

# HPDI32A

High Performance 32-bit Digital I/O

PCI-HPDI32A-DIPHASE2  
PMC-HPDI32A-DIPHASE2

## Linux DIPHASE2 Library User Manual

Manual Revision: July 30, 2007  
Driver Release 1.19.0

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 ©2007, **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>7</b>
1.1. Purpose .....	7
1.2. Acronyms.....	7
1.3. Definitions .....	7
1.4. Software Overview .....	7
1.5. Hardware Overview .....	7
1.6. Reference Material.....	8
<b>2. Installation.....</b>	<b>9</b>
2.1. CPU and Kernel Support .....	9
2.2. The /proc File System.....	9
2.3. The Driver.....	10
2.4. File List.....	10
2.5. Installation .....	10
2.6. Removal.....	10
2.7. Overall Make Script.....	11
2.8. The Library .....	11
2.8.1. Build.....	11
2.8.2. Version .....	11
2.8.3. Using the Library.....	11
2.9. Document Source Code Examples.....	12
2.9.1. Build.....	12
2.9.2. Use.....	12
2.10. Sample Applications .....	13
2.10.1. lbttest .....	13
2.10.2. sbttest.....	14
<b>3. Application Interface.....</b>	<b>16</b>
3.1. Macros .....	16
3.1.1. Interrupt Identification Bits: GSC Specific .....	16
3.1.2. Interrupt Identification Bits: Non-GSC Specific .....	17
3.1.3. IOCTL .....	17
3.1.4. I/O PIO Threshold Options.....	17
3.1.5. I/O Data Sizes.....	18
3.1.6. I/O Transfer Modes .....	18
3.1.7. I/O Transfer Buffer Size.....	18
3.1.8. I/O Timeouts.....	19
3.1.9. mmap() Register Decoding.....	19
3.1.10. Registers .....	19
3.2. Data Types .....	22
3.2.1. hpdi32_debug_data_t .....	22
3.2.2. hpdi32_driver_info_t.....	22
3.2.3. hpdi32_interrupt_t.....	23

3.2.4. hpdi32_io_config_t.....	23
3.2.5. hpdi32_mmap_mem_t.....	24
3.2.6. hpdi32_mmap_t.....	25
3.2.7. hpdi32_reg_t.....	25
3.3. Driver Functions.....	26
3.3.1. close().....	26
3.3.2. ioctl().....	26
3.3.3. mmap().....	27
3.3.4. munmap().....	29
3.3.5. open().....	29
3.3.6. read().....	30
3.3.7. write().....	31
3.4. Library Functions.....	32
3.4.1. hpdi32_dp2_board_loop_back_get().....	33
3.4.2. hpdi32_dp2_board_loop_back_set().....	33
3.4.3. hpdi32_dp2_board_reset().....	34
3.4.4. hpdi32_dp2_gpio_dir_get().....	35
3.4.5. hpdi32_dp2_gpio_dir_set().....	36
3.4.6. hpdi32_dp2_gpio_mod().....	37
3.4.7. hpdi32_dp2_gpio_read().....	38
3.4.8. hpdi32_dp2_gpio_write().....	38
3.4.9. hpdi32_dp2_predrive_get().....	39
3.4.10. hpdi32_dp2_predrive_set().....	40
3.4.11. hpdi32_dp2_reg_mod().....	41
3.4.12. hpdi32_dp2_reg_read().....	42
3.4.13. hpdi32_dp2_reg_write().....	42
3.4.14. hpdi32_dp2_rx_accum_msg_size().....	43
3.4.15. hpdi32_dp2_rx_config_get().....	44
3.4.16. hpdi32_dp2_rx_config_set().....	45
3.4.17. hpdi32_dp2_rx_fifo_almost_get().....	46
3.4.18. hpdi32_dp2_rx_fifo_almost_set().....	47
3.4.19. hpdi32_dp2_rx_fifo_reset().....	48
3.4.20. hpdi32_dp2_tx_config_get().....	48
3.4.21. hpdi32_dp2_tx_config_set().....	50
3.4.22. hpdi32_dp2_tx_fifo_almost_get().....	51
3.4.23. hpdi32_dp2_tx_fifo_almost_set().....	52
3.4.24. hpdi32_dp2_tx_fifo_reset().....	53
3.4.25. hpdi32_dp2_tx_last_msg_size().....	53
3.4.26. hpdi32_dp2_lib_version_get().....	54
3.5. IOCTL Services.....	55
3.5.1. HPDI32_IOCTL_DEBUG_DATA_GET.....	55
3.5.2. HPDI32_IOCTL_DRIVER_INFO_GET.....	56
3.5.3. HPDI32_IOCTL_INT_NOTIFY.....	57
3.5.4. HPDI32_IOCTL_INT_STATUS.....	58
3.5.5. HPDI32_IOCTL_IO_CONFIG.....	58
3.5.6. HPDI32_IOCTL_MMAP_INFO.....	59
3.5.7. HPDI32_IOCTL_NO_COMMAND.....	60
3.5.8. HPDI32_IOCTL_REG_MOD.....	61
3.5.9. HPDI32_IOCTL_REG_READ.....	62
3.5.10. HPDI32_IOCTL_REG_WRITE.....	62
<b>4. Operation.....</b>	<b>64</b>
4.1. Read and Write Operations.....	64

4.2. Data Transmission .....	64
4.3. Data Reception.....	64
4.4. Loop Back Operation.....	64
4.5. Data Transfer Options.....	64
4.5.1. PIO .....	64
4.5.2. Standard DMA.....	64
4.5.3. Demand Mode DMA.....	65
4.6. Interrupt Notification .....	65
4.7. Memory Mapped Resources .....	67
<b>Document History .....</b>	<b>68</b>

## 1. Introduction

This user manual applies to driver version 1.19, release 0. In addition, it applies to DIPHASE2 library version 1.00. The DIPHASE2 library is released as an add-on to the HPDI32 device driver. This user manual covers the DIPHASE2 library and DIPHASE2 centric portions of the device driver interface. Users must refer to the driver user manual for driver installation and driver usage information.

### 1.1. Purpose

The purpose of this document is to describe the interface to the DIPHASE2 Linux library and the DIPHASE2 centric portion of the HPDI32 Linux device driver. This software provides the interface between "Application Software" and the HPDI32A-DIPHASE2 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
Driver	Driver means the kernel mode device driver, which runs in the kernel space with kernel mode privileges.
Application	Application means the user mode process, which runs in the user space with user mode privileges.

### 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. The DIPHASE2 library is provided is a static library (`hpdi32_diphase2.a`) to be linked with user level applications. With the driver and the DIPHASE2 library, user applications are able to open and close a HPDI32A-DIPHASE2 device and, while open, perform read, write and I/O control operations.

### 1.5. Hardware Overview

The HPDI32A-DIPHASE2 is a high-performance DIPHASE encoded serial I/O interface board. The host side connection is 32-bit PCI based. The external I/O interface consists of a high speed DIPHASE encoded serial data stream. The board is capable of transmitting or receiving data at 4.0 megabits per second over the external I/O interface. Onboard transmit and receive FIFOs of up to 128k 32-bit data words each, buffer transfer data between the PCI bus and the cable interface. This allows the HPDI32 to maintain maximum bursts on the cable interface (at least up to the depth of the FIFOs) independent of the PCI bus interface. The onboard FIFOs can also be used to buffer data between the cable interface and the PCI bus to maintain a sustained data throughput for real-time applications.

The HPDI32 offers a half-duplex external I/O interface. The board can either transmit or receive data, but it cannot do both simultaneously. In addition to the DIPHASE data I/O lines, the external interface includes a set of control signals along with 32-bits of discrete general purpose I/O. The board accommodates transmitting and receiving data in messages of both preprogrammed and unrestricted lengths. This includes sending or receiving relatively small blocks of data on demand, or 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.

## 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 *HPDI32A-DIPHASE2 User Manual* from General Standards Corporation.
- The *PCI9080 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 and library are designed to operate with Linux kernel versions 2.2, 2.4 and 2.6 running on a PC system with one or more Intel x86 processors. This release of the driver was tested under the below listed kernels.

Kernel	Distribution	X86	
		32-bit	64-bit
2.6.21	Red Hat Fedora Core 7	Yes	Yes
2.6.18	SUSE 10.2	Yes	Yes
2.6.18	Red Hat Fedora Core 6	Yes	Yes
2.6.16	SUSE 10.1	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 Enterprise Linux Workstation Release 4	Yes	Yes
2.6.9	Red Hat Fedora Core 3	Yes	Yes
2.4.21	Red Hat Enterprise Linux Workstation Release 3	Yes	
2.4.20	Red Hat Linux 9	Yes	
2.4.18	Red Hat Linux 8.0	Yes	
2.4.18	Red Hat Linux 7.3	Yes	
2.4.7	Red Hat Linux 7.2	Yes	
2.2.14	Red Hat Linux 6.2	Yes	

**NOTE:** The driver may have to be rebuilt before being used due to kernel version differences between the GSC build host and the customer's target host.

**NOTE:** The library may have to be rebuilt before being used due to kernel version differences between the GSC build host and the customer's target host.

