (Services)	Description
HPDI32_IO_SINGLE_CYCLE_ABSENT(h, w)	This requests a setting change to Absent.
HPDI32_IO_SINGLE_CYCLE_GET(h, w, g)	This retrieves a current setting.
HPDI32_IO_SINGLE_CYCLE_PRESENT(h, w)	This requests a setting change to Present.
HPDI32_IO_SINGLE_CYCLE_RESET(h, w)	This resets a setting.
HPDI32_IO_SINGLE_CYCLE_SET(h, w, s)	This requests a setting change.
HPDI32_IO_SINGLE_CYCLE_XXX_ABSENT(h)	This requests a setting change to Absent. *
HPDI32_IO_SINGLE_CYCLE_XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IO_SINGLE_CYCLE_XXX_PRESENT(h)	This requests a setting change to Present. *
HPDI32_IO_SINGLE_CYCLE_XXX_RESET(h)	This resets a setting. *
HPDI32_IO_SINGLE_CYCLE_XXX_SET(h, s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.15. I/O Parameter: Status

The purpose of this parameter is to report the current I/O status. The status returned includes the set of GSC\_IO\_STATUS\_XXX fields and bits for the referenced I/O request, which may have completed or may still be active. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_IO_STATUS	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_IO_STATUS__GET(h, w, g)	This retrieves a current status.
HPDI32_IO_STATUS__XXX_GET(h, g)	This retrieves a current status. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

#### 6.4.16. I/O Parameter: Timeout

The purpose of this parameter is to modify and report the API's timeout limit for I/O requests. When I/O requests are made the API will terminate the request if it has not completed in the specified number of seconds. The following tables describe the macros associated with this parameter.

**NOTE:** Applications should avoid setting the timeout limit to zero (0) when using any form of DMA. Doing so may result in inefficient use of DMA and it may be noticeable slower than expected.

**NOTE:** When using Demand Mode DMA applications should set the timeout period long enough to guarantee that successful transfers complete before the timeout limit expires. This is because the transfer will be aborted when the timeout period lapses and because the amount of data transferred will be unknown.

Macro (Parameter)	Description
HPDI32_IO_TIMEOUT	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IO_TIMEOUT_DEFAULT	This selects the default, which is 10 seconds
HPDI32_IO_TIMEOUT_MAX	This selects the maximum timeout limit, which is one hour.
HPDI32_IO_TIMEOUT_NO_WAIT	This sets the timeout to zero (0) seconds. This means the I/O request will terminate rather than wait for additional data transfer to occur.

Macro (Services)	Description
HPDI32_IO_TIMEOUT__GET(h, w, g)	This retrieves a current setting.
HPDI32_IO_TIMEOUT__RESET(h, w)	This resets a setting.
HPDI32_IO_TIMEOUT__SET(h, w, s)	This requests a setting change.
HPDI32_IO_TIMEOUT__XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IO_TIMEOUT__XXX_NO_WAIT(h)	This requests a setting of <i>do not wait</i> . *
HPDI32_IO_TIMEOUT__XXX_RESET(h)	This resets a setting. *
HPDI32_IO_TIMEOUT__XXX_SET(h, s)	This requests a setting change. *

\* The XXX sequence refers to the following individual options: RX for the data reads and TX for the data writes.

## 6.5. Interrupt Parameters

The purpose of the Interrupt Parameters is to permit access to and control of the HPDI32 hardware based interrupts. All Interrupt Parameters are put in a default state when the device is opened and are returned to that state via the `hpdi32_init()` service. The hardware based interrupt configuration is returned to its default state via the `hpdi32_reset()` service. The configuration of the Interrupt Parameters is retained mostly within the HPDI32 firmware registers. Applications have access to the HPDI32 interrupt registers but it is advised that they be accessed only through the Interrupt Parameters services. When using the service `hpdi32_config()`, any number or combination of `HPDI32_WHICH_IRQ_XXX` bits may be used, even none. An interrupt is accessed only if it's respective "which" bit is set. If none is set, then no action will be taken.

**NOTE:** The interrupt related "which" bits include both general and specific definitions for those cable signals which can have dual functionality. The purpose of providing the different forms is to permit greater clarity in application code. These are for reference and usability purposes only and do not refer to different interrupts. In addition, use of any particular definition will not alter which functionality is active at any particular time.

Each of the Interrupt Parameters includes a number of utility service macros. Rather include all variations of these macros, the table list many using an `XXX` string. In the tables the `XXX` sequence generally refers to the following individual options: `TX_E`, `TX_AE`, `TX_AF`, `TX_F`, `RX_E`, `RX_AE`, `RX_AF`, `RX_F` for the Empty, Almost Empty, Almost Full and Full Tx and Rx fill levels, `C0A` and `C0I` for Cable Command Signal 0 active and inactive, respectively, `C1`, `C2`, `C3`, `C4`, `C5` and `C6` for Cable Command Signals one through six, the Flow Control configured Cable Command signals `FVB` and `FVE` for Frame Valid Begin and End, `LV` for Line Valid, `SV` for Status Valid, `RR` for Rx Ready, `TR` for Tx Ready, `RE` for Rx Enable and `TE` for Tx Enable, and the GPIO configured Cable Command signals `GPIO_0`, `GPIO_1`, `GPIO_2`, `GPIO_3`, `GPIO_4`, `GPIO_5`, as well as `GPIO_6H` and `GPIO_6L` for GPIO 6 High and Low.

The following table lists the Interrupt Parameters.

Parameter Macros	Description
<code>HPDI32_IRQ_CALLBACK_ARG</code>	This refers to an arbitrary, application supplied callback argument.
<code>HPDI32_IRQ_CALLBACK_FUNC</code>	This refers to an application supplied callback function for interrupt notification.
<code>HPDI32_IRQ_ENABLE</code>	This refers to enabling or disabling an interrupt.
<code>HPDI32_IRQ_STATE</code>	This refers to the state of an interrupt source.
<code>HPDI32_IRQ_TRIGGER_CONFIG</code>	This refers to the trigger configuration for an interrupt source.

### 6.5.1. Interrupt Parameter: Callback Argument

The purpose of this parameter is to modify and report the application provided argument that is receives as "arg2" during an interrupt callback event. The following tables describe the macros associated with this parameter.

**NOTE:** Applications must remember that the macros `GSC_NO_CHANGE` and `GSC_DEFAULT` have special meaning when applying parameter modifications. If the application specific value being supplied for this parameter happens to equal either of these values, then the results will be according to the API's use of these special values rather than the applications intent.

**NOTE:** This parameter can be accessed and altered during the callback, but the callback must return before subsequent callbacks can be made on the same interrupt.

