

# **HPDI32**

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

**PCI-HPDI32/A  
PMC-HPDI32/A  
PCI64-HPDI32A/AL  
PMC64-HPDI32ALT**

## **Linux Device Driver User Manual**

**Manual Revision: December 20, 2011  
Driver Release Version 2.3.34.0**

**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 © 2002-2011, **General Standards Corporation**

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

**General Standards Corporation**

8302A Whitesburg Dr.

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>6</b>
1.1. Purpose.....	6
1.2. Acronyms.....	6
1.3. Definitions .....	6
1.4. Software Overview .....	6
1.5. Hardware Overview .....	6
1.6. Reference Material.....	7
<b>2. Installation.....</b>	<b>8</b>
2.1. CPU and Kernel Support.....	8
2.1.1. 32-bit Support Under 64-bit Environments .....	8
2.2. The /proc File System .....	8
2.3. File List .....	9
2.4. Directory Structure.....	9
2.5. Installation .....	9
2.6. Removal .....	10
2.7. Overall Make Script.....	10
<b>3. The Driver.....</b>	<b>11</b>
3.1. Build .....	11
3.2. Startup.....	11
3.2.1. Manual Driver Startup Procedures .....	11
3.2.2. Automatic Driver Startup Procedures.....	12
3.3. Verification .....	12
3.4. Version.....	12
3.5. Shutdown .....	13
<b>4. Document Source Code Examples.....</b>	<b>14</b>
4.1. Build .....	14
4.2. Library Use .....	14
<b>5. Sample Applications .....</b>	<b>15</b>
5.1. id - Identify Board.....	15
5.1.1. Build .....	15
5.1.2. Execute .....	15
5.2. regs - Register Access .....	16
5.2.1. Build .....	16
5.2.2. Execute .....	16
5.3. sbtest - Single Board Test .....	17
5.3.1. Build .....	17

5.3.2. Execute .....	17
5.4. signals - Command Signals .....	18
5.4.1. Build .....	18
5.4.2. Execute .....	18
5.5. txrate - Transmit Rate .....	19
5.5.1. Build .....	19
5.5.2. Execute .....	19
<b>6. Driver Interface.....</b>	<b>21</b>
6.1. Macros .....	21
6.1.1. IOCTL .....	21
6.1.2. Registers .....	21
6.2. Functions.....	22
6.2.1. close() .....	22
6.2.2. ioctl() .....	23
6.2.3. open().....	23
6.2.4. read() .....	24
6.2.5. write() .....	25
6.3. IOCTL Services .....	26
6.3.1. HPDI32_IOCTL_CABLE_CMD_MODE_n .....	27
6.3.2. HPDI32_IOCTL_CABLE_CMD_STATE_n.....	27
6.3.3. HPDI32_IOCTL_INITIALIZE .....	28
6.3.4. HPDI32_IOCTL_IRQ_CONFIG_EDGE.....	28
6.3.5. HPDI32_IOCTL_IRQ_CONFIG_HIGH.....	28
6.3.6. HPDI32_IOCTL_IRQ_ENABLE.....	28
6.3.7. HPDI32_IOCTL_QUERY .....	29
6.3.8. HPDI32_IOCTL_REG_MOD.....	31
6.3.9. HPDI32_IOCTL_REG_READ .....	31
6.3.10. HPDI32_IOCTL_REG_WRITE.....	32
6.3.11. HPDI32_IOCTL_RX_AUTO_START .....	32
6.3.12. HPDI32_IOCTL_RX_ENABLE .....	32
6.3.13. HPDI32_IOCTL_RX_FIFO_AE.....	33
6.3.14. HPDI32_IOCTL_RX_FIFO_AF.....	33
6.3.15. HPDI32_IOCTL_RX_FIFO_OVERRUN.....	33
6.3.16. HPDI32_IOCTL_RX_FIFO_RESET .....	34
6.3.17. HPDI32_IOCTL_RX_FIFO_STATUS .....	34
6.3.18. HPDI32_IOCTL_RX_FIFO_UNDERRUN.....	34
6.3.19. HPDI32_IOCTL_RX_IO_ABORT.....	35
6.3.20. HPDI32_IOCTL_RX_IO_DATA_SIZE.....	35
6.3.21. HPDI32_IOCTL_RX_IO_MODE.....	36
6.3.22. HPDI32_IOCTL_RX_IO_OVERRUN .....	36
6.3.23. HPDI32_IOCTL_RX_IO_PIO_THRESHOLD .....	37
6.3.24. HPDI32_IOCTL_RX_IO_TIMEOUT .....	37
6.3.25. HPDI32_IOCTL_RX_IO_UNDERRUN .....	37
6.3.26. HPDI32_IOCTL_RX_LINE_COUNT .....	37
6.3.27. HPDI32_IOCTL_RX_STATUS_COUNT .....	38
6.3.28. HPDI32_IOCTL_TRISTATE_TE_RE .....	38
6.3.29. HPDI32_IOCTL_TX_AUTO_START .....	38
6.3.30. HPDI32_IOCTL_TX_AUTO_STOP .....	39
6.3.31. HPDI32_IOCTL_TX_CLOCK_DIVIDER.....	39
6.3.32. HPDI32_IOCTL_TX_ENABLE .....	39
6.3.33. HPDI32_IOCTL_TX_FIFO_AE.....	40
6.3.34. HPDI32_IOCTL_TX_FIFO_AF .....	40

6.3.35. HPDI32_IOCTL_TX_FIFO_OVERRUN .....	40
6.3.36. HPDI32_IOCTL_TX_FIFO_RESET .....	41
6.3.37. HPDI32_IOCTL_TX_FIFO_STATUS .....	41
6.3.38. HPDI32_IOCTL_TX_FLOW_CONTROL.....	41
6.3.39. HPDI32_IOCTL_TX_IO_ABORT .....	42
6.3.40. HPDI32_IOCTL_TX_IO_DATA_SIZE .....	42
6.3.41. HPDI32_IOCTL_TX_IO_MODE.....	42
6.3.42. HPDI32_IOCTL_TX_IO_OVERRUN .....	43
6.3.43. HPDI32_IOCTL_TX_IO_PIO_THRESHOLD.....	43
6.3.44. HPDI32_IOCTL_TX_IO_TIMEOUT.....	44
6.3.45. HPDI32_IOCTL_TX_LINE_VAL_OFF_CNT .....	44
6.3.46. HPDI32_IOCTL_TX_LINE_VAL_ON_CNT .....	44
6.3.47. HPDI32_IOCTL_TX_REMOTE_THROTTLE.....	45
6.3.48. HPDI32_IOCTL_TX_STATUS_VAL_CNT .....	45
6.3.49. HPDI32_IOCTL_TX_STATUS_VAL_MIR .....	45
6.3.50. HPDI32_IOCTL_USER_JUMPERS .....	46
6.3.51. HPDI32_IOCTL_WAIT_CANCEL.....	46
6.3.52. HPDI32_IOCTL_WAIT_EVENT.....	47
6.3.53. HPDI32_IOCTL_WAIT_STATUS.....	49

<b>Document History .....</b>	<b>51</b>
-------------------------------	-----------

# 1. Introduction

This user manual applies to driver version 2.3.34.0.

## 1.1. Purpose

The purpose of this document is to describe the interface to the HPDI32 Linux device driver. This software provides the interface between "Application Software" and the HPDI32 board. The interface to this board is at the device level.

## 1.2. Acronyms

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

Acronyms	Description
DMA	Direct Memory Access
DMDMA	Demand Mode DMA
GSC	General Standards Corporation
PCI	Peripheral Component Interconnect
PMC	PCI Mezzanine Card

## 1.3. Definitions

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

Term	Definition
Application	Application means the user mode process, which runs in the user space with user mode privileges.
Driver	Driver means the kernel mode device driver, which runs in the kernel space with kernel mode privileges.
HPDI32	This is used as a general reference to any board supported by this driver.

## 1.4. Software Overview

The HPDI32 driver software executes under control of the Linux operating system and runs in Kernel Mode as a Kernel Mode device driver. The HPDI32 device driver is implemented as a standard dynamically loadable Linux device driver written in the C programming language. With the driver, user applications are able to open and close a device and, while open, perform read, write and I/O control operations.

## 1.5. 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 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 extends from sending or receive 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:** 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.

## 1.6. Reference Material

The following reference material may be of particular benefit in using the HPDI32 and this driver. 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. \*
- The *PCI9656 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc. \*

\* 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. Installation

### 2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel versions 2.6, 2.4 and 2.2 running on a PC system with one or more x86 processors. This release of the driver supports the below listed kernels.

Kernel	Distribution	x86	
		32-bit	64-bit
2.6.38	Red Hat Fedora Core 15	Yes	Yes
2.6.35	Red Hat Fedora Core 14	Yes	Yes
2.6.33	Red Hat Fedora Core 13	Yes	Yes
2.6.31	Red Hat Fedora Core 12	Yes	Yes
2.6.29	Red Hat Fedora Core 11	Yes	Yes
2.6.27	Red Hat Fedora Core 10	Yes	Yes
2.6.25	Red Hat Fedora Core 9	Yes	Yes
2.6.23	Red Hat Fedora Core 8	Yes	Yes
2.6.21	Red Hat Fedora Core 7	Yes	Yes
2.6.18	Red Hat Fedora Core 6	Yes	Yes
2.6.15	Red Hat Fedora Core 5	Yes	Yes
2.6.11	Red Hat Fedora Core 4	Yes	Yes
2.6.9	Red Hat Fedora Core 3	Yes	Yes
2.4.21	Red Hat Enterprise Linux Workstation Release 3	Yes	
2.4.7	Red Hat Linux 7.2	Yes	
2.2.14	Red Hat Linux 6.2	Yes	

**NOTE:** The driver will have to be built before being used as it is shipped in source form only.

**NOTE:** The driver has not been tested with a non-versioned kernel.

**NOTE:** The driver has not been tested for SMP operation.

#### 2.1.1. 32-bit Support Under 64-bit Environments

This driver supports 32-bit applications under 64-bit environments. The availability of this feature in the kernel depends on a 64-bit kernel being configured to support 32-bit application compatibility. Additionally, 2.6 kernels prior to 2.6.11 implemented 32-bit compatibility in a way that resulted in some drivers not being able to take advantage of the feature. (In these kernels a driver's IOCTL command codes must be globally unique. Beginning with 2.6.11 this requirement has been lifted.) If the driver is not able to provide 32-bit support under a 64-bit kernel, the "32-bit support" field in the `/proc/hpdi32` file will be "no".