**NOTE:** The library has not been tested on an SMP host.

### 2.2. The /proc File System

While the driver is installed, the text file `/proc/hpdi32` can be read to obtain information about the driver and installed HPDI32 boards. 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: 1.19
built: Jul 30 2007, 09:42:52
boards: 1
types: 32
models: DIPHASE2
```

Entry	Description
version	This gives the driver version number in the form <code>x.xx</code> .
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> .
boards	This identifies the total number of boards the driver detected.
types	This gives a comma separated list of the types of boards installed. 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. A type of "64" identifies a board with a 64-bit PCI interface. A type of "32"

	identifies a board with a 32-bit PCI interface.
models	This gives a comma separated list of the model of boards installed. The list will contain “boards” number of entries. The order corresponds to the device node indexes and minor numbers as given in the /dev directory. A type of “DIPHASE2” identifies an HPDI32A-DIPHASE2 board, while a “*” identifies a generic HPDI32 board.

### 2.3. The Driver

This document pertains to the DIPHASE2 library. For information on building and using the driver, which is also necessary for DIPHASE2 applications, please refer to the driver user manual.

### 2.4. File List

The DIPHASE2 library consists of the below listed files. The archive is described in detail in following subsections. The driver is released as a separate archive.

File	Description
hpdi32.diphase2.tar.gz	This archive contains the library and all related sources.
hpdi32_diphase2_linux_library_um.doc	This is a PDF version of this user manual.

### 2.5. Installation

Install the driver according to the instructions given in the generic HPDI32 Linux driver user manual. Install the DIPHASE2 library and its related files following the below listed steps. This includes the library, the documentation source code, and the sample applications.

1. Change the current directory to the parent HPDI32 driver release directory. The default may typically be `/usr/src/linux/drivers`, though this may vary among distributions and kernel versions.
2. Copy the archive file `hpdi32.diphase2.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/diphase2` in the current directory, and then copies all of the archive’s files into this new directory.

```
tar -xzvf hpdi32.diphase2.tar.gz
```

### 2.6. Removal

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

1. Change to the directory where the library archive was installed. This should be `/usr/src/linux/drivers`. (The path name may vary among distributions and kernel versions.)
2. Issue the below command to remove the library archive and all of the installed library files.

```
rm -rf hpdi32.diphase2.tar.gz hpdi32/diphase2
```

3. To remove the driver and its related sources as well, refer to the generic HPDI32 user manual.

## 2.7. Overall Make Script

A make script is included in the DIPHASE2 installation directory. Executing this script will perform a make for all the DIPHASE2 build targets included in the release. The script is named `diphase2_make` and is located in the `hpdi32/diphase2` directory.

## 2.8. The Library

This library and its related files are contained in the archive file `hpdi32.diphase2.tar.gz`. The archive's library files are listed below. The library is located in the directory `hpdi32/diphase2/lib`. The paragraphs that follow give instructions for building and using the library.

File	Description
<code>*.c</code>	The library source files.
<code>hpdi32_diphase2.a</code>	A pre-built statically linkable library.
<code>hpdi32_diphase2.h</code>	The library header file. This header should be included by DIPHASE2 applications.
<code>makefile</code>	The library make file.
<code>makefile.dep</code>	An automatically generated make dependency file.

### 2.8.1. Build

Follow the below steps to build the library.

1. Change to the directory where the library and its sources were installed. This should be `/usr/src/linux/drivers/hpdi32/diphase2/lib`.
2. Remove all existing build targets by issuing the below command.

```
make -f makefile clean
```

3. Build the library by issuing the below command.

```
make -f makefile release
```

**NOTE:** The “release” target is a debug-free version of the library. To build a debug version of the library specify “debug” as the build target or leave the build target as unspecified.

### 2.8.2. Version

The library version number can be obtained at runtime. Refer to the library function `hpdi32_dp2_lib_version_get()` for complete details (see page 54).

### 2.8.3. Using the Library

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_diphase2.h` in each module referencing a library or driver component. Second, expand the include file search path to search the directory where the library header is located. This should be `/usr/src/linux/drivers/hpdi32/diphase2/lib`. Link time use also has two requires. First, include the static library `hpdi32_diphase2.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. This should also be `/usr/src/linux/drivers/hpdi32/diphase2/lib`.

**NOTE:** DIPHASE2 applications must include the DIPHASE2 header `hpdi32_diphase2.h` rather than the generic HPDI32 header. If applications include the generic HPDI32 header before including the DIPHASE2 header, then the DIPHASE2 functionality will be unavailable.

## 2.9. Document Source Code Examples

The documentation sources are divided between the library archive and the driver archive. The library archive file (`hpdi32.diphase2.tar.gz`) contains all of the DIPHASE2 specific source code examples included in this document. The DIPHASE2 specific examples are distinguishable from the driver examples in that the DIPHASE2 examples include the term “`_dp2_`” in the function names. The samples from the driver archive do not include this term. In addition, the DIPHASE2 document sources are included as a statically linkable library usable with DIPHASE2 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. The archive content is not described here, though its use is described in the following paragraphs. These files are installed into the directory `/usr/src/linux/drivers/hpdi32/diphase2/docsrc`. Refer to the generic driver user manual for information and instructions pertaining to the non-DIPHASE2 specific documentation source code library.

File	Description
<code>*.c</code>	These are the C source files.
<code>hpdi32_dp2_dsl.a</code>	This is a pre-built, linkable version of the library.
<code>hpdi32_dp2_dsl.h</code>	This is the library header file.
<code>makefile</code>	This is the library make file.
<code>makefile.dep</code>	This is an automatically generated make dependency file.

### 2.9.1. Build

Follow the below steps to compile the example files.

1. Change to the directory where the source code example files were installed. This should be `hpdi32/diphase2/docsrc`.
2. Remove all existing build targets by issuing the below command.

```
make -f makefile clean
```

3. Compile the sample files by issuing the below command.

```
make -f makefile release
```

**NOTE:** The “`release`” target is a debug-free version of the library. To build a debug version of the library specify “`debug`” as the build target or leave the build target as unspecified.

### 2.9.2. 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_dp2_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. This should be `/usr/src/linux/drivers/hpdi32/diphase2/docsrc`. Link time use also has two requires. First, include the static library `hpdi32_dp2_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. This should also be `/usr/src/linux/drivers/hpdi32/diphase2/docsrc`.

## 2.10. Sample Applications

The samples referenced here are DIPHASE2 specific. These samples will not function properly with non-DIPHASE2 versions of the HPDI32. Likewise, the sample applications included with the generic driver release will not function properly with HPDI32A-DIPHASE2 boards.

### 2.10.1. lbtest

**CAUTION:** When using the sample application the HPDI32 and any externally attached equipment may be damaged if the HPDI32's external interface has a cable attached. Damage may result because the sample application drives various external output signals. No damage will result if no cable at all is attached.

This sample application performs an automated loop back data transmission and reception test of a user specified DIPHASE2 board. It can be used as the starting point for application development on top of the DIPHASE2 library and the HPDI32 Linux device driver. The application includes the below listed files.

File	Description
*.c	These are the main source files.
lbtest	This is the pre-built sample application.
main.h	This is the application's header file.
makefile	This is a make file.
makefiel.dep	This is a make dependency file.
../utils/*.c	These are DIPHASE2 specific utility sources used by the application.
../utils/*.h	These are the DIPHASE2 specific headers for the utility sources used by the application.
../../utils/*.c	These are generic HPDI32 utility sources used by the application.
../../utils/*.h	These are the headers for the generic HPDI32 utility sources used by the application.

#### 2.10.1.1. Build

Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/hpdi32/diphase2/lbtest`.
2. Remove all existing build targets by issuing the below command.

```
make -f makefile clean
```

3. Build the driver by issuing the below command.

```
make -f makefile debug
```

**NOTE:** The “debug” target is a debug version of the application. To build a debug-free version specify “release” as the build target. The pre-built application included in the library archive is the debug-free version.

#### 2.10.1.2. Execute

**CAUTION:** Damage may occur to the HPDI32 or any attached equipment if either the cable or the attached equipment are not compatible with the HPDI32. Damage may result because the sample application drives various external output signals. No damage will result if no cable at all is attached.

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application was installed.
2. Start the sample application by issuing the command given below. The application uses the command line arguments given to direct its course of action. The arguments are described in the table below. The application will repeat the loop back test as called for by the arguments and will report the accumulated test results. A single iteration should take no more than about 15 seconds.

```
./lbttest <-c> <-C> <-m#> <-n#> <board>
```

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 “#” interactions, where “#” is a decimal number.
board	This is the zero based index of the board to access. This is ignored when only a single board is present. If multiple boards are present and this argument is not given, then the application will prompt the user for the board to access.

### 2.10.2. sbtest

**CAUTION:** When using the sample application the HPDI32 and any externally attached equipment may be damaged if the HPDI32’s external interface has a cable attached. Damage may result because the sample application drives various external output signals. No damage will result if no cable at all is attached.

This sample application provides a command line driven Linux application aimed at testing the DIPHASE2 library and some minimal functionality of a user specified DIPHASE2 board. It can be used as the starting point for application development on top of the DIPHASE2 library and the HPDI32 Linux device driver. The application performs an automated test of some library features. The application includes the below listed files.

File	Description
*.c	These are the primary application source files.
main.h	This is the application’s header file.
makefile	This is a make file.
makefiel.dep	This is a make dependency file.
sbtest	This is the pre-built sample application.
../utils/*.c	These are DIPHASE2 specific utility sources used by the application.
../utils/*.h	These are the DIPHASE2 specific headers for the utility sources used by the application.
../../utils/*.c	These are generic HPDI32 utility sources used by the application.
../../utils/*.h	These are the headers for the generic HPDI32 utility sources used by the application.

#### 2.10.2.1. Build

Follow the below steps to build/rebuild the sample application.

1. Change to the directory where the sample application was installed. This should be `/usr/src/linux/drivers/hpdi32/diphase2/sbtest`.
2. Remove all existing build targets by issuing the below command.

```
make -f makefile clean
```

3. Build the driver by issuing the below command.

```
make -f makefile debug
```

**NOTE:** The “debug” target is a debug version of the application. To build a debug-free version specify “release” as the build target. The pre-built application included in the library archive is the debug-free version.

#### 2.10.2.2. Execute

**CAUTION:** Damage may occur to the HPDI32 or any attached equipment if either the cable or the attached equipment are not compatible with the HPDI32. Damage may result because the sample application drives various external output signals. No damage will result if no cable at all is attached.

Follow the below steps to execute the sample application.

1. Change to the directory where the sample application was installed.
2. Start the sample application by issuing the command given below. The application uses the command line arguments given to direct its course of action. Once started the application will automatically execute a series of tests to verify the operation of the library and the board. The application will repeat the test cycle as called for by the arguments and will report the accumulated test results. The arguments are described in the table below. A single test cycle should take just a few seconds to complete.