Macro (Parameter)	Description
<code>HPDI32_IRQ_CALLBACK_ARG</code>	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IRQ_CALLBACK_ARG_DEFAULT	This is the default, which is zero.

Macro (Services)	Description
HPDI32_IRQ_CALLBACK_ARG_GET(h, w, g)	This retrieves a current setting.
HPDI32_IRQ_CALLBACK_ARG_RESET(h, w)	This resets a setting.
HPDI32_IRQ_CALLBACK_ARG_SET(h, w, s)	This requests a setting change.
HPDI32_IRQ_CALLBACK_ARG_XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IRQ_CALLBACK_ARG_XXX_RESET(h)	This resets a setting. *
HPDI32_IRQ_CALLBACK_ARG_XXX_SET(h, s)	This requests a setting change. *

\* The XXX sequence refers to the service macro options given in paragraph 6.5, page 86.

### 6.5.2. Interrupt Parameter: Callback Function

The purpose of this parameter is to modify and report the application provided callback function pointer for an interrupt callback event. The following tables describe the macros associated with this parameter.

**NOTE:** This parameter can be accessed and altered during the callback, but the callback must return before subsequent callbacks can be made on the same interrupt.

Macro (Parameter)	Description
HPDI32_IRQ_CALLBACK_FUNC	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IRQ_CALLBACK_FUNC_DEFAULT	This is the default, which is NULL.

Macro (Services)	Description
HPDI32_IRQ_CALLBACK_FUNC_GET(h, w, g)	This retrieves a current function pointer.
HPDI32_IRQ_CALLBACK_FUNC_RESET(h, w)	This resets the function pointer.
HPDI32_IRQ_CALLBACK_FUNC_SET(h, w, s)	This requests a function pointer change.
HPDI32_IRQ_CALLBACK_FUNC_XXX_GET(h, g)	This retrieves a current function pointer. *
HPDI32_IRQ_CALLBACK_FUNC_XXX_RESET(h)	This resets a function pointer. *
HPDI32_IRQ_CALLBACK_FUNC_XXX_SET(h, s)	This requests a function pointer change. *

\* The XXX sequence refers to the service macro options given in paragraph 6.5, page 86.

### 6.5.3. Interrupt Parameter: Enable

The purpose of this parameter is to modify and report the enabled state of the respective interrupt. The following tables describe the macros associated with this parameter.

**NOTE:** The Rx FIFO Almost Empty and Rx FIFO Empty interrupts may be used for I/O read requests so should not be disabled by applications. The Tx FIFO Almost Full and Tx FIFO Full interrupts may be used for I/O write requests so should not be disabled by applications. Disabling any of these interrupts can interfere with normal I/O operations. This could result in reduced I/O performance or even data loss.

**NOTE:** Utility access macros are not provided for the Rx FIFO Almost Empty, Rx FIFO Empty, Tx FIFO Almost Full and Tx FIFO Full interrupts. This is to discourage applications from disabling these interrupts.

**NOTE:** When supplying the GSC\_DEFAULT macro as an assignment value for this parameter using any of the below utility service macros, be sure to supply only a single “which” bit. If this is not done, the default assigned will be for that interrupt with the lowest value “which” bit specified.

Macro (Parameter)	Description
HPDI32_IRQ_ENABLE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IRQ_ENABLE_DEFAULT	This is the default which is disabled.
HPDI32_IRQ_ENABLE_NO	This option disables the interrupt.
HPDI32_IRQ_ENABLE_YES	This option enables the interrupt.

Macro (Services)	Description
HPDI32_IRQ_ENABLE_GET(h, w, g)	This retrieves a current setting.
HPDI32_IRQ_ENABLE_RESET(h, w)	This resets a setting.
HPDI32_IRQ_ENABLE_SET(h, w, s)	This requests a setting change.
HPDI32_IRQ_ENABLE_XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IRQ_ENABLE_XXX_NO(h)	This requests that an interrupt be disabled. *
HPDI32_IRQ_ENABLE_XXX_RESET(h)	This resets a setting. *
HPDI32_IRQ_ENABLE_XXX_SET(h, s)	This requests a setting change. *
HPDI32_IRQ_ENABLE_XXX_YES(h)	This requests that an interrupt be enabled. *

\* The XXX sequence refers to the service macro options given in paragraph 6.5, page 86. Refer to the above notes for certain exceptions.

#### 6.5.4. Interrupt Parameter: State

The purpose of this read-only parameter is to report the state of the respective interrupt source. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_IRQ_STATE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IRQ_STATE_ACTIVE	This reflects that the source was active.
HPDI32_IRQ_STATE_INACTIVE	This reflects that the source was inactive.

Macro (Services)	Description
HPDI32_IRQ_STATE_GET(h, w, g)	This retrieves a current state.
HPDI32_IRQ_STATE_XXX_GET(h, g)	This retrieves a current state. *

\* The XXX sequence refers to the service macro options given in paragraph 6.5, page 86.

#### 6.5.5. Interrupt Parameter: Trigger Configuration

The purpose of this parameter is to modify and report the Trigger Configuration of an interrupt. The following tables describe the macros associated with this parameter.

**NOTE:** The Rx FIFO Almost Empty and Rx FIFO Empty interrupts may be used for I/O read requests. The Tx FIFO Almost Full and Tx FIFO Full interrupts may be used for I/O write requests. The Trigger Configuration for these interrupts should not be altered by applications as it can interfere with normal I/O operations. This could result in reduced I/O performance or even data loss.



**NOTE:** Utility access macros are not provided for the Rx FIFO Almost Empty, Rx FIFO Empty, Tx FIFO Almost Full and Tx FIFO Full interrupt Trigger Configuration parameters. This is to discourage applications from altering this parameter for these interrupts.

Macro (Parameter)	Description
HPDI32_IRQ_TRIGGER_CONFIG	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_IRQ_TRIGGER_CONFIG_DEFAULT	This option refers to default which is Edge Hi.
HPDI32_IRQ_TRIGGER_CONFIG_EDGE_HI	This option refers to triggering on a rising edge.
HPDI32_IRQ_TRIGGER_CONFIG_EDGE_LOW	This option refers to triggering on a falling edge.
HPDI32_IRQ_TRIGGER_CONFIG_LEVEL_HI	This option refers to triggering on a high level.
HPDI32_IRQ_TRIGGER_CONFIG_LEVEL_LOW	This option refers to triggering on a low level.