### 2.2. The /proc File System

While the driver is running, the text file `/proc/hpdi32` can be read to obtain information about the driver. Each file entry includes an entry name followed immediately by a colon, a space character, and the entry value. Below is an example of what appears in the file, followed by descriptions of each entry.



```

version: 2.3.34
built: Dec 20 2011, 16:15:18
32-bit support: yes (native)
boards: 1
models: HPDI32
ids: 0x3

```

Entry	Description
version	This gives the driver version number in the form x.x.x.
built	This gives the driver build date and time as a string. It is given in the C form of <code>printf("%s, %s", __DATE__, __TIME__)</code> .
32-bit support	This reports the driver's support for 32-bit applications. This will be either "yes" or "no" for 64-bit driver builds and "yes (native)" for 32-bit builds.
boards	This identifies the total number of boards the driver detected.
models	This gives a comma separated list of the basic model number for each board the driver detected.
ids	This gives a comma separated list of the board identifiers. The identifier is the user jumper value from the Board Status Register, if supported by firmware, and "none" if the board doesn't support the jumpers. The list will contain "boards" number of entries. The order corresponds to the device node indexes and minor numbers as given in the <code>/dev</code> directory. The options are "0x0" through "0x3" and "none".

## 2.3. File List

This release consists of the below listed files. The archive is described in detail in following subsections.

File	Description
hpdi32.linux.tar.gz	This archive contains the driver, the samples and all related sources.
hpdi32_linux_um.pdf	This is a PDF version of this user manual, which is included in the archive.

## 2.4. Directory Structure

The following table describes the directory structure utilized by the installed files. During installation the directory structure is created and populated with the respective files.

Directory	Content
hpdi32	This is the source root directory. The user manual and overall make script are placed here.
hpdi32\docsrc	This directory contains the Document Source Code Examples (section 4, page 14).
hpdi32\driver	This directory contains the driver and its sources (section 3, page 11).
hpdi32\id	This directory contains the Identification sample application (section 0, page 15).
hpdi32\regs	This directory contains the Register Access sample application (section 5.1.2, page 15).
hpdi32\sbtest	This directory contains the Single Board Test application (section 5.4, page 18).
hpdi32\signals	This directory contains the Command Signals sample application (section 5.2.2, page 16).
hpdi32\txrate	This directory contains the Command Signals sample application (section 5.5, page 19).
hpdi32\utils	This directory contains utility sources used by the sample applications.

## 2.5. Installation

Install the driver and its related files following the below listed steps. This includes the device driver, the documentation source code, and the sample applications.

1. Create and change to the directory where the files are to be installed, such as `/usr/src/linux/drivers`. (The path name may vary among distributions and kernel versions.)

2. Copy the archive file `hpdi32.linux.tar.gz` into the current directory.
3. Issue the following command to decompress and extract the files from the provided archive. This creates the directory `hpdi32` in the current directory, and then copies all of the archive's files into this new directory.

```
tar -xzvf hpdi32.linux.tar.gz
```

## 2.6. Removal

Follow the below steps to remove the driver and its related files. This includes the device driver, the documentation source code, and the sample applications.

1. Shutdown the driver as described in section 3.5 on page 13.
2. Change to the directory where the driver archive was installed, which may have been `/usr/src/linux/drivers`. (The path name may vary among distributions and kernel versions.)
3. Issue the below command to remove the driver archive and all of the installed driver files.

```
rm -rf hpdi32.linux.tar.gz hpdi32
```

4. Issue the below command to remove all of the installed device nodes.

```
rm -f /dev/hpdi32*
```

5. If the automated startup procedure was adopted (see section 3.2.2, page 12), then edit the system startup script `rc.local` and remove the line that invokes the HPDI32's start script. The file `rc.local` should be located in the `/etc/rc.d` directory.

## 2.7. Overall Make Script

An overall make script is included in the root installation directory. Executing this script will perform a make for all build targets included in the release, and it will also load the driver. The script is named `make_all`. Follow the below steps to perform an overall make and to load the driver.

1. Change to the driver's directory, which may be `/usr/src/linux/drivers/hpdi32`.
2. Issue the following command to make all archive targets and to load the driver.

```
./make_all
```

### 3. The Driver

The driver and its related files are contained in the archive file `hpdi32.linux.tar.gz`. The driver's files are summarized in the table below.

File	Description
<code>driver/*.c</code>	These are driver source files.
<code>driver/*.h</code>	These are driver header files.
<code>driver/hpdi32.h</code>	This is the main driver header file. This header should be included by HPDI32 applications.
<code>driver/start</code>	This is a shell script to install the driver executable and create the device nodes.
<code>driver/Makefile</code>	This is the driver make file.

#### 3.1. Build

**NOTE:** Building the driver requires installation of the kernel headers.

Follow the below steps to build the driver.

1. Change to the directory where the driver and its sources are installed, which may be `/usr/src/linux/drivers/hpdi32/driver`.
2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the driver by issuing the below command.

```
make all
```

**NOTE:** Due to the differences between the many Linux distributions some build errors may occur. These errors may include system header location differences, which should be easily corrected.

#### 3.2. Startup

**NOTE:** The driver will have to be built before being used as it is provided in source form only.

The startup script used in this procedure is designed to insure that the driver module in the install directory is the module that is loaded. This is accomplished by making sure that an already loaded module is first unloaded before attempting to load the module from the disk drive. In addition, the script also deletes and recreates the device nodes. This is done to insure that the device nodes in use have the same major number as assigned dynamically to the driver by the kernel, and so that the number of device nodes correspond to the number of boards identified by the driver.

##### 3.2.1. Manual Driver Startup Procedures

Start the driver manually by following the below listed steps.

1. Login as root user, as some of the steps require root privileges.
2. Change to the directory where the driver sources are installed, which may be `/usr/src/linux/drivers/hpdi32/driver`.

3. Install the driver module and create the device nodes by executing the below command. If any errors are encountered then an appropriate error message will be displayed.

```
./start
```

**NOTE:** This script must be executed each time the host is rebooted.

**NOTE:** The HPDI32 device node major number is assigned dynamically by the kernel. The minor numbers and the device node suffix numbers are index numbers beginning with zero, and increase by one for each additional board installed.

4. Verify that the device driver module has been loaded by issuing the below command and examining the output. The module name `hpdi32` should be included in the output.

```
lsmod
```

5. Verify that the device nodes have been created by issuing the below command and examining the output. The output should include one node for each installed board.

```
ls -l /dev/hpdi32*
```

### 3.2.2. Automatic Driver Startup Procedures

Start the driver automatically with each system reboot by following the below listed steps.

1. Locate and edit the system startup script `rc.local`, which should be in the `/etc/rc.d` directory. Modify the file by adding the below line so that it is executed with every reboot. The example is based on the driver being installed in `/usr/src/linux/drivers`, though it may have been installed elsewhere.

```
/usr/src/linux/drivers/hpdi32/driver/start
```

2. Load the driver and create the required device nodes by rebooting the system.
3. Verify that the driver is loaded and that the device nodes have been created. Do this by following the verification steps given in the manual startup procedures.

### 3.3. Verification

Follow the below steps to verify that the driver has been properly installed and started.

1. Verify that the file `/proc/hpdi32` is present. If the file is present then the driver is loaded and running. Verify the file's presence by viewing its content with the below command.

```
cat /proc/hpdi32
```

### 3.4. Version

The driver version number can be obtained in a variety of ways. It is reported by the driver both when the driver is loaded and when it is unloaded (depending on kernel configuration options, this may be visible only in places such as `/var/log/messages`). It is reported in the text file `/proc/hpdi32` while the driver is loaded and running.

### 3.5. Shutdown

Shutdown the driver following the below listed steps.

1. Login as root user, as some of the steps require root privileges.
2. If the driver is currently loaded then issue the below command to unload the driver.  
  
`rmmod hpdi32`
3. Verify that the driver module has been unloaded by issuing the below command. The module name `hpdi32` should not be in the listed output.

`lsmod`

## 4. Document Source Code Examples

The archive file `hpdi32.linux.tar.gz` contains all of the source code examples included in this document. In addition, the code is built into a statically linkable library usable with HPDI32 console applications. The library and sources are delivered undocumented and unsupported. The purpose of these files is to verify that the documentation samples compile and to provide a library of working sample code to assist in a user's learning curve and application development effort. These files are located in the `docsrc` subdirectory under the HPDI32 root directory.

File	Description
<code>docsrc/*.c</code>	These are the C source files.
<code>docsrc/hpdi32_dsl.h</code>	This is the library header file.
<code>docsrc/makefile</code>	This is the library make file.
<code>docsrc/makefile.dep</code>	This is an automatically generated make dependency file.

### 4.1. Build

Follow the below steps to compile the example files and build the library.

1. Change to the directory where the documentation sources are installed, which may be `/usr/src/linux/drivers/hpdi32/docsrc`.
2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Compile the sample files and build the library by issuing the below command.

```
make all
```

### 4.2. Library Use

The library is used both at application compile time and at application link time. Compile time use has two requirements. First, include the header file `hpdi32_dsl.h` in each module referencing a library component. Second, expand the include file search path to search the directory where the library header is located, which may be `/usr/src/linux/drivers/hpdi32/docsrc`. Link time use also has two requirements. First, include the static library `hpdi32_dsl.a` in the list of files to be linked into the application. Second, expand the library file search path to search the directory where the library is located, which may be `/usr/src/linux/drivers/hpdi32/docsrc`.

## 5. Sample Applications

**NOTE:** The sample applications are unsupported and are provided without documentation.

### 5.1. id - Identify Board

This sample console application provides a command line driven Linux application that provides detailed board identification information. This can be used with tech support to help identify as much technical information about the board as possible from software. The application's sources are summarized in the below table.

File	Description
id/*.c	These are the application's source files.
id/main.h	This is the application's header file.
id/makefile	This is the application make file.
id/makefile.dep	This is an automatically generated make dependency file.
docsrc/*	These are utility sources used by the application.
utils/*	These are utility sources used by the application.

#### 5.1.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application sources are installed (.../id).
2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the application by issuing the below command.

```
make all
```

#### 5.1.2. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application sources are installed (.../id).
2. Start the sample application by issuing the command given below. Once started the application will automatically output identification information. A single iteration should take less than a second. The command line arguments are described in the table below.

```
./id <index>
```

Argument	Description
index	This is the zero based index of the board to access.

## 5.2. regs - Register Access

This sample console application provides a menu based command line Linux application that permits interactive access to the board's registers, including write access to the GSC specific registers. The application's sources are summarized in the below table.

File	Description
regs/*.c	These are the application's source files.
regs/main.h	This is the application's header file.
regs/makefile	This is the application make file.
regs/makefile.dep	This is an automatically generated make dependency file.
docsrc/*	These are utility sources used by the application.
utils/*	These are utility sources used by the application.

### 5.2.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application sources are installed (.../regs).
2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the application by issuing the below command.

```
make all
```

### 5.2.2. Execute

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application sources are installed (.../regs).
2. Start the sample application by issuing the command given below. The command line argument is described in the table below.

```
./regs <index>
```

Argument	Description
index	This is the zero based index of the board to access.

3. Select the desired options according to the menus presented. Select the main menu exit option when finished.



### 5.3. sbtest - Single Board Test

This sample console application provides a command line driven Linux application that tests the functionality of the driver and a specified board. The application's sources are summarized in the below table.