```
./sbttest <-c> <-C> <-m#> <-n#> <board>
```

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 “#” interactions, where “#” is a decimal number.
board	This is the zero based index of the board to access. This is ignored when only a single board is present. If multiple boards are present and this argument is not given, then the application will prompt the user for the board to access.

### 3. Application Interface

The application interface includes components specific to the DIPHASE2 library and some from the device driver. The interface documented here is applicable to DIPHASE2 boards. For generic HPDI32 interface information refer to the generic driver user manual. The 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 the GSC HPDI32 board for Linux applications. The DIPHASE2 centric interface includes various macros, data types and functions, all of which are described in the following paragraphs. Those portions that are DIPHASE2 specific are defined in the header `hpdi32_diphase2.h`. This header also provides access to the non-DIPHASE2 specific portions of the interface. The HPDI32 headers define numerous items in addition to those described here.

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

**NOTE:** DIPHASE2 application must include the header file `hpdi32_diphase2.h` rather than the HPDI32 driver header file.

#### 3.1. Macros

The DIPHASE2 interface includes the following macros, which are defined via the DIPHASE2 header and the driver header. The headers also contains various other utility type macros, which are provided without documentation.

##### 3.1.1. Interrupt Identification Bits: GSC Specific

This set of macros defines the possible GSC specific interrupt identification bits used by the interrupt notification feature and by the interrupt control, status and configuration registers. These bits refer to interrupt sources implemented in the HPDI32 firmware and which are directly accessible to applications. Individual bits may be bit-wise or'd together in any desired combination. The macros are named for individual interrupt sources and their default configurations. When the Interrupt Configuration registers are present (IELR and IHLR) the condition associated with the interrupt source may be changed. When used with the `HPDI32_IOCTL_INT_NOTIFY` and `HPDI32_IOCTL_INT_STATUS` IOCTL services, these bits are used in the `gsc` fields of the `hpdi32_interrupt_t` structure.

Macros	Description
<code>HPDI32_INT_GSC_CC2_1</code>	This refers to a logic level one on cable signal Cable Command 2.
<code>HPDI32_INT_GSC_CC4_1</code>	This refers to a logic level one on cable signal Cable Command 4.
<code>HPDI32_INT_GSC_CC5_1</code>	This refers to a logic level one on cable signal Cable Command 5.
<code>HPDI32_INT_GSC_CC6_1</code>	This refers to a logic level one on cable signal Cable Command 6.
<code>HPDI32_INT_GSC_RX_BUSY</code>	This refers to the condition where the receiver is busy receiving data.
<code>HPDI32_INT_GSC_RX_DONE</code>	This refers to the Receive Done condition at the end of a message.
<code>HPDI32_INT_GSC_RX_FIFO_AE</code>	This refers to the Rx FIFO Almost Empty status.
<code>HPDI32_INT_GSC_RX_FIFO_AF</code>	This refers to the Rx FIFO Almost Full status.
<code>HPDI32_INT_GSC_RX_FIFO_EMPTY</code>	This refers to the Rx FIFO Empty status.
<code>HPDI32_INT_GSC_RX_FIFO_FULL</code>	This refers to the Rx FIFO Full status.
<code>HPDI32_INT_GSC_RX_FIFO_OVER</code>	This refers to an Rx FIFO overrun condition. This occurs when data is written to the Rx FIFO, but the FIFO is full. The data written is lost.
<code>HPDI32_INT_GSC_RX_FIFO_UNDER</code>	This refers to an Rx FIFO under run condition. This occurs when data is read from the Rx FIFO, but the FIFO is empty. The data read is invalid.
<code>HPDI32_INT_GSC_RX_RUNNING</code>	This refers to the condition where the receiver is running, essentially meaning it has been enabled.
<code>HPDI32_INT_GSC_TX_BUSY</code>	This refers to the condition where the transmitter is busy sending

	data.
HPDI32_INT_GSC_TX_DATA_1	This refers to any logic level one Tx Data bit.
HPDI32_INT_GSC_TX_DATA_FALL	This refers to any falling edge on the Tx Data signal.
HPDI32_INT_GSC_TX_DATA_RISE	This refers to any rising edge on the Tx Data signal.
HPDI32_INT_GSC_TX_DONE	This refers to the Transmit Done condition at the end of a message.
HPDI32_INT_GSC_TX_DRIVE	This refers to the condition where the transmitter is driving cable signals, essentially meaning it has been enabled.
HPDI32_INT_GSC_TX_FIFO_AE	This refers to the Tx FIFO Almost Empty status.
HPDI32_INT_GSC_TX_FIFO_AF	This refers to the Tx FIFO Almost Full status.
HPDI32_INT_GSC_TX_FIFO_EMPTY	This refers to the Tx FIFO Empty status.
HPDI32_INT_GSC_TX_FIFO_FULL	This refers to the Tx FIFO Full status.
HPDI32_INT_GSC_TX_FIFO_OVER	This refers to a Tx FIFO overrun condition. This occurs when data is written to the Tx FIFO, but the FIFO is full. The data written is lost.

### 3.1.2. Interrupt Identification Bits: Non-GSC Specific

This set of macros defines the possible non-GSC specific interrupt identification bits used by the interrupt notification feature. These bits refer to interrupt sources not associated with the HPDI32 firmware and which are not directly accessible to applications. Individual bits may be bit-wise or'd together in any desired combination. With the HPDI32\_IOCTL\_INT\_NOTIFY and HPDI32\_IOCTL\_INT\_STATUS IOCTL services, these bits are used in the two fields named other in the hpdi32\_interrupt\_t structure.

Macros	Description
HPDI32_INT_ANY_OTHER	This refers to any interrupt not specifically named in this list.
HPDI32_INT_DMA_0_DONE	This refers to the DMA channel 0 done interrupt.
HPDI32_INT_DMA_1_DONE	This refers to the DMA channel 1 done interrupt.
HPDI32_INT_PCI	This essentially refers to all interrupts since all HPDI32 interrupts are implemented by being channeled through to the board's PCI interrupt line.

**NOTE:** Each DMA based read/write request may result in multiple DMA Done interrupts. This occurs because the driver breaks read/write requests into individual transfers based on the size of the corresponding transfer buffer.

### 3.1.3. IOCTL

The IOCTL macros are documented following the function call descriptions.

### 3.1.4. I/O PIO Threshold Options

This set of macros defines the predefined I/O PIO Threshold options used by the HPDI32\_IOCTL\_IO\_CONFIG IOCTL service. The values apply to the pio\_threshold fields of the hpdi32\_io\_config\_t structure. Values other than the ones listed are permissible. For efficiency purposes the driver will automatically revert to PIO for suitably small I/O request. The PIO Threshold specifies the level at which does this. If a DMA or DMDMA request is made for PIO Threshold or fewer samples, then the request is performed using PIO.

Macros	Description
HPDI32_IO_PIO_THRESHOLD_DEFAULT	This is the default value.
HPDI32_IO_PIO_THRESHOLD_MIN	This is the minimum threshold value, which is zero. This effectively disables the feature of reverting to PIO transfers.
HPDI32_IO_PIO_THRESHOLD_NO_CHANGE	This is used to specify that the current mode not be changed.

### 3.1.5. I/O Data Sizes

This set of macros defines the possible I/O data size options used by the HPDI32\_IOCTL\_IO\_CONFIG IOCTL service. The options specify the application's desired data size as 8-bits, 16-bits or 32-bits. This parameter refers to the size of the data transferred between the host and the FIFOs only. This has no direct affect on the board's cable interface which is always 32-bits wide. The values apply to the `sample_bits` fields of the `hpdi32_io_config_t` structure.

Macros	Description
HPDI32_IO_SAMPLE_BITS_8	This specifies 8-bit data.
HPDI32_IO_SAMPLE_BITS_16	This specifies 16-bit data.
HPDI32_IO_SAMPLE_BITS_32	This specifies 32-bit data.
HPDI32_IO_SAMPLE_BITS_DEFAULT	This is the default, which is 32-bit data.
HPDI32_IO_SAMPLE_BITS_NO_CHANGE	This is used to specify that the current size not be changed.

### 3.1.6. I/O Transfer Modes

This set of macros defines the possible I/O transfer mode options used by the HPDI32\_IOCTL\_IO\_CONFIG IOCTL service. The modes define the driver's means of transferring data between the host and the HPDI32. The values apply to the `mode` fields of the `hpdi32_io_config_t` structure.

Macros	Description
HPDI32_IO_MODE_DEFAULT	This is the default mode assigned each time the device is opened.
HPDI32_IO_MODE_DMA	Perform reads and writes using standard DMA.
HPDI32_IO_MODE_DMDMA	Perform reads and writes using Demand Mode DMA.
HPDI32_IO_MODE_NO_CHANGE	This is used to specify that the current mode not be changed.
HPDI32_IO_MODE_PIO	Perform reads and writes using PIO.

**NOTE:** Writes using the DMDMA option must be performed with the Tx Enable and Tx Start bits set in the Board Control Register.

**NOTE:** Reads using the DMDMA option must be performed with the Rx Enable bit set in the Board Control Register.

### 3.1.7. I/O Transfer Buffer Size

This set of macros gives the predefined I/O transfer buffer size options used by the HPDI32\_IOCTL\_IO\_CONFIG IOCTL service. These buffers are used for temporary read/write data storage and are a common artifact of the HPDI32 Linux device driver. The values apply to the `samples` fields of the `hpdi32_io_config_t` structure. All values between the minimum and maximum can also be used.

Macros	Description
HPDI32_IO_SAMPLES_DEFAULT	This is the default size assigned each time the device is opened.
HPDI32_IO_SAMPLES_MAX	This is the maximum size of the transfer buffer.
HPDI32_IO_SAMPLES_MIN	This is the minimum size of the transfer buffer.
HPDI32_IO_SAMPLES_NO_CHANGE	This is used to specify that the current size not be changed.