Macro (Services)	Description
HPDI32_IRQ_TRIGGER_CONFIG_GET(h, w, g)	This retrieves a current setting.
HPDI32_IRQ_TRIGGER_CONFIG_RESET(h, w)	This resets a setting.
HPDI32_IRQ_TRIGGER_CONFIG_SET(h, w, s)	This requests a setting change.
HPDI32_IRQ_TRIGGER_CONFIG_XXX_EDGE_HI(h)	This requests a trigger on a rising edge. *
HPDI32_IRQ_TRIGGER_CONFIG_XXX_EDGE_LOW(h)	This requests a trigger on a falling edge. *
HPDI32_IRQ_TRIGGER_CONFIG_XXX_GET(h, g)	This retrieves a current setting. *
HPDI32_IRQ_TRIGGER_CONFIG_XXX_LEV_HI(h)	This requests a trigger on a high level. *
HPDI32_IRQ_TRIGGER_CONFIG_XXX_LEV_LOW(h)	This requests a trigger on a low level. *
HPDI32_IRQ_TRIGGER_CONFIG_XXX_RESET(h)	This resets a setting. *
HPDI32_IRQ_TRIGGER_CONFIG_XXX_SET(h, s)	This requests a setting change. *

\* The XXX sequence refers to the service macro options given in paragraph 6.5, page 86. Refer to the above notes for certain exceptions.

## 6.6. Miscellaneous Parameters

The purpose of the Miscellaneous Parameters is to permit access to and control of HPDI32 parameters which do not readily fit into the other parameter categories. All Miscellaneous Parameters are put in a default state when the device is opened and are returned to that state via the `hpdi32_init()` service. The hardware based Miscellaneous Parameters are returned to their default states via the `hpdi32_reset()` service. The configuration of one or more Miscellaneous Parameters is retained within the HPDI32 firmware registers. Applications have access to these HPDI32 registers but it is advised that these parameters be accessed only through the Miscellaneous Parameter services. When using the service `hpdi32_config()`, the “which” bits argument is ignored. The following table lists the Miscellaneous Parameters.

Parameter Macros	Description
HPDI32_MISC_BOARD_JUMPERS	This refers to the board’s user jumpers.
HPDI32_MISC_FAVOR_TX	This refers to option of favoring transmit operation over receive operations for certain parameters.
HPDI32_MISC_FEATURES	This refers to the set of supported features.
HPDI32_MISC_MAP_GSC_REGS	This refers to mapping of the firmware registers into API and application memory space.
HPDI32_MISC_MAP_GSC_REGS_PTR	This refers to the application accessible pointer to the firmware registers.
HPDI32_MISC_MAP_PLX_REGS	This refers to mapping of the PLX feature set registers into API memory space.
HPDI32_MISC_PCI_BUS_WIDTH	This refers to width of the board’s PCI interface: 32 or 64-bits.
HPDI32_MISC_STRICT_ARGUMENTS	This refers to the processing of certain unrecognized parameter values when applying settings.

HPDI32_MISC_STRICT_CONFIG	This refers to the processing of invalid hardware configuration options when applying settings.
HPDI32_MISC_TX_RX_TRI_STATE	This refers to the tri-stating of the Tx Enable and Rx Enable cable signals when not being driven high.

### 6.6.1. Miscellaneous Parameter: Board Jumpers

The purpose of this read-only parameter is to report the state of the User Jumper pins on the board, for those that support the feature. The jumper state is reported in the lower two bits of the value retrieved. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_MISC_BOARD_JUMPERS	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_MISC_BOARD_JUMPERS__GET(h, g)	This retrieves the current state.

### 6.6.2. Miscellaneous Parameter: Favor Tx

The purpose of this parameter is to control and report the API's favoring of transmit operations over receive operations. When the transmitter is favored a small set of configurable parameters, when appropriately processed, are configured to favor transmit operations. If disabled, these same parameters, when appropriately processed, are configured to favor receive operations. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_MISC_FAVOR_TX	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_MISC_FAVOR_TX_DEFAULT	This is the default, which is <i>disable</i> .
HPDI32_MISC_FAVOR_TX_DISABLE	This option disables the option.
HPDI32_MISC_FAVOR_TX_ENABLE	This option enables the option.

Macro (Services)	Description
HPDI32_MISC_FAVOR_TX_GET(h, g)	This retrieves the current setting.
HPDI32_MISC_FAVOR_TX_NO(h)	This requests that the option be disabled.
HPDI32_MISC_FAVOR_TX_SET(h, s)	This requests a setting change.
HPDI32_MISC_FAVOR_TX_YES(h)	This requests that the option be enabled.

### 6.6.3. Miscellaneous Parameter: Features

The purpose of this read-only parameter is to report the presence of various firmware based HPDI32 features. While this is a read-only parameter the feature being tested must be specified in the respective structure's "set" argument. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_MISC_FEATURES	This is the identifier for this parameter.

Macro (Set Values)	Description
HPDI32_MISC_FEATURES_COUNT	This refers to the number of features supported by this parameter.
HPDI32_MISC_FEATURES_1_CYCLE_DISABLE	This refers to the Board Control Register's Single Cycle Disable bit.
HPDI32_MISC_FEATURES_DMA_CH1	This refers to support for transmitting with Demand Mode DMA using DMA channel 1

HPDI32_MISC_FEATURES_FIFO_SIZE	This refers to the Tx/Rx FIFO Size Registers.
HPDI32_MISC_FEATURES_FSR	This refers to the Feature Set Register.
HPDI32_MISC_FEATURES_GPIO_0_5	This refers to the GPIO 0 to 5 signals available on the external cable interface.
HPDI32_MISC_FEATURES_GPIO_6	This refers to the GPIO 6 signal available on the external cable interface.
HPDI32_MISC_FEATURES_ICR	This refers to the interrupt configuration registers IELR and IHLR.
HPDI32_MISC_FEATURES_OVR_UNDR_RUN	This refers to the Tx/Rx FIFO Over/Under Run bits.
HPDI32_MISC_FEATURES_TX_AUTO_STOP	This refers to the Board Control Register's Tx Start Auto Clear Disable bit.
HPDI32_MISC_FEATURES_USER_JUMPERS	This refers to the presence of the user configurable jumpers on the board.

Macro (Get Values)	Description
HPDI32_MISC_FEATURES_ABSENT	This means the feature is absent from the HPDI32.
HPDI32_MISC_FEATURES_PRESENT	This means the feature is present in the HPDI32.

Macro (Services)	Description
HPDI32_MISC_FEATURES__GET(h, s, g)	This requests support status for a feature.
HPDI32_MISC_FEATURES__XXX(h, g)	This requests support status for feature XXX. *

\* The XXX sequence refers to the parameter value extensions given in the Set Values table. The extension is that test that follows the base parameter macro text.