File	Description
sbtest/*.c	These are the application's source files.
sbtest/main.h	This is the application's header file.
sbtest/makefile	This is the application make file.
sbtest/makefile.dep	This is an automatically generated make dependency file.
docsrc/*	These are utility sources used by the application.
utils/*	These are utility sources used by the application.

#### 5.3.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application sources are installed (.../sbtest).

2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the application by issuing the below command.

```
make all
```

#### 5.3.2. Execute

**NOTE:** This application should be run with no cable attached.

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application sources are installed (.../sbtest).

2. Start the sample application by issuing the command given below. Once started the application will automatically performs a series of test operations. A single iteration should take less than about 90 seconds to complete. The command line arguments are described in the table below.

```
./sbtest <-c> <-C> <-m#> <-n#> <index>
```

Argument	Description
-c	Repeat the operation until an error is encountered.
-C	Repeat the operation, but continue even if errors are encountered.
-m#	When repeating the operation, stop after “#” minutes, where “#” is a decimal number.
-n#	When repeating the operation, stop after “#” iterations, where “#” is a decimal number.
index	This is the zero based index of the board to access.

## 5.4. signals - Command Signals

This sample console application provides a command line driven Linux application that configures the board to drive the cable command signals for GPIO and/or Transmission operations. The purpose of this application is to permit verification of the signal's operation and presence, and to more easily permit test equipment to be configured to capture the signal. The application's sources are summarized in the below table.

File	Description
signals/*.c	These are the application's source files.
signals/main.h	This is the application's header file.
signals/makefile	This is the application make file.
signals/makefile.dep	This is an automatically generated make dependency file.
docsrc/*	These are utility sources used by the application.
utils/*	These are utility sources used by the application.

### 5.4.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application sources are installed (.../signals).
2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the application by issuing the below command.

```
make all
```

### 5.4.2. Execute

**NOTE:** This application should be run with no cable attached.

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application sources are installed (.../signals).
2. Start the sample application by issuing the command given below. Once started the application will generate a series of cable sync pulses. A single iteration should take about 45 seconds to complete with the default arguments. The command line arguments are described in the table below.

```
./signals <-all> <-c> <-C> <-d#> <-gpio> <-m#> <-n#> <-tx> <index>
```

Argument	Description
-all	Drive the cable command signals in all modes supported by the application.
-c	Repeat the operation until an error is encountered.
-C	Repeat the operation, but continue even if errors are encountered.
-d#	Program the Tx Clock Divider to the value given as "#", which is a decimal number
-gpio	Drive the cable command signals in their GPIO modes only.
-m#	When repeating the operation, stop after "#" minutes, where "#" is a decimal number.
-n#	When repeating the operation, stop after "#" iterations, where "#" is a decimal number.
-tx	Drive the cable command signals in their data transmission modes only.
index	This is the zero based index of the board to access.

## 5.5. txrate - Transmit Rate

This sample console application provides a command line driven Linux application that configures the board for maximum transmission rate, then transmits data for a period of time. The purpose of this application is to measure and report the data transfer rate. The application's sources are summarized in the below table.

File	Description
txrate/*.c	These are the application's source files.
txrate/main.h	This is the application's header file.
txrate/makefile	This is the application make file.
txrate/makefile.dep	This is an automatically generated make dependency file.
docsrc/*	These are utility sources used by the application.
utils/*	These are utility sources used by the application.

### 5.5.1. Build

Follow the below steps to build the sample application.

1. Change to the directory where the sample application sources are installed (.../txrate).
2. Remove all existing build targets by issuing the below command.

```
make clean
```

3. Build the application by issuing the below command.

```
make all
```

### 5.5.2. Execute

**NOTE:** This application should be run with no cable attached.

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application sources are installed (.../txrate).
2. Start the sample application by issuing the command given below. Once started the application will transmit a data for a period of time then report the results. A single iteration should less than 15 seconds with the default arguments. The command line arguments are described in the table below.

```
./txrate <-c> <-C> <-dma> <-dmdma> <-m#> <-n#> <-pio> <-s#>  
<-Sae> <-Saf> <-Sw> <index>
```

Argument	Description
-c	Repeat the operation until an error is encountered.
-C	Repeat the operation, but continue even if errors are encountered.
-dma	Transfer data to the board using DMA.
-dmdma	Transfer data to the board using Demand Mode DMA.
-m#	When repeating the operation, stop after “#” minutes, where “#” is a decimal number.
-n#	When repeating the operation, stop after “#” iterations, where “#” is a decimal number.
-pio	Transfer data to the board using PIO, which is repetitive register accesses.
-s	Transmit data for this number of seconds (1-600).
-Sae	Perform throughput rate measurements while adjusting the Tx Almost Empty setting to determine which setting produces the highest rate. This takes many, many hours to

	complete.
-Saf	Perform throughput rate measurements while adjusting the Tx Almost Full setting to determine which setting produces the highest rate. This takes many, many hours to complete.
-Sw	Perform throughput rate measurements while adjusting the write buffer size to determine which size produces the highest rate. This takes many, many hours to complete.
index	This is the zero based index of the board to access.

## 6. Driver Interface

The HPDI32 driver conforms to the device driver standards required by the Linux Operating System and contains the standard driver entry points. The device driver provides a standard driver interface to HPDI32 boards for Linux applications. The interface includes various macros, data types and functions, all of which are described in the following paragraphs. The HPDI32 specific portion of the driver interface is defined in the header file `hpdi32.h`, portions of which are described in this section. The header defines numerous items in addition to those described here.

**NOTE:** Contact General Standards Corporation if additional driver functionality is required.

### 6.1. Macros

The driver interface includes the following macros, which are defined in `hpdi32.h`.

#### 6.1.1. IOCTL

The IOCTL macros are documented in section 6.3 beginning on page 26.

#### 6.1.2. Registers

The following gives the complete set of HPDI32 registers.

##### 6.1.2.1. GSC Registers

The following tables give the complete set of GSC specific HPDI32 registers. Please note that the set of registers supported by any given board may vary according to model and firmware version. For the set of supported registers and detailed definitions of these registers please refer to the appropriate *HPDI32 User Manual*.

Macros	Description
HPDI32_GSC_BCR	Board Control Register (BCR)
HPDI32_GSC_BSR	Board Status Register (BSR)
HPDI32_GSC_FDR	FIFO Data Register (FDR)
HPDI32_GSC_FRR	Firmware Revision Register (FRR)
HPDI32_GSC_FSR	Feature Set Register (FSR)
HPDI32_GSC_ICR	Interrupt Control Register (ICR)
HPDI32_GSC_IELR	Interrupt Edge/Level Register (IELR)
HPDI32_GSC_IHLR	Interrupt High/Low Register (IHLR)
HPDI32_GSC_ISR	Interrupt Status Register (ISR)
HPDI32_GSC_RAR	Rx Almost Register (RAR)
HPDI32_GSC_RFSR	Rx FIFO Size Register (RFSR)
HPDI32_GSC_RFWR	Rx FIFO Words Register (RFWR)
HPDI32_GSC_RLCR	Rx Line Count Register (RLCR)
HPDI32_GSC_RSCR	Rx Status Count Register (RSCR)
HPDI32_GSC_TAR	Tx Almost Register (TAR)
HPDI32_GSC_TCDR	Tx CLock Divider Register (TCDR)
HPDI32_GSC_TFSR	Tx FIFO Size Register (TFSR)
HPDI32_GSC_TFWR	Tx FIFO Words Register (TFWR)
HPDI32_GSC_TLILCR	Tx Line Invalid Length Counter Register (TLILCR)
HPDI32_GSC_TLVLCR	Tx Line Valid Length Counter Register (TLVLCR)
HPDI32_GSC_TSVLCR	Tx Status Valid Length Counter Register (TSVLCR)

### 6.1.2.2. PCI Configuration Registers

Access to the PCI registers is seldom required so these registers are not listed here. For the complete list of the PCI register identifiers refer to the driver header file `gsc_pci9080.h` or `gsc_pci9656.h`, which are automatically included via `hpdi32.h`.

### 6.1.2.3. PLX Feature Set Registers

Access to the PLX registers is seldom required so these registers are not listed here. For the complete list of the PLX register identifiers refer to the driver header files `gsc_pci9080.h` and `gsc_pci9656.h`, which are automatically included via `hpdi32.h`.

## 6.2. Functions

The driver interface includes the following functions.

### 6.2.1. `close()`

This function is the entry point to close a connection to an open HPDI32 board.

Prototype

```
int close(int fd);
```

Argument	Description
<code>fd</code>	This is the file descriptor of the device to be closed.

Return Value	Description
<code>-1</code>	An error occurred. Consult <code>errno</code> .
<code>0</code>	The operation succeeded.

Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#include "hpdi32_dsl.h"

int hpdi32_dsl_close(int fd)
{
    int errs;
    int status;

    status = close(fd);

    if (status == -1)
        printf("ERROR: close() failure, errno = %d\n", errno);

    errs = (status == -1) ? 1 : 0;
    return(errs);
}
```

### 6.2.2. ioctl()

This function is the entry point to performing setup and control operations on an HPDI32 board. This function should only be called after a successful open of the respective device. The specific operation performed varies according to the `request` argument. The `request` argument also governs the use and interpretation of any additional arguments. The set of supported IOCTL services is defined in section 6.3 beginning on page 26.

#### Prototype

```
int ioctl(int fd, int request, ...);
```

Argument	Description
<code>fd</code>	This is the file descriptor of the device to access.
<code>request</code>	This specifies the desired operation to be performed.
<code>...</code>	This is any additional arguments. If <code>request</code> does not call for any additional arguments, then any additional arguments provided are ignored. The HPDI32 IOCTL services use at most one argument.

Return Value	Description
-1	An error occurred. Consult <code>errno</code> .
0	The operation succeeded.

#### Example

```
#include <errno.h>
#include <stdio.h>
#include <sys/ioctl.h>

#include "hpdi32_dsl.h"

int hpdi32_dsl_ioctl(int fd, int request, void *arg)
{
    int errs;
    int status;

    status = ioctl(fd, request, arg);

    if (status == -1)
        printf("ERROR: ioctl() failure, errno = %d\n", errno);

    errs = (status == -1) ? 1 : 0;
    return(errs);
}
```

### 6.2.3. open()

This function is the entry point to open a connection to an HPDI32 board. The pathname to an HPDI32 device node is `/dev/hpdi32n`, where the trailing “*n*” is the zero based index of the board to access.

## Prototype

```
int open(const char* pathname, int flags);
```

Argument	Description
pathname	This is the name of the device to open.
flags	This is the desired read/write access. Use O_RDWR.

**NOTE:** Another form of the `open()` function has a mode argument. This form is not displayed here as the mode argument is ignored when opening an existing file/device.

Return Value	Description
-1	An error occurred. Consult <code>errno</code> .
else	A valid file descriptor.

## Example

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>

#include "hpdi32_dsl.h"

int hpdi32_dsl_open(unsigned int board)
{
    int    fd;
    char   name[80];

    sprintf(name, HPDI32_DEV_BASE_NAME "%u", board);
    fd = open(name, O_RDWR);

    if (fd == -1)
    {
        printf("ERROR: open() failure on %s, errno = %d\n",
               name,
               errno);
    }

    return(fd);
}
```

### 6.2.4. read()

This function is the entry point to reading data from an open HPDI32. This function should only be called after a successful open of the respective device. The function reads up to `count` bytes from the board. The return value is the number of bytes actually read.