**NOTE:** As of version 1.14 the driver's I/O buffer allocation mechanism has been updated. This update produced two changes. The first is that the upper allocation size limit has been increased from 128K bytes to 2M bytes. The second change is that the size of the buffers granted may be larger or smaller than requested, depending on available resources and the kernel's allocation granularity.

### 3.1.8. I/O Timeouts

This set of macros gives the predefined I/O timeout options used by the `HPDI32_IOCTL_IO_CONFIG` IOCTL service. The timeout period is specified in seconds. The values apply to the `timeout` fields of the `hpdi32_io_config_t` structure. All values between the minimum and maximum can also be used. A value of zero (0) specifies that the request be performed without waiting for additional FIFO data/space to become available.

Macros	Description
<code>HPDI32_IO_TIMEOUT_DEFAULT</code>	This is the default timeout assigned each time the device is opened.
<code>HPDI32_IO_TIMEOUT_MAX</code>	This specifies the maximum timeout period.
<code>HPDI32_IO_TIMEOUT_MIN</code>	This specifies the minimum timeout period.
<code>HPDI32_IO_TIMEOUT_NO_CHANGE</code>	This is used to specify that the current timeout not be changed.

### 3.1.9. mmap() Register Decoding

This set of macros can be used when directly accessing individual registers relative to the GSC register block pointer obtained from the `mmap()` service. Each macro takes an `HPDI32_GSC_XXX` register macro as its single argument.

Macros	Description
<code>HPDI32_REG_OFFSET(reg)</code>	This returns a register's offset in bytes from the block's base address.
<code>HPDI32_REG_SIZE(reg)</code>	This returns a register's size in bytes. Values are one (1), two (2) or four (4).

### 3.1.10. Registers

The following tables give the complete set of HPDI32A-DIPHASE2 registers. The tables are divided by register categories. Unless otherwise stated, all registers are accessed by their native size of eight, 16 or 32-bits. The only exception is the PCICCR register, which is 24-bits wide but accessed as if it were 32-bits wide. In this instance the upper eight-bits are to be ignored. Register values are passed as 32-bit entities and bits outside the register's native size are ignored.

#### 3.1.10.1. GSC Registers

The following table gives the complete set of GSC specific HPDI32 registers. For detailed definitions of these registers refer to the *HPDI32A-DIPHASE2 User Manual*.

Macros	Description
<code>HPDI32_GSC_BCR</code>	Board Control Register (BCR)
<code>HPDI32_GSC_BSR</code>	Board Status Register (BSR)
<code>HPDI32_GSC_FDR</code>	FIFO Data Register (FDR)
<code>HPDI32_GSC_FRR</code>	Firmware Revision Register (FRR)
<code>HPDI32_GSC_ICR</code>	Interrupt Control Register (ICR)
<code>HPDI32_GSC_IELR</code>	Interrupt Edge/Level Register (IELR)
<code>HPDI32_GSC_IHLR</code>	Interrupt High/Low Register (IHLR)
<code>HPDI32_GSC_ISR</code>	Interrupt Status Register (ISR)
<code>HPDI32_GSC_RAR</code>	Rx Almost Register (RAR)
<code>HPDI32_GSC_RXIDPR</code>	Rx Input Data Port Register (RXIDPR) The hardware manual may have previously referred to this as the Rx Status Block Length Counter Register.
<code>HPDI32_GSC_RXMLSR</code>	Rx Message Length Status Register (RXMLSR) The hardware manual may have previously referred to this as the Rx Row Length Counter Register.
<code>HPDI32_GSC_TAR</code>	Tx Almost Register (TAR)
<code>HPDI32_GSC_TXMLCR</code>	Tx Message Length Control Register (TXMLCR) The hardware manual may have previously referred to this as the Tx Row Valid Length Register.
<code>HPDI32_GSC_TXMLSR</code>	Tx Message Length Status Register (TXMLSR) The hardware manual may have previously referred to this as the Tx Status Length Register.

HPDI32_GSC_TXODPR	Tx Output Data Port Register (TXODPR)
HPDI32_GSC_TXPDLR	Tx Pre-Drive Length Register (TXPDLR) The hardware manual may have previously referred to this as the Tx Row Invalid Length Register.

### 3.1.10.2. PCI Configuration Registers

The following table gives the set of PCI configuration registers. For detailed definitions of these registers refer to the *PCI9080 Data Book*.

Macros	Description
HPDI32_PCI_BAR0	PCI Base Address Register for Memory Accesses to Local, Runtime, and DMA Registers (PCIBAR0)
HPDI32_PCI_BAR1	PCI Base Address Register for I/O Accesses to Local, Runtime, and DMA Registers (PCIBAR1)
HPDI32_PCI_BAR2	PCI Base Address Register for Memory Accesses to Local Address Space 0 (PCIBAR2)
HPDI32_PCI_BAR3	PCI Base Address Register for Memory Accesses to Local Address Space 1 (PCIBAR3)
HPDI32_PCI_BAR4	Unused Base Address (PCIBAR4)
HPDI32_PCI_BAR5	Unused Base Address (PCIBAR5)
HPDI32_PCI_BISTR	PCI Built-In Self Test Register (PCIBISTR)
HPDI32_PCI_CCR	PCI Class Code Register (PCICCR)
HPDI32_PCI_CIS	PCI Cardbus CIS Pointer Register (PCICIS)
HPDI32_PCI_CLSR	PCI Cache Line Size Register (PCICLSR)
HPDI32_PCI_CR	PCI Command Register (PCICR)
HPDI32_PCI_ERBAR	PCI Expansion ROM Base Address (PCIERBAR)
HPDI32_PCI_HTR	PCI Header Type Register (PCIHTR)
HPDI32_PCI_IDR	PCI Configuration ID Register (PCIIDR)
HPDI32_PCI_ILR	PCI Interrupt Line Register (PCIILR)
HPDI32_PCI_IPR	PCI Interrupt Pin Register (PCIIPR)
HPDI32_PCI_LTR	PCI Latency Timer Register (PCILTR)
HPDI32_PCI_MGR	PCI Min_Gnt Register (PCIMGR)
HPDI32_PCI_MLR	PCI Max_Lat Register (PCIMLR)
HPDI32_PCI_REV	PCI Revision ID Register (PCIREV)
HPDI32_PCI_SID	PCI Subsystem ID Register (PCISID)
HPDI32_PCI_SR	PCI Status Register (PCISR)
HPDI32_PCI_SVID	PCI Subsystem Vendor ID Register (PCISVID)

**NOTE:** A PCIIDR value of 0x908010B5 identifies the PCI interface chip as a PLX PCI9080, which is a 32-bit PCI bridge chip. A PCISVID value of 0x10B5 identifies that the PCISID register is assigned by PLX. A PCISID value of 0x2400 identifies 32-bit versions of the HPDI32. Refer to the GSC Firmware Revision Register for additional identification information.

### 3.1.10.3. PLX PCI9080 Feature Set Registers

The following table gives the complete set of PLX PCI9080 feature set registers. For detailed definitions of these registers refer to the *PCI9080 Data Book*.

#### Local Configuration Registers

Macros	Description
HPDI32_PLX_BIGEND	Big/Little Endian Descriptor Register (BIGEND)
HPDI32_PLX_DMCFGA	PCI Configuration Address Register for Direct Master to PCI IO/CFG (DMCFGA)
HPDI32_PLX_DMLBAI	Local Bus Base Address Register for Direct Master to PCI IO/CFG (DMLBAI)

HPDI32_PLX_DMLBAM	Local Bus Base Address Register for Direct Master to PCI Memory (DMLBAM)
HPDI32_PLX_DMPBAM	PCI Base Address Register for Direct Master to PCI Memory (DMPBAM)
HPDI32_PLX_DMRR	Local Range Register for Direct Master to PCI (DMRR)
HPDI32_PLX_EROMBA	Expansion ROM Local Base Address Register (EROMBA)
HPDI32_PLX_EROMRR	Expansion ROM Range Register (EROMRR)
HPDI32_PLX_LAS0BA	Local Address Space 0 Local Base Address Register (LAS0BA)
HPDI32_PLX_LAS0RR	Local Address Space 0 Range Register for PCI-to-Local Bus (LAS0RR)
HPDI32_PLX_LAS1BA	Local Address Space 1 Local Base Address Register (LAS1BA)
HPDI32_PLX_LAS1RR	Local Address Space 1 Range Register for PCI-to-Local Bus (LAS1RR)
HPDI32_PLX_LBRD0	Local Address Space 0/Expansion ROM Bus Region Descriptor Register (LBRD0)
HPDI32_PLX_LBRD1	Local Address Space 1 Bus Region Descriptor Register (LBRD1)
HPDI32_PLX_MARBR	Mode Arbitration Register (MARBR)

### Runtime Registers

Macros	Description
HPDI32_PLX_CNTRL	Serial EEPROM Control, CPI Command Codes, User I/O, Init Control Register (CNTRL)
HPDI32_PLX_INTCSR	Interrupt Control/Status Register (INTCSR)
HPDI32_PLX_L2PDBELL	Local-to-PCI Doorbell Register (L2PDBELL)
HPDI32_PLX_MBOX0	Mailbox Register 0 (MBOX0)
HPDI32_PLX_MBOX1	Mailbox Register 1 (MBOX1)
HPDI32_PLX_MBOX2	Mailbox Register 2 (MBOX2)
HPDI32_PLX_MBOX3	Mailbox Register 3 (MBOX3)
HPDI32_PLX_MBOX4	Mailbox Register 4 (MBOX4)
HPDI32_PLX_MBOX5	Mailbox Register 5 (MBOX5)
HPDI32_PLX_MBOX6	Mailbox Register 6 (MBOX6)
HPDI32_PLX_MBOX7	Mailbox Register 7 (MBOX7)
HPDI32_PLX_P2LDBELL	PCI-to-Local Doorbell Register (P2LDBELL)
HPDI32_PLX_PCIHIDR	PCI Permanent Configuration ID Register (PCIHIDR)
HPDI32_PLX_PCIHREV	PCI Permanent Revision ID Register (PCIHREV)

### DMA Registers

Macros	Description
HPDI32_PLX_DMAARB	DMA Arbitration Register (DMAARB)
HPDI32_PLX_DMACSR0	DMA Channel 0 Command/Status Register (DMACSR0)
HPDI32_PLX_DMACSR1	DMA Channel 1 Command/Status Register (DMACSR1)
HPDI32_PLX_DMADPR0	DMA Channel 0 Descriptor Pointer Register (DMADPR0)
HPDI32_PLX_DMADPR1	DMA Channel 1 Descriptor Pointer Register (DMADPR1)
HPDI32_PLX_DMALADR0	DMA Channel 0 Local Address Register (DMALADR0)
HPDI32_PLX_DMALADR1	DMA Channel 1 Local Address Register (DMALADR1)
HPDI32_PLX_DMAMODE0	DMA Channel 0 Mode Register (DMAMODE0)
HPDI32_PLX_DMAMODE1	DMA Channel 1 Mode Register (DMAMODE1)
HPDI32_PLX_DMAPADR0	DMA Channel 0 PCI Address Register (DMAPADR0)
HPDI32_PLX_DMAPADR1	DMA Channel 1 PCI Address Register (DMAPADR1)
HPDI32_PLX_DMASIZ0	DMA Channel 0 Transfer Size Register (DMASIZ0)
HPDI32_PLX_DMASIZ1	DMA Channel 1 Transfer Size Register (DMASIZ1)
HPDI32_PLX_DMATHR	DMA Threshold Register (DMATHR)

### Message Queue Registers

Macros	Description
HPDI32_PLX_IFHPR	Inbound Free Head Pointer Register (IFHPR)

HPDI32_PLX_IFTP	Inbound Free Tail Pointer Register (IFTP)
HPDI32_PLX_IPHP	Inbound Post Head Pointer Register (IPHP)
HPDI32_PLX_IPTP	Inbound Post Tail Pointer Register (IPTP)
HPDI32_PLX_IQP	Inbound Queue Port Register (IQP)
HPDI32_PLX_MQCR	Messaging Queue Configuration Register (MQCR)
HPDI32_PLX_OFHP	Outbound Free Head Pointer Register (OFHP)
HPDI32_PLX_OFTP	Outbound Free Tail Pointer Register (OFTP)
HPDI32_PLX_OPHP	Outbound Post Head Pointer Register (OPHP)
HPDI32_PLX_OPLFIM	Outbound Post List FIFO Interrupt Mask Register (OPLFIM)
HPDI32_PLX_OPLFIS	Outbound Post List FIFO Interrupt Status Register (OPLFIS)
HPDI32_PLX_OPTP	Outbound Post Tail Pointer Register (OPTP)
HPDI32_PLX_OQP	Outbound Queue Port Register (OQP)
HPDI32_PLX_QBAR	Queue Base Address Register (QBAR)
HPDI32_PLX_QSR	Queue Status/Control Register (QSR)

## 3.2. Data Types

This driver interface includes the following data types.

### 3.2.1. hpdi32\_debug\_data\_t

This structure is used for driver debugging purposes and is used only during driver development.

#### Definition