#### 6.6.4. Miscellaneous Parameter: GSC Register Mapping

The purpose of this parameter is to control and report the mapping of GSC registers into application and API memory space. This parameter should always be enabled, even if unused by applications. If it is disabled, the API's access to HPDI32 firmware registers operates with reduced efficiency. The following tables describe the macros associated with this parameter.

**NOTE:** There are circumstances where this feature cannot be enabled and utilized. This is usually limited to embedded hosts where the BIOS doesn't place all PCI memory access regions on CPU Page Size boundaries. If the BIOS cannot be configured to utilize such boundaries, then API performance is degraded.

**NOTE:** Parameter access utility macros are limited for this parameter as it should always be enabled. The parameter should only be disabled for testing purposes.

Macro (Parameter)	Description
HPDI32_MISC_MAP_GSC_REGS	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_MISC_MAP_GSC_REGS_DEFAULT	This is the default, which is <i>enabled</i> .
HPDI32_MISC_MAP_GSC_REGS_DISABLE	This refers to the <i>disabled</i> state. When disabled access to firmware registers must go through the Device Driver, which reduces efficiency.
HPDI32_MISC_MAP_GSC_REGS_ENABLE	This refers to the <i>enabled</i> state. When enabled access to firmware registers is done entirely within the API, which increases efficiency.

Macro (Services)	Description
HPDI32_MISC_MAP_GSC_REGS__ENABLE(h)	This requests that the option be enabled.

HPDI32_MISC_MAP_GSC_REGS_GET(h,g)	This requests the current setting.
HPDI32_MISC_MAP_GSC_REGS_RESET(h)	This resets the setting.
HPDI32_MISC_MAP_GSC_REGS_SET(h,s)	This requests a setting change.

### 6.6.5. Miscellaneous Parameter: GSC Register Mapping Pointer

The purpose of this read-only parameter is to retrieve the pointer the API uses for direct access to HPDI32 firmware registers. If the GSC Register Mapping feature is enabled the application can use the pointer to directly access HPDI32 firmware registers. If disabled, the pointer returned is NULL. The following tables describe the macros associated with this parameter.

**NOTE:** There are circumstances where this feature cannot be utilized. This is usually limited to embedded hosts here the BIOS doesn't place all PCI memory access regions on CPU Page Size boundaries. If the BIOS cannot be configured to utilize such boundaries, then API performance is degraded.

Macro (Parameter)	Description
HPDI32_MISC_MAP_GSC_REGS_PTR	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_MISC_MAP_GSC_REGS_PTR_GET(h,g)	This requests the current pointer.

### 6.6.6. Miscellaneous Parameter: PLX Register Mapping

The purpose of this parameter is to control and report the mapping of PLX registers into API memory space. This parameter should always be enabled, even though it is not directly usable by applications. If it is disabled, the API's access to PLX registers operates with reduced efficiency. The following tables describe the macros associated with this parameter.

**NOTE:** There are circumstances where this feature cannot be enabled and utilized. This is usually limited to embedded hosts here the BIOS doesn't place all PCI memory access regions on CPU Page Size boundaries. If the BIOS cannot be configured to utilize such boundaries, then API performance is degraded.

**NOTE:** Parameter access utility macros are limited for this parameter as it should always be enabled. The parameter should only be disabled for testing purposes.

Macro (Parameter)	Description
HPDI32_MISC_MAP_PLX_REGS	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_MISC_MAP_PLX_REGS_DEFAULT	This is the default, which is <i>enabled</i> .
HPDI32_MISC_MAP_PLX_REGS_DISABLE	This refers to the <i>disabled</i> state. When disabled access to PLX registers must go through the Device Driver, which reduces efficiency.
HPDI32_MISC_MAP_PLX_REGS_ENABLE	This refers to the <i>enabled</i> state. When enabled access to PLX registers is done entirely within the API, which increases efficiency.

Macro (Services)	Description
HPDI32_MISC_MAP_PLX_REGS_ENABLE(h)	This request that the option be enabled.
HPDI32_MISC_MAP_PLX_REGS_GET(h,g)	This requests the current setting.
HPDI32_MISC_MAP_PLX_REGS_RESET(h)	This resets the current setting.
HPDI32_MISC_MAP_PLX_REGS_SET(h,s)	This request a change to the current setting.

### 6.6.7. Miscellaneous Parameter: PCI Bus Width

The purpose of this read-only parameter is to retrieve the HPDI32 board's PCI bus width. This parameter is provided for informational purposes only. The following tables describe the macros associated with this parameter.

**NOTE:** While this parameter identifies the bus width of the board's PCI interface, this has no bearing on the size of the PCI slot the board is plugged into. The API does not have this information.

Macro (Parameter)	Description
HPDI32_MISC_PCI_BUS_WIDTH	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_MISC_PCI_BUS_WIDTH_32	This reflects that the board has a 32-bit bus.
HPDI32_MISC_PCI_BUS_WIDTH_64	This reflects that the board has a 64-bit bus.

Macro (Services)	Description
HPDI32_MISC_PCI_BUS_WIDTH__GET(h, g)	This requests the board's PCI bus width.

### 6.6.8. Miscellaneous Parameter: Strict Arguments

The purpose of this parameter is to control and retrieve the setting that governs the API's handling of certain unrecognized values. For example, if the setting supplied when adjusting this parameter is not listed in the appropriate table below, then the API can either respond with an error condition or infer the application's intent per the value that was received. If Strict Argument processing is enabled, then processing is terminated with an error status. Otherwise the API is lenient and will try to proceed gracefully. This policy is applicable to parameter processing only, and applies to most parameters. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_MISC_STRICT_ARGUMENTS	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_MISC_STRICT_ARGUMENTS_DEFAULT	This is the default, which is lenient processing.
HPDI32_MISC_STRICT_ARGUMENTS_DISABLE	This refers to lenient processing.
HPDI32_MISC_STRICT_ARGUMENTS_ENABLE	This refers to strict processing.

Macro (Services)	Description
HPDI32_MISC_STRICT_ARGUMENTS__GET(h, g)	This requests the current setting.
HPDI32_MISC_STRICT_ARGUMENTS__NO(h)	This requests lenient processing.
HPDI32_MISC_STRICT_ARGUMENTS__RESET(h)	This resets the setting.
HPDI32_MISC_STRICT_ARGUMENTS__SET(h, s)	This requests a setting change.
HPDI32_MISC_STRICT_ARGUMENTS__YES(h)	This requests strict processing.