## Prototype

```
int read(int fd, void *buf, size_t count);
```

Argument	Description
fd	This is the file descriptor of the device to access.



buf	The data read will be put here.
count	This is the desired number of bytes to read. This must be a multiple of the configured data size (HPDI32_IOCTL_RX_IO_DATA_SIZE, section 6.3.20, page 35).

Return Value	Description
-1	An error occurred. Consult <code>errno</code> .
0 to count	The operation succeeded. For blocking I/O a return value less than <code>count</code> indicates that the request timed out. For non-blocking I/O a return value less than <code>count</code> indicates that the operation ended prematurely due to the Rx FIFO becoming empty during the request.

### Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#include "hpdi32_dsl.h"

int hpdi32_dsl_read(int fd, void* buf, size_t samples)
{
    size_t  bytes;
    __s32   size;
    int     status;

    size    = -1;
    status  = hpdi32_dsl_ioctl(fd,
                               HPDI32_IOCTL_RX_IO_DATA_SIZE,
                               &size);

    if (status == 0)
    {
        bytes  = samples * (size / 8);
        status = read(fd, buf, bytes);

        if (status == -1)
            printf("ERROR: read() failure, errno = %d\n", errno);
        else
            status /= (size / 8);
    }

    return(status);
}
```

### 6.2.5. write()

This function is the entry point to writing data to an open HPDI32. This function should only be called after a successful open of the respective device. The function writes up to `count` bytes to the board. The return value is the number of bytes actually written.

#### Prototype

```
int write(int fd, const void *buf, size_t count);
```

Argument	Description
fd	This is the file descriptor of the device to access.
buf	The data written comes from here. If this pointer is NULL, then the data written will be that which is already present in the write transfer buffer. If the write transfer buffer is mapped using the <code>mmap()</code> feature, then this must be NULL.
count	This is the desired number of bytes to write. This must be a multiple of the configured data size ( <code>HPDI32_IOCTL_TX_IO_DATA_SIZE</code> , section 6.3.40, page 42).

Return Value	Description
-1	An error occurred. Consult <code>errno</code> .
0 to count	The operation succeeded. For blocking I/O a return value less than <code>count</code> indicates that the request timed out. For non-blocking I/O a return value less than <code>count</code> indicates that the operation ended prematurely due to the Tx FIFO becoming full during the request.

### Example

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#include "hpdi32_dsl.h"

int hpdi32_dsl_write(int fd, const void* buf, size_t samples)
{
    size_t  bytes;
    __s32   size;
    int     status;

    size    = -1;
    status  = hpdi32_dsl_ioctl(fd,
                               HPDI32_IOCTL_TX_IO_DATA_SIZE,
                               &size);

    if (status == 0)
    {
        bytes  = samples * (size / 8);
        status = write(fd, buf, bytes);

        if (status == -1)
            printf("ERROR: write() failure, errno = %d\n", errno);
        else
            status /= (size / 8);
    }

    return(status);
}
```

## 6.3. IOCTL Services

The HPDI32 driver implements the following IOCTL services. Each service is described along with the applicable `ioctl()` function arguments. In the definitions given the optional argument is identified as `arg`. Unless otherwise

stated the return value definitions are those defined for the `ioctl()` function call and any error codes are accessed via `errno`.

### 6.3.1. HPDI32\_IOCTL\_CABLE\_CMD\_MODE\_ *n*

These services configure the operating modes for Cable Command signals zero to six. The command signals and corresponding service macros are given in the below table.

Cable Command	Service Macro
0	HPDI32_IOCTL_CABLE_CMD_MODE_0
1	HPDI32_IOCTL_CABLE_CMD_MODE_1
2	HPDI32_IOCTL_CABLE_CMD_MODE_2
3	HPDI32_IOCTL_CABLE_CMD_MODE_3
4	HPDI32_IOCTL_CABLE_CMD_MODE_4
5	HPDI32_IOCTL_CABLE_CMD_MODE_5
6	HPDI32_IOCTL_CABLE_CMD_MODE_6

#### Usage

<code>ioctl()</code> Argument	Description
<code>request</code>	HPDI32_IOCTL_CABLE_CMD_MODE_ <i>n</i>
<code>arg</code>	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
HPDI32_CABLE_CMD_MODE_FC	This refers to the Flow Control option.
HPDI32_CABLE_CMD_MODE_IN	This refers to the GPIO Input option.
HPDI32_CABLE_CMD_MODE_OUT_LO	This refers to the GPIO Output Low option.
HPDI32_CABLE_CMD_MODE_OUT_HI	This refers to the GPIO Output High option.

### 6.3.2. HPDI32\_IOCTL\_CABLE\_CMD\_STATE\_ *n*

These services retrieve the current signal state for Cable Command signals zero to six, irrespective of their current configuration. The command signals and corresponding service macros are given in the below table.

Cable Command	Service Macro
0	HPDI32_IOCTL_CABLE_CMD_STATE_0
1	HPDI32_IOCTL_CABLE_CMD_STATE_1
2	HPDI32_IOCTL_CABLE_CMD_STATE_2
3	HPDI32_IOCTL_CABLE_CMD_STATE_3
4	HPDI32_IOCTL_CABLE_CMD_STATE_4
5	HPDI32_IOCTL_CABLE_CMD_STATE_5
6	HPDI32_IOCTL_CABLE_CMD_STATE_6

#### Usage

<code>ioctl()</code> Argument	Description
<code>request</code>	HPDI32_IOCTL_CABLE_CMD_STATE_ <i>n</i>
<code>arg</code>	__s32*

The current state is reported as one of the following values.

Value	Description
HPDI32_CABLE_CMD_STATE_0	The signal is at logic level zero.
HPDI32_CABLE_CMD_STATE_1	The signal is at logic level one.

### 6.3.3. HPDI32\_IOCTL\_INITIALIZE

This service returns all driver interface settings for the board to the state they were in when the board was first opened. This includes both hardware based settings and software based settings.

Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_INITIALIZE
arg	Not used.

### 6.3.4. HPDI32\_IOCTL\_IRQ\_CONFIG\_EDGE

This service configures firmware interrupts to be either edge triggered or level triggered. If a bit is set, then the interrupt is edge triggered. If a bit is clear, then the interrupt is level triggered.

Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_IRQ_CONFIG_EDGE
arg	__s32*

Valid argument values include any bitwise combination of the bits defined for the HPDI32\_IOCTL\_IRQ\_ENABLE service (section 6.3.5, page 28), or -1 to retrieve the current configurations.

### 6.3.5. HPDI32\_IOCTL\_IRQ\_CONFIG\_HIGH

This service configures firmware interrupts to be either high or low triggered. High refers to either a high level or a rising edge, depending on the interrupt's edge/level configuration. Low refers to either a low level or a falling edge, depending on the interrupt's edge/level configuration. If a bit is set, then the interrupt is high triggered. If a bit is clear, then the interrupt is low triggered.

Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_IRQ_CONFIG_HIGH
arg	__s32*

Valid argument values include any bitwise combination of the bits defined for the HPDI32\_IOCTL\_IRQ\_ENABLE service (section 6.3.5, page 28), or -1 to retrieve the current configurations.

### 6.3.6. HPDI32\_IOCTL\_IRQ\_ENABLE

This service enables a specified set of firmware interrupts. If a bit is set, then the interrupt is enabled. If a bit is clear, then the interrupt is disabled. When an interrupt is generated it is serviced and disabled by the driver.

Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_IRQ_ENABLE

arg	__s32*
-----	--------

Valid argument values include any bitwise combination of the following bits, or -1 to retrieve the current configuration.

Value	Description
HPDI32_IRQ_CC0_FV_E_GPIO6	This refers to the Cable Command 0 signal, whose default configuration is to trigger on a falling edge. This signal may be configured as Frame Valid or GPIO 6.
HPDI32_IRQ_CC0_FV_S_GPIO6	This refers to the Cable Command 0 signal, whose default configuration is to trigger on a rising edge. This signal may be configured as Frame Valid or GPIO 6.
HPDI32_IRQ_CC1_LV_GPIO0	This refers to the Cable Command 1 signal. This signal may be configured as Line Valid or GPIO 0.
HPDI32_IRQ_CC2_SV_GPIO1	This refers to the Cable Command 2 signal. This signal may be configured as Status Valid or GPIO 1.
HPDI32_IRQ_CC3_RR_GPIO2	This refers to the Cable Command 3 signal. This signal may be configured as Rx Ready or GPIO 2.
HPDI32_IRQ_CC4_TR_GPIO3	This refers to the Cable Command 4 signal. This signal may be configured as Tx Data Ready or GPIO 3.
HPDI32_IRQ_CC5_TE_GPIO4	This refers to the Cable Command 4 signal. This signal may be configured as Tx Enabled or GPIO 4.
HPDI32_IRQ_CC6_RE_GPIO5	This refers to the Cable Command 5 signal. This signal may be configured as Rx Enabled or GPIO 5.
HPDI32_IRQ_RX_FIFO_AE	This refers to the Rx FIFO's Almost Empty status.
HPDI32_IRQ_RX_FIFO_AF	This refers to the Rx FIFO's Almost Full status.
HPDI32_IRQ_RX_FIFO_EMPTY	This refers to the Rx FIFO's empty status.
HPDI32_IRQ_RX_FIFO_FULL	This refers to the Rx FIFO's full status.
HPDI32_IRQ_TX_FIFO_AE	This refers to the Tx FIFO's Almost Empty status.
HPDI32_IRQ_TX_FIFO_AF	This refers to the Tx FIFO's Almost Full status.
HPDI32_IRQ_TX_FIFO_EMPTY	This refers to the Tx FIFO's empty status.
HPDI32_IRQ_TX_FIFO_FULL	This refers to the Tx FIFO's full status.

### 6.3.7. HPDI32\_IOCTL\_QUERY

This service queries the driver for various pieces of information about the board and the driver.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_QUERY
arg	__s32*

Valid argument values are as follows.

Value	Description
HPDI32_QUERY_BSR_D18_XCVR	This indicates if BSR bit D18 reports the transceiver selection.
HPDI32_QUERY_BUS_WIDTH	This returns the board's PCI interface bus width in bits, which is either 32 or 64.
HPDI32_QUERY_CLOCK_MAX	This returns the maximum cable interface clock rate in hertz, which is either 25,000,000 or 50,000,000.
HPDI32_QUERY_COUNT	This returns the number of query options supported by the IOCTL service.