```
typedef struct
{
    __u32    device[32];
    __u32    driver[32];
} hpdi32_debug_data_t;
```

Fields	Description
device	This array gives debug data specific to the device.
driver	This array gives debug data common to all devices.

### 3.2.2. hpdi32\_driver\_info\_t

This structure defines the data fields for the information returned by the HPDI32\_IOCTL\_DRIVER\_INFO\_GET IOCTL service.

#### Definition

```
typedef struct
{
    __s8    version[8];
    __s8    built[32];
} hpdi32_driver_info_t;
```

Fields	Description
version	This field gives the driver version number as a string in the form of X.XX.
built	This field gives the driver build date and time as a string. It is given in the C form of printf("%s, %s", __DATE__, __TIME__).

### 3.2.3. hpdi32\_interrupt\_t

This structure defines the interrupt notification fields used by the HPDI32\_IOCTL\_INT\_NOTIFY and HPDI32\_IOCTL\_INT\_STATUS IOCTL services. Read the details of the individual services for additional information.

#### Definition

```
typedef struct
{
    struct
    {
        __u32    gsc;
        __u32    other;
    } notify;

    struct
    {
        __u32    gsc;
        __u32    other;
    } status;
} hpdi32_interrupt_t;
```

Fields	Description
notify	These fields specify the interrupts for which notification is desired. If a bit is set, then notification is desired. If clear then notification is not desired.
notify.gsc	This field identifies the GSC specific interrupts for which notification is desired.
notify.other	This field identifies the non-GSC specific interrupts for which notification is desired.
status	For notification requests these fields identify the requested interrupts which are not in use by the driver and which are now under application control. For status requests these fields report which of the requested interrupts have occurred.
status.gsc	This field gives information on the GSC specific interrupts.
status.other	This field gives information on the non-GSC specific interrupts.

### 3.2.4. hpdi32\_io\_config\_t

This structure defines the I/O configuration data fields use by the HPDI32\_IOCTL\_IO\_CONFIG IOCTL service.

#### Definition

```
typedef struct
{
    struct
    {
        __s32    mode;
        __s32    timeout;
        __u32    samples;
        __s32    sample_bits;
        __s32    pio_threshold;
    } read;

    struct
    {
        __s32    mode;
    }
}
```

```

    __s32  timeout;
    __u32  samples;
    __s32  sample_bits;
    __s32  pio_threshold;
} write;
} hpdi32_io_config_t;

```

Fields	Description
read	This structure contains the parameters referring to read() operations.
read.mode	This field specifies the desired operating mode (PIO, DMA, ...).
read.timeout	This field specifies the desired timeout period in seconds.
read.samples	This field specifies the desired size of the transfer buffer in samples
read.sample_bits	This field specifies the size of the read data samples in bits. Use the predefined macros HPDI32_IO_SAMPLE_BITS_8/16/32. *
read.pio_threshold	This field specifies the threshold limit at which a read request will automatically revert to PIO mode. If a DMA or DMDMA read requests this number or fewer samples, then it will be performed using PIO mode.
write	This structure contains the parameters referring to write() operations.
write.mode	This field specifies the desired operating mode (PIO, DMA, ...).
write.timeout	This field specifies the desired timeout period in seconds.
write.samples	This field specifies the desired size of the transfer buffer in samples
write.sample_bits	This field specifies the size of the write data samples in bits. Use the predefined macros HPDI32_IO_SAMPLE_BITS_8/16/32. *
write.pio_threshold	This field specifies the threshold limit at which a write request will automatically revert to PIO mode. If a DMA or DMDMA write requests this number or fewer samples, then it will be performed using PIO mode.

\* Data value bit D0 is always aligned with cable data signal D0. Also, this configuration setting only affects data transfers between the host and FIFOs. This setting has no direct affect on the cable interface.

**NOTE:** A transfer buffer size cannot be changed while that buffer is mapped via mmap(). An mmap() call must be accompanied by a corresponding munmap() call in order to change the buffer's size. However, the read buffer's size, for example, can be changed while the write buffer and/or the GSC register block is mapped.

**NOTE:** As of version 1.14 the driver's I/O buffer allocation mechanism has been updated. This update produced two changes. The first is that the upper allocation size limit has been increased from 128K bytes to 2M bytes. The second change is that the size of the buffers granted may be larger or smaller than requested, depending on available resources and the kernel's allocation granularity.

### 3.2.5. hpdi32\_mmap\_mem\_t

This structure is used as part of the driver's mmap implementation, which gives applications direct access to various driver resources. The structure gives the mmap() and munmap() parameters for individually accessible memory areas.

#### Definition

```

typedef struct
{
    __u32  offset;
    __u32  size;
} hpdi32_mmap_mem_t;

```

Fields	Description
offset	This is the value to pass as the <code>mmap()</code> <code>offset</code> parameter.
size	This is the value to pass as the <code>mmap()</code> and <code>munmap()</code> <code>length</code> parameters.

### 3.2.6. `hpdi32_mmap_t`

This structure is used as part of the driver's `mmap` implementation, which gives applications direct access to various driver resources. The structure gives the `mmap()` and `munmap()` parameters for each of the accessible memory areas.

#### Definition

```
typedef struct
{
    hpdi32_mmap_mem_t    regs;
    hpdi32_mmap_mem_t    read;
    hpdi32_mmap_mem_t    write;
} hpdi32_mmap_t;
```

Fields	Description
regs	This structure defines the <code>mmap</code> parameters required for accessing the GSC specific registers.
regs.offset	This is the value to pass as the <code>mmap()</code> <code>offset</code> parameter.
regs.size	This is the value to pass as the <code>mmap()</code> and <code>munmap()</code> <code>length</code> parameters.
read	This structure defines the <code>mmap</code> parameters required for accessing driver's read transfer buffer.
read.offset	This is the value to pass as the <code>mmap()</code> <code>offset</code> parameter.
read.size	This is the value to pass as the <code>mmap()</code> and <code>munmap()</code> <code>length</code> parameters.
write	This structure defines the <code>mmap</code> parameters required for accessing driver's write transfer buffer.
write.offset	This is the value to pass as the <code>mmap()</code> <code>offset</code> parameter.
write.size	This is the value to pass as the <code>mmap()</code> and <code>munmap()</code> <code>length</code> parameters.

**NOTE:** On some systems access of the GSC registers block via `mmap()` is unsupported. This occurs mostly with embedded motherboards because the BIOS tries to conserve resources when mapping PCI device memory and I/O regions into the processor's address space. Use of the `mmap()` feature to access the GSC registers requires that the memory region be placed on a boundary the size of the processors physical memory page. In cases where this does not occur the driver will set the field `regs.size` to zero (0) to reflect that the registers are not `mmap()` accessible. Attempts to use `mmap()` when this does occur may return a NULL pointer.

### 3.2.7. `hpdi32_reg_t`

This structure defines the data fields for the information involved in the register access IOCTL services. Read the details of the individual services for additional information.

#### Definition

```
typedef struct
{
    __u32    reg;
    __u32    value;
    __u32    mask;
} hpdi32_reg_t;
```

Fields	Description
reg	This field identifies the register to be accessed.
value	This field identifies the value retrieved by read operations and the value to apply by write operations.
mask	This field identifies the register bits from the <code>value</code> field that are to be applied during the read-modify-write IOCTL service. If a bit is set in the mask, then the corresponding <code>value</code> bit is applied to the register. If a mask bit is not set then the corresponding register bit is left unchanged.

### 3.3. Driver Functions

This driver interface includes the following functions.

#### 3.3.1. close()

This function is the entry point to close a connection to an open HPDI32 board. This function should only be called after a successful open of the respective device.

Prototype

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

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

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

Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_close(int fd, int verbose)
{
    int status;

    status = close(fd);

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

    return(status);
}
```

#### 3.3.2. 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 a following section. All IOCTL requests for a given device are synchronized and are completed in the order received.