### 6.6.9. Miscellaneous Parameter: Strict Configuration

The purpose of this parameter is to control and retrieve the setting that governs the API's handling of certain invalid hardware configuration requests. Support for the parameter is not yet incorporated into the API. For example, if altering a GPIO setting for a Cable Command signal not configured for GPIO, the API can either respond with an error condition or infer the application's intent per the request. If strict processing is enabled, then processing is terminated with an error status. Otherwise the API is lenient and will try to proceed gracefully. This policy is applicable to parameter processing only, and applies to hardware based parameters only. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_MISC_STRICT_CONFIG	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_MISC_STRICT_CONFIG_DEFAULT	This is the default, which is lenient processing.
HPDI32_MISC_STRICT_CONFIG_DISABLE	This refers to lenient processing.
HPDI32_MISC_STRICT_CONFIG_ENABLE	This refers to strict processing.

Macro (Services)	Description
HPDI32_MISC_STRICT_CONFIG_GET(h, g)	This requests the current setting.
HPDI32_MISC_STRICT_CONFIG_NO(h)	This requests lenient processing.
HPDI32_MISC_STRICT_CONFIG_RESET(h)	This resets the setting.
HPDI32_MISC_STRICT_CONFIG_SET(h, s)	This requests a setting change.
HPDI32_MISC_STRICT_CONFIG_YES(h)	This requests strict processing.

#### 6.6.10. Miscellaneous Parameter: Tx/Rx Tri-State

The purpose of this parameter is to control and retrieve the HPDI32's tri-stating of the Tx Enable and Rx Enable signals when not driven high. (In the hardware user manual this is referred to as Test Mode as it was introduced for connected two HPDI32 boards back-to-back for testing purposes.) The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_MISC_TX_RX_TRI_STATE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_MISC_TX_RX_TRI_STATE_DEFAULT	This is the default, which is <i>disabled</i> .
HPDI32_MISC_TX_RX_TRI_STATE_DISABLE	This refers to the <i>disabled</i> option, in which the Tx Enable and Rx Enable signals are always driven.
HPDI32_MISC_TX_RX_TRI_STATE_ENABLE	This refers to the <i>enabled</i> option, in which the Tx Enable and Rx Enable signals are driven only when high.

Macro (Services)	Description
HPDI32_MISC_TX_RX_TRI_STATE_ENABLE_GET(h, g)	This requests the current setting.
HPDI32_MISC_TX_RX_TRI_STATE_ENABLE_NO(h)	This request that the option be <i>disabled</i> .
HPDI32_MISC_TX_RX_TRI_STATE_ENABLE_RESET(h)	This resets the setting.
HPDI32_MISC_TX_RX_TRI_STATE_ENABLE_SET(h, s)	This requests a setting change.
HPDI32_MISC_TX_RX_TRI_STATE_ENABLE_YES(h)	This request that the option be <i>enabled</i> .

### 6.7. Receiver Parameters

The purpose of the Receiver Parameters is to permit access to and control of those parameters that pertain exclusively to the HPDI32's receiver features. All Receiver Parameters are put in a default state when the device is opened and are returned to that state via the `hpdi32_init()` and `hpdi32_reset()` services. The configuration of these parameters is retained within the HPDI32 firmware registers. Applications have access to the HPDI32 registers but it is advised that these features be accessed only through the Receiver Parameter services. When using the service `hpdi32_config()`, the "which" bits argument is ignored. The following table lists the Receiver Parameters.

Parameter Macros	Description
HPDI32_RX_ENABLE	This refers to enabling or disabling the receiver.
HPDI32_RX_OVERRUN	This refers to the Rx FIFO being overrun with additional data when already full.
HPDI32_RX_ROW_COUNT	This refers to the number of data samples received during a frame while the

	Line Valid signal is active.
HPDI32_RX_STATE	This refers to active or inactive state of the receiver.
HPDI32_RX_STATUS_COUNT	This refers to the number of data samples received during a frame while the Status Valid signal is active.
HPDI32_RX_UNDER_RUN	This refers to the Rx FIFO being read when it is already empty.

### 6.7.1. Receiver Parameter: Rx Enable

The purpose of this parameter is to control and retrieve the enable state of the receiver. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_RX_ENABLE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_RX_ENABLE_DEFAULT	This is the default, which is <i>disabled</i> .
HPDI32_RX_ENABLE_NO	This refers to the <i>disabled</i> option, when prevents data transfer.
HPDI32_RX_ENABLE_YES	This refers to the <i>enabled</i> option, which permits data transfer.

Macro (Services)	Description
HPDI32_RX_ENABLE_GET(h, g)	This requests the current setting.
HPDI32_RX_ENABLE_NO(h)	This request that the option be <i>disabled</i> .
HPDI32_RX_ENABLE_RESET(h)	This reset the setting.
HPDI32_RX_ENABLE_SET(h, s)	This requests a setting change.
HPDI32_RX_ENABLE_YES(h)	This request that the option be <i>enabled</i> .

### 6.7.2. Receiver Parameter: Rx Overrun

The purpose of this parameter is to control and retrieve the Rx Overrun condition, when supported in the HPDI32. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_RX_OVERRUN	This is the identifier for this parameter.

Macro (Set Values)	Description
HPDI32_RX_OVERRUN_CLEAR	This refers to clearing the condition.
HPDI32_RX_OVERRUN_DEFAULT	This is the default, which is to <i>clear</i> the condition.
HPDI32_RX_OVERRUN_IGNORE	This refers to ignoring the condition (do not clear it).

Macro (Get Values)	Description
HPDI32_RX_OVERRUN_NO	This reflects that the condition does not exist.
HPDI32_RX_OVERRUN_YES	This reflects that the condition does exist.

Macro (Services)	Description
HPDI32_RX_OVERRUN__CLEAR(h)	This requests that the condition be <i>cleared</i> .
HPDI32_RX_OVERRUN__GET(h, g)	This requests the current condition.
HPDI32_RX_OVERRUN__SET(h, s)	This requests a setting change.

### 6.7.3. Receiver Parameter: Row Count

The purpose of this read-only parameter is to retrieve the count of data samples received over the external cable interface during the last frame's Line Valid active period. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_RX_ROW_COUNT	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_RX_ROW_COUNT__GET(h, g)	This requests the current count.

#### 6.7.4. Receiver Parameter: State

The purpose of this parameter is to retrieve the Rx Enabled state of the receiver. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_RX_STATE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_RX_STATE_ACTIVE	This means the receiver is active since it is enabled.
HPDI32_RX_STATE_INACTIVE	This means the receiver is inactive since it is disabled.

Macro (Services)	Description
HPDI32_RX_STATE__GET(h, g)	This requests the current state.

#### 6.7.5. Receiver Parameter: Status Count

The purpose of this read-only parameter is to retrieve the count of data samples received over the external cable interface during the last frame's Status Valid active period. In the `hpd32_rx_config_t` structure the parameter is listed separately. In the `hpd32_config()` service the parameter is accessed only via the parameter identifier. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_RX_STATUS_COUNT	This is the identifier for this parameter.