HPDI32_QUERY_DEVICE_TYPE	This returns the identifier value for the board's type. This should be GSC_DEV_TYPE_HPDI32.
HPDI32_QUERY_DMDMA_1	This indicates if Demand Mode DMA is supported on DMA channel 1.
HPDI32_QUERY_FEATURE_SET_REG	This indicates if the firmware supports the Feature Set Register.
HPDI32_QUERY_FIFO_SIZE_REGS	This indicates if the firmware supports the Rx FIFO Size Register and the Tx FIFO Size Register.
HPDI32_QUERY_FIFO_SIZE_RX	This indicates the depth of the Rx FIFO in 32-bit words. A value of zero indicates the FIFO size is not known.
HPDI32_QUERY_FIFO_SIZE_TX	This indicates the depth of the Tx FIFO in 32-bit words. A value of zero indicates the FIFO size is not known.
HPDI32_QUERY_FIFO_WORDS_REGS	This indicates if the firmware supports the Rx FIFO Words Register and the Tx FIFO Words Register.
HPDI32_QUERY_FORM_FACTOR	This indicates the board's native form factor. The options are listed below.
HPDI32_QUERY_GPIO_0_5	This indicates if the GPIO 0 to 5 configuration options are available for Cable Command signals 1 through 6, respectively.
HPDI32_QUERY_GPIO_6	This indicates if the GPIO 6 configuration option is available for Cable Command signal 0.
HPDI32_QUERY_IRQ_CONFIG_REGS	This indicates if the firmware supports the Interrupt Edge/Level Register and the Interrupt High/Low Register.
HPDI32_QUERY_OVER_UNDER_RUN	This indicates if the firmware supports the Rx FIFO Overrun bit (BSR D21), the Rx FIFO Underrun bit (BSR D22), and the Tx FIFO Overrun bit (BSR D23).
HPDI32_QUERY_SINGLE_CYC_DIS	This indicates if the firmware supports the Single Cycle Disable Bit (BCR D11).
HPDI32_QUERY_TX_AUTO_STOP	This indicates if the firmware supports the Tx Start Auto-Clear Disable bit (BCR D6).
HPDI32_QUERY_TX_CLOCK_DIV_MAX	This indicates the maximum supported value for the Tx Clock Divider.
HPDI32_QUERY_TX_CLOCK_DIV_MIN	This indicates the minimum supported value for the Tx Clock Divider.
HPDI32_QUERY_USER_JUMPERS	This indicates if the firmware supports the User Jumper status bits (BSR D16 and D17).
HPDI32_QUERY_XCVR_TYPE	This indicates the board's transceiver type. The options are listed below.

Valid return values are as indicated in the above table and as given in the below table.

Value	Description
HPDI32_IOCTL_QUERY_ERROR	Either there was a processing error or the query option is unrecognized.

Valid return values for the Form Factor query are as follows.

Value	Description
HPDI32_QUERY_FF_CPCI	The form factor is Compact PCI.
HPDI32_QUERY_FF_PC104P	The form factor is PC-104+.
HPDI32_QUERY_FF_PCI	The form factor is PCI.
HPDI32_QUERY_FF_PMC	The form factor is PMC.
HPDI32_QUERY_FF_UNKNOWN	The form factor is unknown.

Valid return values for the Transceiver query are as follows.

Value	Description
HPDI32_QUERY_XCVR_PECL	The board has PECL transceivers.
HPDI32_QUERY_XCVR_RS485	The board has RS-485 transceivers.
HPDI32_QUERY_XCVR_UNKNOWN	This indicates that the transceiver type is unknown.

### 6.3.8. HPDI32\_IOCTL\_REG\_MOD

This service performs a read-modify-write of an HPDI32 register. This includes only the GSC firmware registers. The PCI and PLX Feature Set Registers are read-only. Refer to `hpdi32.h` for the complete list of GSC firmware registers.

Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_REG_MOD
arg	<code>gsc_reg_t*</code>

Definition

```
typedef struct
{
    __u32    reg;
    __u32    value;
    __u32    mask;
} gsc_reg_t;
```

Fields	Description
reg	This is set to the identifier for the register to access.
value	This contains the value for the register bits to modify.
mask	This specifies the set of bits to modify. If a bit here is set, then the respective register bit is modified. If a bit here is zero, then the respective register bit is unmodified.

### 6.3.9. HPDI32\_IOCTL\_REG\_READ

This service reads the value of an HPDI32 register. This includes the PCI registers, the PLX Feature Set Registers and the GSC firmware registers. Refer to `hpdi32.h`, `gsc_pci9080.h` and to `gsc_pci9656.h` for the complete list of accessible registers.

Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_REG_READ
arg	<code>gsc_reg_t*</code>

Definition

```
typedef struct
{
    __u32    reg;
    __u32    value;
    __u32    mask;
} gsc_reg_t;
```

Fields	Description
reg	This is set to the identifier for the register to access.
value	This is the value read from the specified register.
mask	This is ignored for read request.

### 6.3.10. HPDI32\_IOCTL\_REG\_WRITE

This service writes a value to an HPDI32 register. This includes only the GSC firmware registers. The PCI and PLX Feature Set Registers are read-only. Refer to `hpdi32.h` for a complete list of the GSC firmware registers.

#### Usage

<code>ioctl()</code> Argument	Description
request	HPDI32_IOCTL_REG_WRITE
arg	<code>gsc_reg_t*</code>

#### Definition

```
typedef struct
{
    __u32    reg;
    __u32    value;
    __u32    mask;
} gsc_reg_t;
```

Fields	Description
reg	This is set to the identifier for the register to access.
value	This is the value to write to the specified register.
mask	This is ignored for write request.

### 6.3.11. HPDI32\_IOCTL\_RX\_AUTO\_START

This service configures the driver to automatically enable the receiver when a data read request is made.

#### Usage

<code>ioctl()</code> Argument	Description
request	HPDI32_IOCTL_RX_AUTO_START
arg	<code>__s32*</code>

Valid argument values include the options listed below.

Value	Description
-1	Retrieve the current setting.
HPDI32_AUTO_START_NO	Do not enable the receiver when performing read requests.
HPDI32_AUTO_START_YES	Enable the receiver when performing read requests. This is the default

### 6.3.12. HPDI32\_IOCTL\_RX\_ENABLE

This service enables or disables the receiver.



## Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_ENABLE
arg	__s32*

Valid argument values include the options listed below.

<b>Value</b>	<b>Description</b>
-1	Retrieve the current setting.
HPDI32_RX_ENABLE_NO	This disables the receiver.
HPDI32_RX_ENABLE_YES	This enables the receiver.

**6.3.13. HPDI32\_IOCTL\_RX\_FIFO\_AE**

This service configures the Rx FIFO fill level at which the Almost Empty status is asserted. The status is asserted when the FIFO contains *Almost Empty* or fewer samples. The Rx FIFO is reset when a setting is applied to the board.

## Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_FIFO_AE
arg	__s32*

Valid argument values are from zero to 0xFFFF, or -1 to retrieve the current setting.

**6.3.14. HPDI32\_IOCTL\_RX\_FIFO\_AF**

This service configures the Rx FIFO fill level at which the Almost Full status is asserted. The status is asserted when the FIFO can receive *Almost Full* or fewer samples before being full. The Rx FIFO is reset when a setting is applied to the board.

## Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_FIFO_AF
arg	__s32*

Valid argument values are from zero to 0xFFFF, or -1 to retrieve the current setting.

**6.3.15. HPDI32\_IOCTL\_RX\_FIFO\_OVERRUN**

This service operates on the Rx FIFO overrun status.

**NOTE:** An overrun occurs when data is clocked into the Rx FIFO while the FIFO is already full. This typically occurs only when the rate at which data enters the FIFO exceeds the rate at which an application is able read the data.

## Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_FIFO_OVERRUN
arg	__s32*

Valid argument values supplied to the service are as follows.

Value	Description
-1	Retrieve the current state.
HPDI32_FIFO_ERROR_CLEAR	Clear the overrun status.
HPDI32_FIFO_ERROR_TEST	Report the current status.

The current state is reported as one of the following values. This service will always return the current overrun state.

Value	Description
HPDI32_FIFO_ERROR_NO	The FIFO has not experienced an overrun condition.
HPDI32_FIFO_ERROR_YES	The FIFO has experienced an overrun condition.

### 6.3.16. HPDI32\_IOCTL\_RX\_FIFO\_RESET

This service resets the Rx FIFO, which clears the content and applies any pending Almost Empty or Almost Full settings.

**NOTE:** When Almost Empty and Almost Full settings are applied via their respective IOCTL services, the driver automatically performs a FIFO reset to apply the changes. The Almost Empty level is adjusted via the service HPDI32\_IOCTL\_RX\_FIFO\_AE (section 6.3.13, page 33). The Almost Full level is adjusted via the service HPDI32\_IOCTL\_RX\_FIFO\_AF (section 6.3.14, page 33).

Usage

ioctl( ) Argument	Description
request	HPDI32_IOCTL_RX_FIFO_RESET
arg	Not used.

### 6.3.17. HPDI32\_IOCTL\_RX\_FIFO\_STATUS

This service retrieves the current Rx FIFO fill level status.

Usage

ioctl( ) Argument	Description
request	HPDI32_IOCTL_RX_FIFO_STATUS
arg	__s32*

The current status is reported as one of the following values.

Value	Description
HPDI32_FIFO_STATUS_ALMOST_EMPTY	The FIFO contains <i>Almost Empty</i> values or fewer.
HPDI32_FIFO_STATUS_ALMOST_FULL	The FIFO has room to accept <i>Almost Full</i> additional values or fewer before becoming full.
HPDI32_FIFO_STATUS_EMPTY	The FIFO is empty.
HPDI32_FIFO_STATUS_FULL	The FIFO is full.
HPDI32_FIFO_STATUS_MEDIUM	The FIFO's fill level is between the <i>Almost Empty</i> mark and the <i>Almost Full</i> mark.

### 6.3.18. HPDI32\_IOCTL\_RX\_FIFO\_UNDERRUN

This service operates on the Rx FIFO underrun status.

**NOTE:** An Rx FIFO underrun occurs when the FIFO is read while empty. This can typically only when an applications reads from the FIFO directly.

#### Usage

<b>ioctl( ) Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_FIFO_UNDERRUN
arg	__s32*

Valid argument values supplied to the service are as follows.

<b>Value</b>	<b>Description</b>
-1	Retrieve the current state.
HPDI32_FIFO_ERROR_CLEAR	Clear the underrun status.
HPDI32_FIFO_ERROR_TEST	Report the current status.

The current state is reported as one of the following values. This service will always return the current underrun status.

<b>Value</b>	<b>Description</b>
HPDI32_FIFO_ERROR_NO	The FIFO has not experienced an underrun condition.
HPDI32_FIFO_ERROR_YES	The FIFO has experienced an underrun condition.

### 6.3.19. HPDI32\_IOCTL\_RX\_IO\_ABORT

This service aborts an ongoing read( ) request.