Macro (Services)	Description
HPDI32_RX_STATUS_COUNT__GET(h, g)	This requests the current count.

#### 6.7.6. Receiver Parameter: Rx Under Run

The purpose of this parameter is to control and retrieve the Rx Under Run condition, when supported in the HPDI32. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_RX_UNDER_RUN	This is the identifier for this parameter.

Macro (Set Values)	Description
HPDI32_RX_UNDER_RUN_CLEAR	This refers to clearing the condition.
HPDI32_RX_UNDER_RUN_DEFAULT	This is the default, which is to <i>clear</i> the condition.
HPDI32_RX_UNDER_RUN_IGNORE	This refers to ignoring the condition (do not clear it).

Macro (Get Values)	Description
HPDI32_RX_UNDER_RUN_NO	This reflects that the condition does not exist.
HPDI32_RX_UNDER_RUN_YES	This reflects that the condition does exist.

Macro (Services)	Description
HPDI32_RX_UNDER_RUN__CLEAR(h)	This requests that the condition be <i>cleared</i> .
HPDI32_RX_UNDER_RUN__GET(h, g)	This requests the current condition.



HPDI32_RX_UNDER_RUN__SET(h,s)	This requests a setting change.
-------------------------------	---------------------------------

## 6.8. Transmitter Parameters

The purpose of the Transmitter Parameters is to permit access to and control of those parameters that pertain exclusively to the HPDI32's transmitter features. All Transmitter Parameters are put in a default state when the device is opened and are returned to that state via the `hpdi32_init()` service. The firmware based Transmitter Parameters are also returned to their initial state via the `hpdi32_reset()` service. The configuration of some of these parameters is retained within the HPDI32 firmware registers. Applications have access to the HPDI32 registers but it is advised that these features be accessed only through the Transmitter Parameter services. When using the service `hpdi32_config()`, the "which" bits argument is ignored. The following table lists the Receiver Parameters.

Parameter Macros	Description
HPDI32_TX_AUTO_START	This refers to automatically starting transmission when a write request occurs.
HPDI32_TX_AUTO_STOP	This refers to automatically stopping the transmitter when the Tx FIFO hits empty, even if momentarily.
HPDI32_TX_CLOCK_DIVIDER	This refers to the divider that goes between the on-board oscillator and the transmitter.
HPDI32_TX_ENABLE	This refers to enabling or disabling the transmitter.
HPDI32_TX_FLOW_CONTROL	This refers to enabling or disabling data flow out the cable.
HPDI32_TX_LINE_VALID_OFF_COUNT	This refers to length of the Line Valid off period.
HPDI32_TX_LINE_VALID_ON_COUNT	This refers to length of the Line Valid on period.
HPDI32_TX_OVERRUN	This refers to Tx FIFO receiving data when it is already full.
HPDI32_TX_REMOTE_THROTTLE	This refers to the remote hardware control the flow of transmit data.
HPDI32_TX_REMOTE_THROTTLE_STATE	This refers to state of the remote hardware's data flow control input.
HPDI32_TX_STATE	This refers to state of the transmitter.
HPDI32_TX_STATUS_VALID_COUNT	This refers to length of the Status Valid on period.
HPDI32_TX_STATUS_VALID_MIRROR	This refers to mirroring of the Status Valid pulse on the Line Valid signal.

### 6.8.1. Transmitter Parameter: Auto Start

The purpose of this parameter is to control and retrieve the API's Auto Start feature for the transmitter. If enabled (the default), then the API will automatically set the Tx Start bit in the firmware's Board Control Register to initiate data transfer during write requests. If the parameter is disabled, then the application is responsible for setting the Tx Start bit, when appropriate. The following tables describe the macros associated with this parameter.

**NOTE:** In the HPDI32 firmware, the Tx Start bit operates in parallel with the Tx Remote Throttling feature. If Remote Throttling is enabled and the Tx Start bit is set, then data will be transferred even if the Remote Throttling input from the remote device says to halt data transfer. This is likely to result in data loss.

**NOTE:** When the Auto Start parameter is enabled, the API will disable the Remote Throttle parameter. When the Remote Throttle parameter is enabled, the API will disable the Auto Start parameter.

Macro (Parameter)	Description
HPDI32_TX_AUTO_START	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_AUTO_START_DEFAULT	This is the default, which is the <i>disable</i> option.
HPDI32_TX_AUTO_START_NO	This disabled the option.
HPDI32_TX_AUTO_START_YES	This enables the option.

Macro (Services)	Description
HPDI32_TX_AUTO_START__GET(h, g)	This requests the current setting.
HPDI32_TX_AUTO_START__NO(h)	This request that the option be <i>disabled</i> .
HPDI32_TX_AUTO_START__RESET(h)	This resets the setting.
HPDI32_TX_AUTO_START__SET(h, s)	This requests a setting change.
HPDI32_TX_AUTO_START__YES(h)	This request that the option be <i>enabled</i> .

### 6.8.2. Transmitter Parameter: Auto Stop

The purpose of this parameter is to control and retrieve the API's Auto Stop feature for the transmitter. If disabled (the default), then once data transmission from the Tx FIFO begins, data transfer from the FIFO will remain enabled (permitting subsequent data flow) even if the FIFO becomes empty. The empty situation can occur because additional data is not being put into the FIFO or because the FIFO runs dry when data is pulled out faster than it is put in. In general, the FIFO can run dry either because the data transmission rate exceeds the PCI bus transfer rate or because the PCI bus temporarily stalls under system loading and overhead issues. If enabled, then transmission becomes disabled the instant the Tx FIFO hits the empty state. The following tables describe the macros associated with this parameter.

**NOTE:** This parameter should remain disabled unless an application has a specific need to enable it. If enabled, data transmission can appear to halt inexplicably on some systems, or it can have a negative impact on overall throughput.

**NOTE:** The Tx Auto Stop feature operates both when the Tx Auto Start parameter is enabled and when the Tx Flow Control parameter enables transfer.

Macro (Parameter)	Description
HPDI32_TX_AUTO_STOP	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_AUTO_STOP_DEFAULT	This is the default, which is the <i>enable</i> option.
HPDI32_TX_AUTO_STOP_NO	This disabled the option.
HPDI32_TX_AUTO_STOP_YES	This enables the option.