#### Usage

<b>ioctl( ) Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_IO_ABORT
arg	__s32*

The results are reported as one of the following values.

<b>Value</b>	<b>Description</b>
HPDI32_IO_ABORT_NO	A read( ) request was not aborted as none were ongoing.
HPDI32_IO_ABORT_YES	An ongoing read( ) request was aborted.

### 6.3.20. HPDI32\_IOCTL\_RX\_IO\_DATA\_SIZE

This service configures the number of cable interface data bits used for read operations. Data bit D0 is always aligned with cable signal D0.

**NOTE:** The data size refers to the number of Rx FIFO data bits used when data is read from the FIFO. FIFO data values are always 32-bit wide and any unused bits are simply ignored.

#### Usage

<b>ioctl( ) Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_IO_DATA_SIZE
arg	__s32*

Valid argument values supplied to the service are as follows.

Value	Description
-1	Retrieve the current setting.
HPDI32_IO_DATA_SIZE_8_BITS	The data size is 8-bits.
HPDI32_IO_DATA_SIZE_16_BITS	The data size is 16-bits.
HPDI32_IO_DATA_SIZE_32_BITS	The data size is 32-bits. This is the default.

### 6.3.21. HPDI32\_IOCTL\_RX\_IO\_MODE

This service selects the mechanism used to retrieve data from the Rx FIFO during read requests.

Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_RX_IO_MODE
arg	__s32*

Valid argument values supplied to the service are as follows.

Value	Description
-1	Retrieve the current setting.
GSC_IO_MODE_DMA	Data is retrieved using DMA. In this mode the data must already reside in the FIFO.
GSC_IO_MODE_DMDMA	Data is retrieved using Demand Mode DMA. In this mode the request can exceed the FIFO size as data is retrieved from the FIFO as it becomes available. This is the default.
GSC_IO_MODE_PIO	Data is retrieved using repetitive register accesses.

### 6.3.22. HPDI32\_IOCTL\_RX\_IO\_OVERRUN

This service configures the read service to check for an Rx FIFO overrun before performing read operations. Data is lost when there is an overrun. If the check is performed and an overrun is detected, then the read service immediately returns an error.

**NOTE:** The check for an overrun is performed upon entry to the read service. The read service does not check for overruns that occur while the read is in progress. For in-progress overruns an application must perform the check manually or wait for the check performed by a subsequent read request.

Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_RX_IO_OVERRUN
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
HPDI32_IO_ERROR_CHECK	Perform the check. This is the default.
HPDI32_IO_ERROR_IGNORE	Do not perform the check.

**6.3.23. HPDI32\_IOCTL\_RX\_IO\_PIO\_THRESHOLD**

This service sets the threshold at which DMA read requests will instead resort to PIO mode. When the number of samples in a read request is less than or equal to this value, then the operation will automatically use PIO instead of DMA. This is intended to improve efficiency as small read requests can be performed more efficiently when done using PIO rather than DMA.

**Usage**

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_IO_PIO_THRESHOLD
arg	__s32*

Valid argument values are any non-negative number, or -1 to retrieve the current setting. The default is 32 samples.

**6.3.24. HPDI32\_IOCTL\_RX\_IO\_TIMEOUT**

This service sets the timeout limit for read requests. The value is expressed in seconds. The timeout limit is the total amount of time allowed for a single read() request. When this time limit has expired the service will terminate. When this occurs the read() return value will be less than the number of bytes requested, and possibly zero.

**Usage**

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_IO_TIMEOUT
arg	__s32*

Valid argument values are in the range from zero to 3600, and -1. A value of zero tells the driver not to sleep in order to wait for more data, and should only be used with PIO mode reads. A value of -1 is used to retrieve the current setting. The default is 10 seconds.

**6.3.25. HPDI32\_IOCTL\_RX\_IO\_UNDERRUN**

This service operates on the Rx FIFO underrun status.

**Usage**

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_RX_IO_UNDERRUN
arg	__s32*

Valid argument values supplied to the service are as follows.

<b>Value</b>	<b>Description</b>
-1	Retrieve the current state.
HPDI32_IO_ERROR_CHECK	Check the underrun status. This is the default.
HPDI32_IO_ERROR_IGNORE	Ignore the current status.

**6.3.26. HPDI32\_IOCTL\_RX\_LINE\_COUNT**

This service reports the number of values received during the most recent frame during the periods in which the Line Valid signal was asserted.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_RX_LINE_COUNT
arg	__s32*

The count returned is from zero to 0xFFFFFFFF.

**6.3.27. HPDI32\_IOCTL\_RX\_STATUS\_COUNT**

This service reports the number of values received during the most recent frame during the periods in which the Status Valid signal was asserted.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_RX_STATUS_COUNT
arg	__s32*

The count returned is from zero to 0xFFFFFFFF.

**6.3.28. HPDI32\_IOCTL\_TRISTATE\_TE\_RE**

This service configures the Tx Enabled and Rx Enabled signals on the cable interface. When these signals appear on the cable interface they may be either driven or tri-stated. This is intended to accommodate the circumstance where two HPDI32 boards are connected back-to-back, which is when the signals must be tri-stated.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TRISTATE_TE_RE
arg	__s32*

Valid argument values include the options listed below.

<b>Value</b>	Description
-1	Retrieve the current setting.
HPDI32_TRISTATE_TE_RE_NO	The signals are driven on the cable interface.
HPDI32_TRISTATE_TE_RE_YES	The signals are tri-stated.

**6.3.29. HPDI32\_IOCTL\_TX\_AUTO\_START**

This service configures the driver to automatically enable the transmitter and start transmission when a data write request is made.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TX_AUTO_START
arg	__s32*

Valid argument values include the options listed below.

Value	Description
-1	Retrieve the current setting.
HPDI32_AUTO_START_NO	Do not enable the transmitter or start transmission when performing write requests.
HPDI32_AUTO_START_YES	Enable the transmitter and start transmission when performing write requests. This is the default.

### 6.3.30. HPDI32\_IOCTL\_TX\_AUTO\_STOP

This service configures the driver to automatically end data transmission any time the Tx FIFO becomes empty.

**NOTE:** This service is operable only if it is supported by firmware. Refer to the HPDI32\_IOCTL\_QUERY\_IOCTL service option HPDI32\_QUERY\_TX\_AUTO\_STOP (section 6.3.7, page 29) to determine if this feature is supported by firmware.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_AUTO_STOP
arg	__s32*

Valid argument values include the options listed below.

Value	Description
-1	Retrieve the current setting.
HPDI32_AUTO_STOP_NO	Do not terminate transmission when the Tx FIFO becomes empty. This is the default option when the feature is supported by firmware.
HPDI32_AUTO_STOP_YES	Terminate transmission when the Tx FIFO becomes empty. This is the default option when the feature is unsupported by firmware.

### 6.3.31. HPDI32\_IOCTL\_TX\_CLOCK\_DIVIDER

This service sets the clock divider used with the Tx Clock's onboard oscillator.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_CLOCK_DIVIDER
arg	__s32*

Valid argument values are in the range from zero to 0xFFFF, and -1. A value of -1 is used to retrieve the current setting.

### 6.3.32. HPDI32\_IOCTL\_TX\_ENABLE

This service enables or disables the transmitter.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_ENABLE
arg	__s32*

Valid argument values include the options listed below.

Value	Description
-1	Retrieve the current setting.
HPDI32_TX_ENABLE_NO	This disables the transmitter.
HPDI32_TX_ENABLE_YES	This enables the transmitter.

### 6.3.33. HPDI32\_IOCTL\_TX\_FIFO\_AE

This service configures the Tx FIFO fill level at which the Almost Empty status is asserted. The status is asserted when the FIFO contains *Almost Empty* or fewer samples. The Tx FIFO is reset when a setting is applied to the board.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_FIFO_AE
arg	__s32*

Valid argument values are from zero to 0xFFFF, or -1 to retrieve the current setting.

### 6.3.34. HPDI32\_IOCTL\_TX\_FIFO\_AF

This service configures the Tx FIFO fill level at which the Almost Full status is asserted. The status is asserted when the FIFO can receive *Almost Full* or fewer samples before being full. The Tx FIFO is reset when a setting is applied to the board.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_FIFO_AF
arg	__s32*

Valid argument values are from zero to 0xFFFF, or -1 to retrieve the current setting.

### 6.3.35. HPDI32\_IOCTL\_TX\_FIFO\_OVERRUN

This service operates on the Tx FIFO overrun status.

**NOTE:** A overrun occurs when data is written to the Tx FIFO while it is already full. This can typically occur only when an application writes to the Tx FIFO directly.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_FIFO_OVERRUN
arg	__s32*

Valid argument values supplied to the service are as follows.

Value	Description
-1	Retrieve the current state.
HPDI32_FIFO_ERROR_CLEAR	Clear the overrun status.
HPDI32_FIFO_ERROR_TEST	Report the current status.



The current state is reported as one of the following values. This service will always return the current overrun status.

Value	Description
HPDI32_FIFO_ERROR_NO	The FIFO has not experienced an overrun condition.
HPDI32_FIFO_ERROR_YES	The FIFO has experienced an overrun condition.

### 6.3.36. HPDI32\_IOCTL\_TX\_FIFO\_RESET

This service resets the Tx FIFO, which clears the content and applies any pending Almost Empty or Almost Full settings.

**NOTE:** When Almost Empty or Almost Full settings are applied via their respective IOCTL services, the driver automatically performs a FIFO reset to apply the changes. The Almost Empty level is adjusted via the service HPDI32\_IOCTL\_TX\_FIFO\_AE (section 6.3.33, page 40). The Almost Full level is adjusted via the service HPDI32\_IOCTL\_TX\_FIFO\_AF (section 6.3.34, page 40).

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_FIFO_RESET
arg	Not used.

### 6.3.37. HPDI32\_IOCTL\_TX\_FIFO\_STATUS

This service retrieves the current Tx FIFO fill level status.

#### Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_TX_FIFO_STATUS
arg	__s32*

The current status is reported as one of the following values.

Value	Description
HPDI32_FIFO_STATUS_ALMOST_EMPTY	The FIFO contains <i>Almost Empty</i> values or fewer.
HPDI32_FIFO_STATUS_ALMOST_FULL	The FIFO has room to accept <i>Almost Full</i> additional values or fewer before becoming full.
HPDI32_FIFO_STATUS_EMPTY	The FIFO is empty.
HPDI32_FIFO_STATUS_FULL	The FIFO is full.
HPDI32_FIFO_STATUS_MEDIUM	The FIFO's fill level is between the <i>Almost Empty</i> mark and the <i>Almost Full</i> mark.

### 6.3.38. HPDI32\_IOCTL\_TX\_FLOW\_CONTROL

This service starts and stops data transmission. The transmitter must be enabled for data transmission to be started (HPDI32\_IOCTL\_TX\_ENABLE, section 6.3.32, page 39).

**NOTE:** If this service is used to stop data transmission, then the desired operation may be negated if the Auto Start feature is enabled. Refer to the HPDI32\_IOCTL\_TRISTATE\_TE\_RE IOCTL service (section 6.3.28, page 38).

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TX_FLOW_CONTROL
arg	__s32*

Valid argument values include the options listed below.

<b>Value</b>	Description
-1	Retrieve the current setting.
HPDI32_TX_FLOW_CONTROL_START	Data transmission is enabled.
HPDI32_TX_FLOW_CONTROL_STOP	Data transmission is disabled.

**6.3.39. HPDI32\_IOCTL\_TX\_IO\_ABORT**

This service aborts an ongoing `write()` request.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TX_IO_ABORT
arg	__s32*

The results are reported as one of the following values.

<b>Value</b>	Description
HPDI32_IO_ABORT_NO	A <code>write()</code> request was not aborted as none were ongoing.
HPDI32_IO_ABORT_YES	An ongoing <code>write()</code> request was aborted.

**6.3.40. HPDI32\_IOCTL\_TX\_IO\_DATA\_SIZE**

This service configures the number of cable interface data bits used for write operations. Data bit D0 is always aligned with cable signal D0.

**NOTE:** The data size refers to the number of Tx FIFO data bits used when data is written to the FIFO. FIFO data values are always 32-bit wide and any unused bits are indeterminate.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TX_IO_DATA_SIZE
arg	__s32*

Valid argument values supplied to the service are as follows.

<b>Value</b>	Description
-1	Retrieve the current setting.
HPDI32_IO_DATA_SIZE_8_BITS	The data size is 8-bits.
HPDI32_IO_DATA_SIZE_16_BITS	The data size is 16-bits.
HPDI32_IO_DATA_SIZE_32_BITS	The data size is 32-bits. This is the default.

**6.3.41. HPDI32\_IOCTL\_TX\_IO\_MODE**

This service selects the mechanism used to move data to the Tx FIFO during write requests.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TX_IO_MODE
arg	__s32*

Valid argument values supplied to the service are as follows.

<b>Value</b>	Description
-1	Retrieve the current setting.
GSC_IO_MODE_DMA	Data is moved using DMA. In this mode the data written must fit in the FIFO.
GSC_IO_MODE_DMDMA	Data is moved using Demand Mode DMA. In this mode the request can exceed the FIFO size as data is moved to the FIFO as space becomes available. This is the default.
GSC_IO_MODE_PIO	Data is written using repetitive register accesses.

**6.3.42. HPDI32\_IOCTL\_TX\_IO\_OVERRUN**

This service configures the write service to check for a Tx FIFO overrun before performing write operations. Data is lost when there is an overrun. If the check is performed and an overrun is detected, then the write service immediately returns an error.

**NOTE:** The check for an overrun is performed upon entry to the write service. The write service does not check for overruns that occur while the write is in progress. For in-progress overruns an application must perform the check manually or wait for the check performed by a subsequent write request.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TX_IO_OVERRUN
arg	__s32*

Valid argument values are as follows.

<b>Value</b>	Description
-1	Retrieve the current setting.
HPDI32_IO_ERROR_CHECK	Perform the check. This is the default.
HPDI32_IO_ERROR_IGNORE	Do not perform the check.

**6.3.43. HPDI32\_IOCTL\_TX\_IO\_PIO\_THRESHOLD**

This service sets the threshold at which DMA write requests will instead resort to PIO mode. When the number of samples in a write request is less than or equal to this value, then the operation will automatically use PIO instead of DMA. This is intended to improve efficiency as small write requests can be performed more efficiently when done using PIO rather than DMA.

## Usage

<b>ioctl()</b> Argument	Description
request	HPDI32_IOCTL_TX_IO_PIO_THRESHOLD
arg	__s32*

Valid argument values are any non-negative number, or -1 to retrieve the current setting. The default is 32 samples.

**6.3.44. HPDI32\_IOCTL\_TX\_IO\_TIMEOUT**

This service sets the timeout limit for write requests. The value is expressed in seconds. The timeout limit is the total amount of time allowed for a single `write()` request. When this time limit has expired the service will terminate. When this occurs the `write()` return value will be less than the number of bytes requested, and possibly zero.

**Usage**

<b>ioctl()</b> Argument	<b>Description</b>
request	HPDI32_IOCTL_TX_IO_TIMEOUT
arg	__s32*

Valid argument values are in the range from zero to 3600, and -1. A value of zero tells the driver not to sleep in order to wait for more space in the Tx FIFO, and should only be used with PIO mode writes. A value of -1 is used to retrieve the current setting. The default is 10 seconds.

**6.3.45. HPDI32\_IOCTL\_TX\_LINE\_VAL\_OFF\_CNT**

This service sets the number of transmit clock cycles that the Line Valid signal is negated preceding each Tx Line Valid assertion period. Data is not clocked out the cable interface while the Line Valid signal is negated.

**NOTE:** This parameter, as well as the Line Valid signal itself, is ignored if the Cable Command 1 signal is configured for GPIO operation rather than for Line Valid operation.

**Usage**

<b>ioctl()</b> Argument	<b>Description</b>
request	HPDI32_IOCTL_TX_LINE_VAL_OFF_CNT
arg	__s32*

Valid argument values are in the range from zero to 0xFFFF, and -1. A value of -1 is used to retrieve the current setting.

**6.3.46. HPDI32\_IOCTL\_TX\_LINE\_VAL\_ON\_CNT**

This service sets the number of transmit clock cycles that the Line Valid signal is asserted following each Tx Line Valid negation period. Data is clocked out the cable interface while the Line Valid signal is asserted.

**NOTE:** This parameter, as well as the Line Valid signal itself, is ignored if the Cable Command 1 signal is configured for GPIO operation rather than for Line Valid operation.

**Usage**

<b>ioctl()</b> Argument	<b>Description</b>
request	HPDI32_IOCTL_TX_LINE_VAL_ON_CNT
arg	__s32*

Valid argument values are any non-negative number, and -1. A value of -1 is used to retrieve the current setting. To apply the 32-bit equivalent of -1 an application must access the Tx Line Valid Length Count Register (TLVLCR) directly.

**6.3.47. HPDI32\_IOCTL\_TX\_REMOTE\_THROTTLE**

This service enables or disables the transmitter's ability to pause data transmission via the Rx Ready cable signal, as driven by a receiving device.

**NOTE:** The board's Tx Flow Control feature (HPDI32\_IOCTL\_TX\_FLOW\_CONTROL, section 6.3.38, page 41) operates in parallel with the Remote Throttle feature. When using Remote Throttling the Tx Flow Control feature should be set to stop data flow and the Tx Auto Start feature (HPDI32\_IOCTL\_TX\_AUTO\_START, section 6.3.29, page 38) should be disabled.

**Usage**

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_TX_REMOTE_THROTTLE
arg	__s32*

Valid argument values are as follows.

<b>Value</b>	<b>Description</b>
-1	Retrieve the current setting.
HPDI32_TX_REMOTE_THROTTLE_NO	This option disables remote throttling.
HPDI32_TX_REMOTE_THROTTLE_YES	This option enables remote throttling.

**6.3.48. HPDI32\_IOCTL\_TX\_STATUS\_VAL\_CNT**

This service sets the number of transmit clock cycles that the Status Valid signal is asserted at the beginning of each data frame. Data is clocked out the cable interface while the Status Valid signal is asserted.

**NOTE:** This parameter, as well as the Status Valid signal itself, is ignored if the Cable Command 2 signal is configured for GPIO operation rather than for Status Valid operation.

**Usage**

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_TX_STATUS_VAL_CNT
arg	__s32*

Valid argument values are any non-negative 32-bit number, and -1. A value of -1 is used to retrieve the current setting. To apply the 32-bit equivalent of -1 an application must access the Tx Status Valid Length Count Register (TSVLCR) directly.

**6.3.49. HPDI32\_IOCTL\_TX\_STATUS\_VAL\_MIR**

This service configures the asserted Status Valid signal to be mirrored onto, or assert, the Line Valid signal.

**NOTE:** This parameter, as well as the Status Valid signal itself, is ignored if the Cable Command 2 signal is configured for GPIO operation rather than for Status Valid operation.

**Usage**

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_TX_STATUS_VAL_MIR
arg	__s32*

Valid argument values are as follows.

Value	Description
-1	Retrieve the current setting.
HPDI32_TX_STATUS_VAL_MIR_NO	While asserted, the Status Valid signal is not mirrored onto the Line Valid signal. The Line Valid signal remains negated while the Status Valid signal is asserted.
HPDI32_TX_STATUS_VAL_MIR_YES	While asserted, the Status Valid signal is mirrored onto the Line Valid signal. The Line Valid signal is asserted while the Status Valid signal is asserted.

### 6.3.50. HPDI32\_IOCTL\_USER\_JUMPERS

This service retrieves the status for the user installable jumpers.

Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_USER_JUMPERS
arg	__s32*

The current state is reported as one of the following values.

Value	Description
0x0	Either neither jumper is installed or the board doesn't the jumper feature. *
0x1	Only jumper one is installed
0x2	Only jumper two is installed
0x3	Both jumpers are installed

\* Refer to the HPDI32\_IOCTL\_QUERY IOCTL service's HPDI32\_QUERY\_USER\_JUMPERS option to determine if the board supports the user jumper feature (section 6.3.7, page 29).

### 6.3.51. HPDI32\_IOCTL\_WAIT\_CANCEL

This service resumes all threads blocked via HPDI32\_IOCTL\_WAIT\_EVENT IOCTL calls (section 6.3.52, page 47), according to the provided criteria. When a blocked thread is waiting for any event specified in the structure, then the thread is resumed.

**NOTE:** The driver itself makes use of the wait services for various internal operations. Driver initiated waits are unaffected by application cancel requests.

Usage

ioctl() Argument	Description
request	HPDI32_IOCTL_WAIT_CANCEL
arg	gsc_wait_t*

Definition

```
typedef struct
{
    __u32    flags;
    __u32    main;
    __u32    gsc;
    __u32    alt;
```

```

__u32    io;
__u32    timeout_ms;
__u32    count;
} gsc_wait_t;

```

Fields	Description
flags	This is unused by wait cancel operations.
main	This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be cancelled. Refer to section 6.3.52.2 on page 48.
gsc	This specifies the set of HPDI32_WAIT_GSC_* events whose wait requests are to be cancelled. Refer to section 6.3.52.3 on page 48.
alt	This is unused by the HPDI32 driver and should be zero.
io	This specifies the set of GSC_WAIT_IO_* events whose wait requests are to be cancelled. Refer to section 6.3.52.4 on page 49.
timeout_ms	This is unused by wait cancel operations.
count	Upon return this indicates the number of waits that were cancelled.

### 6.3.52. HPDI32\_IOCTL\_WAIT\_EVENT

This service blocks a thread until any one of a specified set of events occurs, or until a timeout lapses, whichever occurs first. The set of possible events to wait for are specified in the structure's `main`, `gsc`, `alt` and `io` fields. All field values must be valid and at least one event must be specified. If the thread is resumed because one of the referenced events has occurred, then the bit for the respective event is the only event bit that will be set. All other event bits and fields will be zero. (Multiple event bits will be set only if the events occur simultaneously.)