Macro (Services)	Description
HPDI32_TX_AUTO_STOP__GET(h, g)	This requests the current setting.
HPDI32_TX_AUTO_STOP__NO(h)	This request that the option be <i>disabled</i> .
HPDI32_TX_AUTO_STOP__RESET(h)	This resets the setting.
HPDI32_TX_AUTO_STOP__SET(h, s)	This requests a setting change.
HPDI32_TX_AUTO_STOP__YES(h)	This request that the option be <i>enabled</i> .

### 6.8.3. Transmitter Parameter: Tx Clock Divider

The purpose of this parameter is to control and retrieve the value in the HPDI32 Tx Clock Divider Register. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_CLOCK_DIVIDER	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_CLOCK_DIVIDER_DEFAULT	This is the default, which is zero (0).
HPDI32_TX_CLOCK_DIVIDER_MAX	This is the maximum value that can be written to the register.

Macro (Services)	Description
HPDI32_TX_CLOCK_DIVIDER_GET(h, g)	This requests the current setting.
HPDI32_TX_CLOCK_DIVIDER_SET(h, s)	This requests a setting change.

#### 6.8.4. Transmitter Parameter: Tx Enable

The purpose of this parameter is to control and retrieve the enable state of the transmitter. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_ENABLE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_ENABLE_DEFAULT	This is the default, which is <i>disabled</i> .
HPDI32_TX_ENABLE_NO	This refers to the <i>disabled</i> option, when prevents data transfer.
HPDI32_TX_ENABLE_YES	This refers to the <i>enabled</i> option, which permits data transfer.

Macro (Services)	Description
HPDI32_TX_ENABLE_GET(h, g)	This requests the current setting.
HPDI32_TX_ENABLE_NO(h)	This request that the option be <i>disabled</i> .
HPDI32_TX_ENABLE_RESET(h)	This resets the setting.
HPDI32_TX_ENABLE_SET(h, s)	This requests a setting change.
HPDI32_TX_ENABLE_YES(h)	This request that the option be <i>enabled</i> .

#### 6.8.5. Transmitter Parameter: Flow Control

The purpose of this parameter is to control and retrieve the API's enabling or inhibiting of transmit data flow when the transmitter is enabled. If enabled, then transmit data is permitted to flow. If disabled, then data flow is halted. This parameter operates by manipulating the Tx Start bit in the firmware's Board Control Register, which functions in parallel with the Tx Remote Throttling parameter. Manipulating this Flow Control parameter has no affect on the Tx Remote Throttling parameter and should be used only when the Remote Throttling parameter is disabled. The following tables describe the macros associated with this parameter.

**NOTE:** In the HPDI32 firmware, the Tx Start bit operates in parallel with the Tx Remote Throttling feature. If Remote Throttling is enabled and the Tx Start bit is set, then data will be transferred even if the Remote Throttling input from remote device says to halt data transfer. This is likely to result in data loss. The Tx Flow Control parameter should not be exercised while Remote Throttling is enabled.

**NOTE:** Use of this parameter is applicable mostly when Tx Auto Start is disabled. While it can be used as data is flowing out the external cable interface, halted data flow will resume movement as the API exercises the Tx Auto Start feature.

**NOTE:** Applications applying this parameter to halt data flow must be aware that it could result in an I/O timeout.

Macro (Parameter)	Description
HPDI32_TX_FLOW_CONTROL	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_FLOW_CONTROL_DEFAULT	This is the default, which is to do nothing.
HPDI32_TX_FLOW_CONTROL_DISABLE	This disables data flow.
HPDI32_TX_FLOW_CONTROL_ENABLE	This enables data flow.
HPDI32_TX_FLOW_CONTROL_IGNORE	This option takes no action.

Macro (Services)	Description
HPDI32_TX_FLOW_CONTROL_GET(h, g)	This requests the current setting.
HPDI32_TX_FLOW_CONTROL_RESET(h)	This resets the setting.
HPDI32_TX_FLOW_CONTROL_SET(h, s)	This requests a setting change.
HPDI32_TX_FLOW_CONTROL_START(h)	This request that the data flow.
HPDI32_TX_FLOW_CONTROL_STOP(h)	This request that the data stop flowing.

### 6.8.6. Transmitter Parameter: Line Valid Off Count

The purpose of this parameter is to control and retrieve the number of cable clock cycles that the Line Valid signal is held low before going high, during a frame. This parameter operates by accessing the board's Tx Line Invalid Length Count Register. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_LINE_VALID_OFF_COUNT	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_LINE_VALID_OFF_COUNT_DEFAULT	This is the default, which disables the “off” period.
HPDI32_TX_LINE_VALID_OFF_COUNT_DISABLE	This disables the “off” period, which sets the register to zero (0).
HPDI32_TX_LINE_VALID_OFF_COUNT_MAX	This is the maximum period length.

Macro (Services)	Description
HPDI32_TX_LINE_VALID_OFF_COUNT_DISABLE(h)	This requests that the “off” period be disabled.
HPDI32_TX_LINE_VALID_OFF_COUNT_GET(h, g)	This requests the current setting.
HPDI32_TX_LINE_VALID_OFF_COUNT_RESET(h)	This resets the setting.
HPDI32_TX_LINE_VALID_OFF_COUNT_SET(h, s)	This requests a setting change.

### 6.8.7. Transmitter Parameter: Line Valid On Count

The purpose of this parameter is to control and retrieve the number of cable clock cycles that the Line Valid signal is held high after being low, during a frame. This parameter operates by accessing the board's Tx Line Valid Length Count Register. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_LINE_VALID_ON_COUNT	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_LINE_VALID_ON_COUNT_DEFAULT	This is the default, which disables the “on” period.
HPDI32_TX_LINE_VALID_ON_COUNT_DISABLE	This disables the “on” period, which sets the register to zero (0).
HPDI32_TX_LINE_VALID_ON_COUNT_MAX	This is the maximum period length.

Macro (Services)	Description
HPDI32_TX_LINE_VALID_ON_COUNT_DISABLE(h)	This requests that the “on” period be disabled.
HPDI32_TX_LINE_VALID_ON_COUNT_GET(h, g)	This requests the current setting.
HPDI32_TX_LINE_VALID_ON_COUNT_RESET(h)	This resets the setting.
HPDI32_TX_LINE_VALID_ON_COUNT_SET(h, s)	This requests a setting change.

### 6.8.8. Transmitter Parameter: Tx Overrun

The purpose of this parameter is to control and retrieve the Tx Overrun condition, when supported in the HPDI32. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_OVERRUN	This is the identifier for this parameter.

Macro (Set Values)	Description
HPDI32_TX_OVERRUN_CLEAR	This refers to clearing the condition.
HPDI32_TX_OVERRUN_DEFAULT	This is the default, which is to <i>clear</i> the condition.
HPDI32_TX_OVERRUN_IGNORE	This refers to ignoring the condition (do not clear it).