**NOTE:** A wait timeout is reported via the `gsc_wait_t` structure's `flags` field having the `GSC_WAIT_FLAG_TIMEOUT` flag set, rather than via an `ETIMEDOUT` error.

#### Usage

<code>ioctl()</code> Argument	Description
<code>request</code>	<code>HPDI32_IOCTL_WAIT_EVENT</code>
<code>arg</code>	<code>gsc_wait_t*</code>

#### Definition

```

typedef struct
{
    __u32    flags;
    __u32    main;
    __u32    gsc;
    __u32    alt;
    __u32    io;
    __u32    timeout_ms;
    __u32    count;
} gsc_wait_t;

```

Fields	Description
flags	This must initially be zero. Upon return this indicates the reason that the thread was resumed. Refer to section 6.3.52.1 on page 48.
main	This specifies any number of GSC_WAIT_MAIN_* events that the thread is to wait for. Refer to section 6.3.52.2 on page 48.
gsc	This specifies any number of HPDI32_WAIT_GSC_* events that the thread is to wait for. Refer to section 6.3.52.3 on page 48.

alt	This is unused by the HPDI32 driver and must be zero.
io	This specifies any number of GSC_WAIT_IO_* events that the thread is to wait for. Refer to section 6.3.52.4 on page 49.
timeout_ms	This specified the maximum amount of time, in milliseconds, that the thread is to wait for any of the referenced events. This must be greater than zero. Upon return the value will be the approximate amount of time actually waited.
count	This is unused by wait event operations and must be zero.

#### 6.3.52.1. gsc\_wait\_t.flags Options

Upon return from a wait request the wait structure's flags field will indicate the reason that the thread was resumed. Only one of the below option will be set.

Fields	Description
GSC_WAIT_FLAG_CANCEL	The wait request was cancelled.
GSC_WAIT_FLAG_DONE	One of the referenced events occurred.
GSC_WAIT_FLAG_TIMEOUT	The timeout period lapsed before a referenced event occurred.

#### 6.3.52.2. gsc\_wait\_t.main Options

The wait structure's main field may specify any of the below primary interrupt options. These interrupt options are supported by the HPDI32 and other General Standards products.

Fields	Description
GSC_WAIT_MAIN_DMA0	This refers to the DMA Done interrupt on DMA engine number zero.
GSC_WAIT_MAIN_DMA1	This refers to the DMA Done interrupt on DMA engine number one.
GSC_WAIT_MAIN_GSC	This refers to any of the Interrupt Control/Status Register interrupts.
GSC_WAIT_MAIN_OTHER	This generally refers to an interrupt generated by another device sharing the same interrupt as the HPDI32.
GSC_WAIT_MAIN_PCI	This refers to any interrupt generated by the HPDI32.
GSC_WAIT_MAIN_SPURIOUS	This refers to board interrupts which should never be generated.
GSC_WAIT_MAIN_UNKNOWN	This refers to board interrupts whose source could not be identified.

#### 6.3.52.3. gsc\_wait\_t.gsc Options

The wait structure's gsc field may specify any combination of the below interrupt options. These are the interrupt options referenced in the Interrupt Control Register. Applications are responsible for enabling the desired interrupt options. Refer to HPDI32\_IOCTL\_IRQ\_ENABLE (section 6.3.6, page 28). If a board supports the Interrupt Edge/Level Register then interrupts configured for level triggering are disabled by the driver when they occur. If a board does not support this register then all interrupt are disabled by the driver when they occur.

Value	Description
HPDI32_WAIT_GSC_CC0_FV_E_GPIO6	This refers to the Cable Command 0 signal, whose default configuration is to trigger on a falling edge. This signal may be configured as Frame Valid or GPIO 6.
HPDI32_WAIT_GSC_CC0_FV_S_GPIO6	This refers to the Cable Command 0 signal, whose default configuration is to trigger on a rising edge. This signal may be configured as Frame Valid or GPIO 6.
HPDI32_WAIT_GSC_CC1_LV_GPIO0	This refers to the Cable Command 1 signal. This signal may be configured as Line Valid or GPIO 0.
HPDI32_WAIT_GSC_CC2_SV_GPIO1	This refers to the Cable Command 2 signal. This signal may be configured as Status Valid or GPIO 1.
HPDI32_WAIT_GSC_CC3_RR_GPIO2	This refers to the Cable Command 3 signal. This signal may be configured as Rx Ready or GPIO 2.



HPDI32_WAIT_GSC_CC4_TR_GPIO3	This refers to the Cable Command 4 signal. This signal may be configured as Tx Data Ready or GPIO 3.
HPDI32_WAIT_GSC_CC5_TE_GPIO4	This refers to the Cable Command 4 signal. This signal may be configured as Tx Enabled or GPIO 4.
HPDI32_WAIT_GSC_CC6_RE_GPIO5	This refers to the Cable Command 5 signal. This signal may be configured as Rx Enabled or GPIO 5.
HPDI32_WAIT_GSC_RX_FIFO_AE	This refers to the Rx FIFO's Almost Empty status.
HPDI32_WAIT_GSC_RX_FIFO_AF	This refers to the Rx FIFO's Almost Full status.
HPDI32_WAIT_GSC_RX_FIFO_EMPTY	This refers to the Rx FIFO's empty status.
HPDI32_WAIT_GSC_RX_FIFO_FULL	This refers to the Rx FIFO's full status.
HPDI32_WAIT_GSC_TX_FIFO_AE	This refers to the Tx FIFO's Almost Empty status.
HPDI32_WAIT_GSC_TX_FIFO_AF	This refers to the Tx FIFO's Almost Full status.
HPDI32_WAIT_GSC_TX_FIFO_EMPTY	This refers to the Tx FIFO's empty status.
HPDI32_WAIT_GSC_TX_FIFO_FULL	This refers to the Tx FIFO's full status.

#### 6.3.52.4. gsc\_wait\_t.io Options

The wait structure's io field may specify any of the below event options. These events are generated in response to application board data read requests.

Fields	Description
GSC_WAIT_IO_RX_ABORT	This refers to read requests which have been aborted.
GSC_WAIT_IO_RX_DONE	This refers to read requests which have been satisfied.
GSC_WAIT_IO_RX_ERROR	This refers to read requests which end due to an error.
GSC_WAIT_IO_RX_TIMEOUT	This refers to read requests which end due to the timeout period lapse.
GSC_WAIT_IO_TX_ABORT	This refers to write requests which have been aborted.
GSC_WAIT_IO_TX_DONE	This refers to write requests which have been satisfied.
GSC_WAIT_IO_TX_ERROR	This refers to write requests which end due to an error.
GSC_WAIT_IO_TX_TIMEOUT	This refers to write requests which end due to the timeout period lapse.

#### 6.3.53. HPDI32\_IOCTL\_WAIT\_STATUS

This service count all threads blocked via the HPDI32\_IOCTL\_WAIT\_EVENT IOCTL service (section 6.3.52, page 47), according to the provided criteria. A match is made when a waiting thread's wait criteria matches any of the criteria specified in the structure passed to this service.

**NOTE:** The driver itself makes use of the wait services for various internal operations. Driver initiated waits are ignored by application status requests.

#### Usage

ioctl( ) Argument	Description
request	HPDI32_IOCTL_WAIT_STATUS
arg	gsc_wait_t*

#### Definition

```
typedef struct
{
    __u32    flags;
    __u32    main;
    __u32    gsc;
    __u32    alt;
    __u32    io;
```

```

    __u32    timeout_ms;
    __u32    count;
} gsc_wait_t;

```

Fields	Description
flags	This is unused by wait status operations.
main	This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be counted. Refer to section 6.3.52.2 on page 48.
gsc	This specifies the set of HPDI32_WAIT_GSC_* events whose wait requests are to be counted. Refer to section 6.3.52.3 on page 48.
alt	This is unused by the HPDI32 driver and should be zero.
io	This specifies the set of GSC_WAIT_IO_* events whose wait requests are to be counted. Refer to section 6.3.52.4 on page 49.
timeout_ms	This is unused by wait status operations.
count	Upon return this indicates the number of waits that met any of the specified criteria.

## Document History

Revision	Description
December 20, 2011	Updated to release version 2.3.34.0.
November 2, 2011	Updated to release version 2.2.32.0. Various editorial changes.
Aug 10, 2010	Updated to release version 2.1.17.0. Added several sample applications.
May 21, 2010	Updated to release version 2.0.16.0. Overhauled the driver and the documentation. Removed mmap interface support. Updated the CPU and Kernel Support information. Updated the comments for the Initialize IOCTL service. Added a number of new services.
July 30, 2007	Updated to release version 1.19.0. The driver was updated for the 2.6.19 kernel.
September 29, 2006	Updated to release version 1.18.0. Added Data Size and PIO Threshold I/O parameters.
August 23, 2006	Updated to release version 1.17.0. Updated to support 64-bit kernels and newer 2.6 kernels.
May 30, 2006	Updated to release version 1.16.0. Various minor updates. Updated the irq sample application.
April 24, 2006	Updated to release version 1.15.1.
April 18, 2006	Updated to release version 1.15.0.
February 22, 2006	Updated to release version 1.14.0. Added a make script. Modified the size limit for I/O buffers.
December 19, 2005	Updated to release version 1.13.0.
October 17, 2005	Updated to release version 1.12.1. Correct line below to show 1.12.0.
September 1, 2005	Updated to release version 1.12.0.
January 25, 2005	Updated to release version 1.11.0.
January 14, 2005	Reorganized the directory structure. Ported to the 2.6 kernel.
March 29, 2004	Made correction to interrupt notification example code and documentation. Removed the “tainting” remarks as the driver is now covered by GPL.
May 23, 2003	Minor updates for updated driver.
April 29, 2003	Added note about this being a non-GPL driver. Typographic and formatting corrections.
April 28, 2003	Updated mmap ( ) support, interrupt sharing support and use with DIO24 boards.
July 29, 2002	Ported to the 2.4 kernel (2.4.7-10 with Red Hat 7.2 and 2.4.18-3 with Red Hat 7.3).
June 24, 2002	The HPDI32_IOCTL_INT_NOTIFY IOCTL service enables PCI interrupts unconditionally. Changed MAP_PRIVATE to MAP_SHARED. Removed some extraneous text. Minor corrections. Added Demand Mode DMA documentation as the option is now supported.
June 6, 2002	Reduced restrictions on simultaneous use of mmap ( ) with other driver features.
June 5, 2002	Updated data on the use of standard DMA. Added a section on repetitious data transmission.
May 30, 2002	Added support for the PCI64-HPDI32.
May 24, 2002	Added mmap ( ) support. Expanded DMA information. Various miscellaneous corrections.
April 24, 2002	Initial driver release.