Macro (Get Values)	Description
HPDI32_TX_OVERRUN_NO	This reflects that the condition does not exist.
HPDI32_TX_OVERRUN_YES	This reflects that the condition does exist.

Macro (Services)	Description
HPDI32_TX_OVERRUN__CLEAR(h)	This request that the condition be <i>cleared</i> .
HPDI32_TX_OVERRUN__GET(h,g)	This requests the current condition.
HPDI32_TX_OVERRUN__SET(h,s)	This requests a setting change.

### 6.8.9. Transmitter Parameter: Remote Throttle

The purpose of this parameter is to control and retrieve the board's Remote Throttling feature. If disabled, the default, then data flow is controlled locally rather than by the remote device. If enabled, then the receiving device must drive the cable's Rx Ready signal to control data transfer. The following tables describe the macros associated with this parameter.

**NOTE:** In the HPDI32 firmware, the Tx Remote Throttling feature operates in parallel with the Tx Start bit. If Remote Throttling is enabled and the Tx Start bit is set, then data will be transferred even if the Remote Throttling input from the remote device says to halt data transfer. This is likely to result in data loss.

**NOTE:** When the Remote Throttle parameter is enabled, the API will disable the Auto Start parameter. When the Auto Start parameter is enabled, the API will disable the Remote Throttle parameter. The setting of both parameters must be coordinated when using the `hpdi32_tx_config_t` structure, in which the Auto Start parameter appears first.

Macro (Parameter)	Description
HPDI32_TX_REMOTE_THROTTLE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_REMOTE_THROTTLE_DEFAULT	This is the default, which is the <i>disable</i> option.
HPDI32_TX_REMOTE_THROTTLE_DISABLE	This disabled the option.
HPDI32_TX_REMOTE_THROTTLE_ENABLE	This enables the option.

Macro (Services)	Description
HPDI32_TX_REMOTE_THROTTLE__DISABLE(h)	This request that the option be <i>disabled</i> .
HPDI32_TX_REMOTE_THROTTLE__ENABLE(h)	This request that the option be <i>enabled</i> .
HPDI32_TX_REMOTE_THROTTLE__GET(h,g)	This requests the current setting.
HPDI32_TX_REMOTE_THROTTLE__RESET(h)	This resets the setting.
HPDI32_TX_REMOTE_THROTTLE__SET(h,s)	This requests a setting change.

### 6.8.10. Transmitter Parameter: Remote Throttle State

The purpose of this read-only parameter is to retrieve the board's Remote Throttling state. The state is reported as active if the cable signal is configured for Flow Control, if Remote Throttling is enabled, and the signal is driven high. The state is otherwise reported as inactive. The following tables describe the macros associated with this parameter.

**NOTE:** In the HPDI32 firmware, the Tx Remote Throttling feature operates in parallel with the Tx Start bit. If Remote Throttling is enabled and the Tx Start bit is set, then data will be transferred even if the Remote Throttling input from the remote device says to halt data transfer. This is likely to result in data loss.

Macro (Parameter)	Description
HPDI32_TX_REMOTE_THROTTLE_STATE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_REMOTE_THROTTLE_STATE_ACTIVE	Remote Throttling is permitting data flow.
HPDI32_TX_REMOTE_THROTTLE_STATE_INACTIVE	Remote Throttling of data is inactive for one or more of the reasons described above.

Macro (Services)	Description
HPDI32_TX_REMOTE_THROTTLE_STATE_GET(h, g)	This requests the current state.

### 6.8.11. Transmitter Parameter: Tx State

The purpose of this read-only parameter is to retrieve the board's data transmission state. If active, then the data transmission process is active, either through local or remote control. The state is otherwise reported as inactive. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_STATE	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_STATE_ACTIVE	The data transmission process is active.
HPDI32_TX_STATE_INACTIVE	The data transmission process is inactive.

Macro (Services)	Description
HPDI32_TX_STATE_GET(h, g)	This requests the current state.

### 6.8.12. Transmitter Parameter: Status Valid Count

The purpose of this parameter is to control and retrieve the number of cable clock cycles that the Status Valid signal is initially high, during a frame. This parameter operates by accessing the board's Tx Status Valid Length Count Register. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_STATUS_VALID_COUNT	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_STATUS_VALID_COUNT_DEFAULT	This is the default, which disables the "on" period.
HPDI32_TX_STATUS_VALID_COUNT_DISABLE	This disables the "on" period, which sets the register to zero (0).
HPDI32_TX_STATUS_VALID_COUNT_MAX	This is the maximum period length. This is actually slightly less than can be written to the register as the true

	maximum conflicts with one of the special API macros.
--	---

Macro (Services)	Description
HPDI32_TX_STATUS_VALID_COUNT_DISABLE(h)	This requests that the “on” period be disabled.
HPDI32_TX_STATUS_VALID_COUNT_GET(h,g)	This requests the current setting.
HPDI32_TX_STATUS_VALID_COUNT_RESET(h)	This resets the setting.
HPDI32_TX_STATUS_VALID_COUNT_SET(h,s)	This requests a setting change.

### 6.8.13. Transmitter Parameter: Status Valid Mirror

The purpose of this parameter is to control and retrieve the board feature that forces the Line Valid signal high during the Status Valid high period (the Status Valid high state is mirrored onto the Line Valid signal). The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
HPDI32_TX_STATUS_VALID_MIRROR	This is the identifier for this parameter.

Macro (Values)	Description
HPDI32_TX_STATUS_VALID_MIRROR_DEFAULT	This is the default, which disables mirroring.
HPDI32_TX_STATUS_VALID_MIRROR_DISABLE	This disables mirroring.
HPDI32_TX_STATUS_VALID_MIRROR_ENABLE	This enables mirroring.

Macro (Services)	Description
HPDI32_TX_STATUS_VALID_MIRROR_DISABLE(h)	This requests that mirroring be disabled.
HPDI32_TX_STATUS_VALID_MIRROR_ENABLE(h)	This requests that mirroring be enabled.
HPDI32_TX_STATUS_VALID_MIRROR_GET(h,g)	This requests the current setting.
HPDI32_TX_STATUS_VALID_MIRROR_reset(h)	This resets the setting.
HPDI32_TX_STATUS_VALID_MIRROR_SET(h,s)	This requests a setting change.

## Document History

Revision	Description
March 21, 2008	Updated to release 6.0.0, which included porting the SDK to Linux (32-bit and 64-bit)?
January 15, 2007	Updated to release 5.0.2.
March 27, 2006	Minor corrections. Updated to SDK release 5.0.1.
August 18, 2005	Initial release.