### Prototype

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

Argument	Description
fd	This is the file descriptor of the device to access.
request	This specifies the desired operation to be performed.
...	This is any additional arguments. If request 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 errno.
0	The operation succeeded.

### Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_ioctl(int fd, int request, void *arg, int verbose)
{
    int status;

    status = ioctl(fd, request, arg);

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

    return(status);
}
```

### 3.3.3. mmap()

This function is the entry point to gaining direct access to various driver resources. This feature maps driver memory resources into application space and gives the application a pointer by which it can access each mapped resource. Using this service an application can directly read and write GSC specific registers, write directly to the driver's write() transfer buffer and read directly from the driver's read() transfer buffer. This function should only be called after a successful open of the respective device. All mmap() requests are synchronized with read(), write() and ioctl() requests and are completed in the order received.

**WARNING:** Attempts to access outside a mapped memory area will result in segmentation faults. Attempts to access a resource after the area has been unmapped will also result in segmentation faults.

**NOTE:** On some systems access of the GSC registers block via mmap() is unsupported. This occurs mostly with embedded motherboards because the BIOS tries to conserve resources when mapping PCI device memory and I/O regions into the processor's address space. Use of the

`mmap()` feature to access the GSC registers requires that the memory region be placed on a boundary the size of the processors physical memory page. In cases where this does not occur the driver will set the field `regs.size` to zero (0) to reflect that the registers are not `mmap()` accessible. Attempts to use `mmap()` when this does occur may return a NULL pointer.

Prototype

```
void* mmap(
    void* start,
    size_t length,
    int prot,
    int flags,
    int fd,
    off_t offset);
```

Argument	Description
start	This is the desired starting address, which should be zero (0).
length	This is the desired length of the mapped memory resource. The value passed must be the value returned by the HPDI32_IOCTL_MMAP_INFO IOCTL service in the <code>size</code> field of the appropriate <code>hpdi32_mmap_mem_t</code> structure.
prot	This is the desired access and should equal "PROT_READ   PROT_WRITE".
flags	This should be the value "MAP_SHARED".
fd	This is the file descriptor returned by the <code>open()</code> function.
offset	This is the desired offset into the driver's mapped resources. The value passed must be the value returned by the HPDI32_IOCTL_MMAP_INFO IOCTL service in the <code>offset</code> field of the appropriate <code>hpdi32_mmap_mem_t</code> structure.

Return Value	Description
MAP_FAILURE	An error occurred. Consult <code>errno</code> .
else	The operation succeeded. This is a pointer that is used to directly access the mapped resource.

Example

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

#include "HPDI32DocSrcLib.h"

void* hpdi32_mmap(int fd, const hpdi32_mmap_mem_t* mem, int verbose)
{
    void* vp;

    vp = mmap( 0,
               mem->size,
               PROT_READ | PROT_WRITE,
               MAP_SHARED,
               fd,
               mem->offset);

    if ((verbose) && (vp == MAP_FAILED))
    {
        vp = NULL;
        printf("mmap() failure, errno = %d\n", errno);
    }
}
```

```

    }
    return(vp);
}

```

### 3.3.4. munmap()

This function is the entry point to releasing a memory resource previously mapped via `mmap()`. This function should only be called after a successful open of the respective device, and then only after a successful `mmap()` request. The `munmap()` request that releases the last mapped resource is synchronized with `read()`, `write()` and `ioctl()` services and are completed in the order received.

**WARNING:** Attempts to access a resource after it has been released will result in segmentation faults.

#### Prototype

```
int munmap(void* start, size_t length);
```

Argument	Description
start	This is the address of the area to be released. This should be the pointer returned by the corresponding <code>mmap()</code> call.
length	This is the size of the area to be released. The value passed should be the value returned by the <code>HPDI32_IOCTL_MMAP_INFO</code> IOCTL service in the <code>size</code> field of the appropriate <code>hpdi32_mmap_mem_t</code> structure.

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

#### Example

```

#include <errno.h>
#include <stdio.h>
#include <sys/mman.h>

#include "HPDI32DocSrcLib.h"

int hpdi32_munmap(void* vp, const hpdi32_mmap_mem_t* mem, int verbose)
{
    int status;

    status = munmap(vp, mem->size);

    if ((verbose) && (status == -1))
        printf("munmap() failure, errno = %d\n", errno);

    return(status);
}

```

### 3.3.5. open()

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

## 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 "HPDI32DocSrcLib.h"

int hpdi32_open(unsigned int board, int verbose)
{
    int    fd;
    char   name[80];

    sprintf(name, "/dev/hpdi32%u", board);
    fd = open(name, O_RDWR);

    if ((verbose) && (fd == -1))
        printf("open() failure on %s, errno = %d\n", name, errno);

    return(fd);
}
```

### 3.3.6. read()

This function is the entry point to reading received 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 receive FIFO. The data transfer parameters are controlled via the `HPDI32_IOCTL_IO_CONFIG` IOCTL service. Read requests can be made which exceed the size of the driver's read transfer buffer. When this occurs the driver breaks the overall request into smaller requests that do not exceed this buffer's size. Using the `HPDI32_IOCTL_IO_CONFIG` IOCTL service an application can vary the size of this buffer to optimize overall throughput. All read requests for a given device are synchronized and are completed in the order received.

## 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. If this pointer is NULL, then the data will be read into the

	read transfer buffer and will not be copied to application buffers. If the read transfer buffer is mapped using the <code>mmap( )</code> feature, then this must be NULL.
<code>count</code>	This is the desired number of bytes to read. This must be a multiple of four (4). If the <code>buf</code> argument is NULL, then this value is quietly limited to the size of the read transfer buffer.

Return Value	Description
-1	An error occurred. Consult <code>errno</code> .
0 to <code>count</code>	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 when the receive FIFO became empty during the request.

**NOTE:** The size of the read transfer buffer can be configured by an application via the `HPDI32_IOCTL_IO_CONFIG` IOCTL service.

**NOTE:** For standard DMA based read requests it is the application's responsibility to insure that the amount of data requested does not exceed the amount of data available. When requests exceed the amount of available data, the excess data returned will be indeterminate. The Rx FIFO Under Run will be recorded in the Board Status Register, if the Rx Under Run feature is supported in firmware.

### Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_read(int fd, __u32 *buf, size_t samples, int verbose)
{
    size_t bytes;
    int status;

    bytes = samples * 4;
    status = read(fd, buf, bytes);

    if (status >= 0)
        status /= 4;
    else if (verbose)
        printf("read() failure, errno = %d\n", errno);

    return(status);
}
```

### 3.3.7. write()

This function is the entry point to writing transmit 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 transmit FIFO. The data transfer parameters are controlled via the `HPDI32_IOCTL_IO_CONFIG` IOCTL service. Write requests can be made which exceed the size of the driver's write transfer buffer. When this occurs the driver breaks the overall request into smaller requests that do not exceed this buffer's size. Using the `HPDI32_IOCTL_IO_CONFIG` IOCTL service an application can vary the size of this buffer to optimize overall throughput. All write requests for a given device are synchronized and are completed in the order received.

## 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 four (4). If the <code>buf</code> argument is NULL, then this value is quietly limited to the size of the write transfer buffer.

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 when the transmit FIFO became full during the request.

**NOTE:** The size of the write transfer buffer can be configured by an application via the `HPDI32_IOCTL_IO_CONFIG` IOCTL service.

**NOTE:** For standard DMA based write requests it is the application's responsibility to insure that the amount of data submitted does not exceed the amount of FIFO space available. When requests exceed the amount of available space, the excess data is lost. The Tx FIFO Overrun will be recorded in the Board Status Register, if the Tx Overrun feature is supported in firmware.

## Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_write(int fd, const __u32 *buf, size_t samples, int verbose)
{
    size_t bytes;
    int status;

    bytes = samples * 4;
    status = write(fd, buf, bytes);

    if (status >= 0)
        status /= 4;
    else if (verbose)
        printf("write() failure, errno = %d\n", errno);

    return(status);
}
```

## 3.4. Library Functions

This DIPHASE2 library interface includes the following functions.

### 3.4.1. hpdi32\_dp2\_board\_loop\_back\_get()

This function is the entry point to requesting the current loop back enable/disable setting.

#### Prototype

```
int hpdi32_dp2_board_loop_back_get(int fd, int* enable);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	The enabled state is recorded here, if the pointer is not NULL. The value is zero (0) if loop back is disabled and one (1) if it is enabled.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the errno variable.

#### Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_loop_back_show(int fd, int verbose)
{
    int err;
    int lb;

    err = hpdi32_dp2_board_loop_back_get(fd, &lb);

    if (verbose == 0)
    {
    }
    else if (err)
    {
        printf("hpdi32_dp2_board_loop_back_get() failure");
        printf(", errno = %d\n", err);
    }
    else
    {
        printf("Loop Back: %s\n", lb ? "Enabled" : "Disabled");
    }

    return(err);
}
```

### 3.4.2. hpdi32\_dp2\_board\_loop\_back\_set()

This function is the entry point to updating the current loop back enable/disable setting.

#### Prototype

```
int hpdi32_dp2_board_loop_back_set(int fd, int enable);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	If zero (0) then loop back is disabled. If positive then loop back is enabled. If negative then the setting is not changed.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_loop_back_enable(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_board_loop_back_set(fd, 1);

    if ((verbose) && (err))
    {
        printf("hpdi32_dp2_board_loop_back_set() failure");
        printf(", errno = %d\n", err);
    }

    return(err);
}
```

**3.4.3. hpdi32\_dp2\_board\_reset()**

This function is the entry point to performing a complete board reset.

Prototype

```
int hpdi32_dp2_board_reset(int fd);
```

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

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_board_reset(int fd, int verbose)
{
    int err;
```

```

err = hpdi32_dp2_board_reset(fd);

if ((verbose) && (err))
    printf("hpdi32_dp2_board_reset() failure, errno = %d\n", err);

return(err);
}

```

### 3.4.4. hpdi32\_dp2\_gpio\_dir\_get()

This function is the entry point to retrieving the direction settings for the GPIO cable signals.

#### Prototype

```

int hpdi32_dp2_gpio_dir_get(
    int      fd,
    int*     out_hwhb,
    int*     out_hwlb,
    int*     out_lwhb,
    int*     out_lwlb);

```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
out_hwhb	The direction setting for the high word, high byte (D31-D24) is recorded here, if the pointer is non-NULL. The value is zero (0) if the port is configured as an input and one (1) if configured as an output.
out_hwlb	The direction setting for the high word, low byte (D23-D16) is recorded here, if the pointer is non-NULL. The value is zero (0) if the port is configured as an input and one (1) if configured as an output.
out_lwhb	The direction setting for the low word, high byte (D15-D8) is recorded here, if the pointer is non-NULL. The value is zero (0) if the port is configured as an input and one (1) if configured as an output.
out_lwlb	The direction setting for the low word, low byte (D7-D0) is recorded here, if the pointer is non-NULL. The value is zero (0) if the port is configured as an input and one (1) if configured as an output.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

#### Example

```

#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_gpio_dir_show(int fd, int verbose)
{
    int err;
    int hwhb;
    int hwlb;
    int lwhb;
    int lwlb;
}

```

```

err = hpdi32_dp2_gpio_dir_get(fd, &hwhb, &hwlb, &lwhb, &lwlb);

if (verbose == 0)
{
}
else if (err)
{
    printf("hpdi32_dp2_gpio_dir_get() failure, errno = %d\n", err);
}
else
{
    printf("GPIO Direction:\n");
    printf("  D31-D24: %s\n", hwhb ? "Output" : "Input");
    printf("  D23-D16: %s\n", hwlb ? "Output" : "Input");
    printf("  D15-D08: %s\n", lwhb ? "Output" : "Input");
    printf("  D07-D00: %s\n", lwlb ? "Output" : "Input");
}

return(err);
}
    
```

### 3.4.5. hpdi32\_dp2\_gpio\_dir\_set()

This function is the entry point to updating the direction settings for the GPIO cable signals.

#### Prototype

```

int hpdi32_dp2_gpio_dir_set(
    int fd,
    int out_hwhb,
    int out_hwlb,
    int out_lwhb,
    int out_lwlb);
    
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
out_hwhb	This is the desired direction setting for the high word, high byte (D31-D24). A positive value configures the port as an output. A zero value (0) configures the port as an input. A negative value causes the setting to remain as is.
out_hwlb	This is the desired direction setting for the high word, low byte (D23-D16). A positive value configures the port as an output. A zero value (0) configures the port as an input. A negative value causes the setting to remain as is.
out_lwhb	This is the desired direction setting for the low word, high byte (D15-D8). A positive value configures the port as an output. A zero value (0) configures the port as an input. A negative value causes the setting to remain as is.
out_lwlb	This is the desired direction setting for the low word, low byte (D7-D0). A positive value configures the port as an output. A zero value (0) configures the port as an input. A negative value causes the setting to remain as is.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_gpio_dir_init(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_gpio_dir_set(fd, 1, 1, 1, 1);

    if ((err) && (verbose))
        printf("hpdi32_dp2_gpio_dir_set() failure, errno = %d\n", err);

    return(err);
}
```

**3.4.6. hpdi32\_dp2\_gpio\_mod()**

This function is the entry point to performing a read-modify-write of the GPIO output data. Only the referenced output latch bits will be modified. Only those configured as outputs will affect the cable output signals.

Prototype

```
int hpdi32_dp2_gpio_mod(int fd, __u32 value, __u32 mask);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
value	This is the value to be applied.
mask	This is the set of value bits that are to be modified. If a mask bit is set, then the corresponding value bit is applied. Otherwise the corresponding value bit is ignored.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_gpio_lwlb_low(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_gpio_mod(fd, 0x000000FF, 0x00000000);

    if ((verbose) && (err))
        printf("hpdi32_dp2_gpio_mod() failure, errno = %d\n", err);

    return(err);
}
```

### 3.4.7. hpdi32\_dp2\_gpio\_read()

This function is the entry point to reading the current cable GPIO signal states.

Prototype

```
int hpdi32_dp2_gpio_read(int fd, __u32* value);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
value	The current signal state for all 32-signals is recorded here, if the pointer is non-NULL.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_gpio_rx(int fd, __u32* value, int verbose)
{
    int err;

    err = hpdi32_dp2_gpio_read(fd, value);

    if ((verbose) && (err))
        printf("hpdi32_dp2_gpio_read() failure, errno = %d\n", err);

    return(err);
}
```

### 3.4.8. hpdi32\_dp2\_gpio\_write()

This function is the entry point to updating the latched outputs for all 32 GPIO output latches. Only those configured as outputs will affect the cable output signals.

Prototype

```
int hpdi32_dp2_gpio_write(int fd, __u32 value);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
value	This is the value to apply to the output latches.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

**Example**

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_gpio_tx(int fd, __u32 value, int verbose)
{
    int err;

    err = hpdi32_dp2_gpio_write(fd, value);

    if ((verbose) && (err))
        printf("hpdi32_dp2_gpio_write() failure, errno = %d\n", err);

    return(err);
}
```

**3.4.9. hpdi32\_dp2\_predrive\_get()**

This function is the entry point to retrieving the current Pre-Drive configuration, which is how the transmitter drives the data signal before sending a message.

**Prototype**

```
int hpdi32_dp2_predrive_get(
    int    fd,
    int*   enable,
    int*   high,
    int*   count);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	The current enable state is recorded here, if the pointer is non-NULL. The value is zero (0) if Pre-Drive is disabled and one (1) if it is enabled.
high	The current level state is recorded here, if the pointer is non-NULL. The value is zero (0) if the level is set to <i>low</i> and one (1) if set to <i>high</i> .
count	The current period count is recorded here, if the pointer is non-NULL. The count refers to increments of 31.25 nanoseconds.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

**Example**

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_predrive_show(int fd, int verbose)
{
    int count;
    int err;
```

```

int enable;
int high;

err = hpdi32_dp2_predrive_get(fd, &enable, &high, &count);

if (verbose == 0)
{
}
else if (err)
{
    printf("hpdi32_dp2_predrive_get() failure, errno = %d\n", err);
}
else
{
    printf("PreDrive Configuration:\n");
    printf("  Enabled: %s\n", enable ? "Yes" : "No");
    printf("  Level:   %s\n", high ? "High" : "Low");
    printf("  Period:  %0.3fns\n", (float) 31.25 * count);
}

return(err);
}

```

### 3.4.10. hpdi32\_dp2\_predrive\_set()

This function is the entry point to updating the current Pre-Drive configuration, which is how the transmitter drives the data signal before sending a message.

#### Prototype

```
int hpdi32_dp2_predrive_set(int fd, int enable, int high, int count);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	This enables or disables Pre-Drive operation. If the value is zero (0) then Pre-Drive is disabled. If the value is positive then Pre-Drive is enabled. If the value is negative, then the enable state is not changed.
high	This adjusts the Pre-Drive level when enabled. If the value is zero (0) then the data signal is driven low during the Pre-Drive period. If the value is positive then the data signal is driven high during the Pre-Drive period. If the value is negative, then the drive level is not changed.
count	This specifies the Pre-Drive period in increments of 31.25 nanoseconds. If the count is zero, then Pre-Drive is disabled (and the enable argument's value is ignored) and the recorded period is unaltered. If the value is from one (1) to 0xFFFF, then the Pre-Drive period is set accordingly. If the value exceeds 0xFFFF then an error is reported and no settings are applied. If the value is negative, then the Pre-Drive period is not changed.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

#### Example

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

```
#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_predrive_init(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_predrive_set(fd, 1, 0, 240);

    if ((verbose) && (err))
        printf("hpdi32_dp2_predrive_set() failure, errno = %d\n", err);

    return(err);
}
```

### 3.4.11. hpdi32\_dp2\_reg\_mod()

This function is the entry point to performing a read-modify-write on an application specified register. Only the GSC firmware registers can be modified. The PCI and PLX registers are read-only.

#### Prototype

```
int hpdi32_dp2_reg_mod(int fd, __u32 reg, __u32 value, __u32 mask);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
reg	This is the register to be accessed.
value	This is the value to be applied.
mask	This is the set of value bits that are to be modified. If a mask bit is set, then the corresponding value bit is applied. Otherwise the corresponding value bit is ignored.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the errno variable.

#### Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_tx_start(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_reg_mod(    fd,
                                HPDI32_GSC_BCR,
                                ~0L,
                                HPDI32_GSC_BCR_TX_START);

    if ((verbose) && (err))
        printf("hpdi32_dp2_reg_mod() failure, errno = %d\n", err);

    return(err);
}
```

}

### 3.4.12. hpdi32\_dp2\_reg\_read()

This function is the entry point to reading the current value of a register. All device registers can be read.

Prototype

```
int hpdi32_dp2_reg_read(int fd, __u32 reg, __u32* value);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
reg	This is the identifier for the register to access.
value	The current register value is recorded here, if the pointer is non-NULL.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the errno variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_tx_started(int fd, int verbose)
{
    __u32    bcr;
    int      err;

    err = hpdi32_dp2_reg_read(fd, HPDI32_GSC_BCR, &bcr);

    if (verbose == 0)
        ;
    else if (err)
        printf("hpdi32_dp2_reg_read() failure, errno = %d\n", err);
    else if (bcr & HPDI32_GSC_BCR_TX_START)
        printf("Tx Started: Yes\n");
    else
        printf("Tx Started: No\n");

    return(err);
}
```

### 3.4.13. hpdi32\_dp2\_reg\_write()

This function is the entry point to writing a value to a register. Only the GSC firmware registers can be modified. The PCI and PLX registers are read-only.

Prototype

```
int hpdi32_dp2_reg_write(int fd, __u32 reg, __u32 value);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
reg	This is the identifier for the register to access.
value	This is the value to write to the specified register.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_bcr_clear(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_reg_write(fd, HPDI32_GSC_BCR, 0);

    if ((verbose) && (err))
        printf("hpdi32_dp2_reg_write() failure, errno = %d\n", err);

    return(err);
}
```

**3.4.14. hpdi32\_dp2\_rx\_accum\_msg\_size()**

This function is the entry point to retrieving the recorded accumulated number of bytes received. Refer to the hardware user manual for information on when this count is reset.

Prototype

```
int hpdi32_dp2_rx_accum_msg_size(int fd, __u32* size);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
size	The current count is recorded here, if the pointer is non-NULL.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_rx_msg_size_show(int fd, int verbose)
{
    int    err;
    __u32 size;
```

```

err = hpdi32_dp2_rx_accum_msg_size(fd, &size);

if (verbose == 0)
{
}
else if (err)
{
    printf( "hpdi32_dp2_rx_accum_msg_size() failure, errno = %d\n",
            err);
}
else
{
    printf( "Accumulated Rx Message Size: %lu Bytes\n",
            (long) size);
}

return(err);
}

```

### 3.4.15. hpdi32\_dp2\_rx\_config\_get()

This function is the entry point to retrieving the current receiver configuration.

Prototype

```

int hpdi32_dp2_rx_config_get(
    int      fd,
    int*     enable,
    int*     big_e,
    int*     hdr_size,
    int*     all_data);

```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	The receiver's enable/disable state is recorded here, if the pointer is non-NULL. The value is zero (0) if the receiver is disabled and one (1) if it is enabled.
big_e	The receiver's Endianness setting is recorded here, if the pointer is non-NULL. The value is zero (0) for Little Endian operation and one (1) for Big Endian operation.
hdr_size	The receiver's header Size setting is recorded here, if the pointer is non-NULL. The value is zero (0) for 10 bytes headers and one (1) for 14 bytes headers.
all_data	The receiver's message reception setting is recorded here, if the pointer is non-NULL. The value is zero (0) if the receiver records data according to the message size encoded in the message, and one (1) if the receiver is to record all incoming data until the data signal goes idle.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

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

```

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_rx_config_show(int fd, int verbose)
{
    int all;
    int big;
    int enable;
    int err;
    int hdr;

    err = hpdi32_dp2_rx_config_get(fd, &enable, &big, &hdr, &all);

    if (verbose == 0)
    {
    }
    else if (err)
    {
        printf("hpdi32_dp2_rx_config_get() failure, errno = %d\n",
            err);
    }
    else
    {
        printf("Rx Configuration:\n");
        printf("  Enabled:      %s\n", enable ? "Yes" : "No");
        printf("  Endianness:  %s\n", big ? "Big" : "Little");
        printf("  Header Size: %d bytes\n", hdr ? 14 : 10);
        printf("  All Data:    %s\n", all ? "Yes" : "No");
    }

    return(err);
}

```

### 3.4.16. hpdi32\_dp2\_rx\_config\_set()

This function is the entry point to updating the current receiver configuration.

#### Prototype

```

int hpdi32_dp2_rx_config_set(
    int fd,
    int enable,
    int big_e,
    int hdr_size,
    int all_data);

```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	This is the desired enable/disable setting. A positive value enables the receiver. A value of zero (0) disables the receiver. The setting is unchanged when the value is negative.
big_e	This is the desired Endianness setting. A positive value selects Big Endian operation. A value of zero (0) selects Little Endian operation. The setting is unchanged when the value is negative.
hdr_size	This is the desired header size setting. A positive value selects 14 byte headers. A value of zero (0) selects 10 byte headers. The setting is unchanged when the value is negative.
all_data	This is the desired data recording operation setting. For a positive value the receiver

	records all incoming data until the data signal goes idle. For a value of zero (0) the receiver record data according to the message size encoded in the received data. The setting is unchanged when the value is negative.
--	--

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

**Example**

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_rx_config_reset(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_rx_config_set(fd, 0, 0, 0, 0);

    if ((verbose) && (err))
    {
        printf( "hpdi32_dp2_rx_config_set() failure, errno = %d\n",
                err);
    }

    return(err);
}
```

**3.4.17. hpdi32\_dp2\_rx\_fifo\_almost\_get()**

This function is the entry point to retrieving the recorded Rx FIFO Almost Full and Almost Empty levels.

**Prototype**

```
int hpdi32_dp2_rx_fifo_almost_get(int fd, int* af, int* ae);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
af	The current Almost Full level is recorded here, if the pointer is non-NULL.
ae	The current Almost Empty level is recorded here, if the pointer is non-NULL.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

**Example**

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_rx_fifo_almost_show(int fd, int verbose)
{
```

```

int ae;
int af;
int err;

err = hpdi32_dp2_rx_fifo_almost_get(fd, &af, &ae);

if (verbose == 0)
{
}
else if (err)
{
    printf("hpdi32_dp2_rx_fifo_almost_get() failure");
    printf(", errno = %d\n", err);
}
else
{
    printf("Rx FIFO Almost Levels:\n");
    printf("  Almost Full:  %d\n", af);
    printf("  Almost Empty: %d\n", ae);
}

return(err);
}

```

### 3.4.18. hpdi32\_dp2\_rx\_fifo\_almost\_set()

This function is the entry point to updating the Rx FIFO Almost Full and Almost Empty levels. The Rx FIFO is reset when one or both values are modified, and the current Rx FIFO content is lost. If neither value is set, then the Rx FIFO is not reset.

#### Prototype

```
int hpdi32_dp2_rx_fifo_almost_set(int fd, int af, int ae);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
af	This is the desired Almost Full level. If the value is negative, then the setting is not changed. If the value is over 0xFFFF, then an error reported and no changes are applied. Any value from zero (0) to 0xFFFF is applied and the Rx FIFO reset.
ae	This is the desired Almost Empty level. If the value is negative, then the setting is not changed. If the value is over 0xFFFF, then an error reported and no changes are applied. Any value from zero (0) to 0xFFFF is applied and the Rx FIFO reset.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

#### Example

```

#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_rx_fifo_almost_init(int fd, int verbose)

```

```

{
    int err;

    err = hpdi32_dp2_rx_fifo_almost_set(fd, 16, 16);

    if ((verbose) && (err))
    {
        printf("hpdi32_dp2_rx_fifo_almost_set() failure");
        printf(", errno = %d\n", err);
    }

    return(err);
}

```

### 3.4.19. hpdi32\_dp2\_rx\_fifo\_reset()

This function is the entry point to resetting the Rx FIFO. The current Rx FIFO content is lost when the FIFO is reset.

#### Prototype

```
int hpdi32_dp2_rx_fifo_reset(int fd);
```

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

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the errno variable.

#### Example

```

#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_rx_fifo_clear(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_rx_fifo_reset(fd);

    if ((verbose) && (err))
    {
        printf("hpdi32_dp2_rx_fifo_reset() failure, errno = %d\n",
            err);
    }

    return(err);
}

```

### 3.4.20. hpdi32\_dp2\_tx\_config\_get()

This function is the entry point to retrieving the current transmitter configuration.

Prototype

```
int hpdi32_dp2_tx_config_get(
    int    fd,
    int*   enable,
    int*   start,
    int*   big_e,
    int*   raw,
    __u32* length);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	The transmitter's enable/disable state is recorded here, if the pointer is non-NULL. The value is zero (0) if the transmitter is disabled and one (1) if it is enabled.
start	The transmitter's Tx Start state is recorded here, if the pointer is non-NULL. The value is zero (0) if Tx Start is clear and one (1) if it is set.
big_e	The transmitter's Endianness setting is recorded here, if the pointer is non-NULL. The value is zero (0) for Little Endian operation and one (1) for Big Endian operation.
raw	The transmitter's raw data transmission setting is recorded here, if the pointer is non-NULL. The value is zero (0) for sending encoded data (the default) and one (1) for sending raw, unencoded data.
length	The transmitter's message length setting is recorded here, if the pointer is non-NULL. The value reported is the actual recorded length, which is in 31.25 nanosecond increments.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the errno variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_tx_config_show(int fd, int verbose)
{
    int    big;
    int    enable;
    int    err;
    __u32  length;
    int    raw;
    int    start;

    err = hpdi32_dp2_tx_config_get( fd,
                                    &enable,
                                    &start,
                                    &big,
                                    &raw,
                                    &length);

    if (verbose == 0)
    {
    }
    else if (err)
    {
```

```

        printf("hpdi32_dp2_t_config_get() failure, errno = %d\n", err);
    }
    else
    {
        printf("Tx Configuration:\n");
        printf("  Enabled:      %s\n", enable ? "Yes" : "No");
        printf("  Tx Start:     %s\n", start ? "Set" : "Clear");
        printf("  Endianness:  %s\n", big ? "Big" : "Little");
        printf("  Raw Output:  %s\n", raw ? "Yes" : "No (unencoded)");
        printf("  Msg length:  %d\n", length);
    }

    return(err);
}

```

### 3.4.21. hpdi32\_dp2\_tx\_config\_set()

This function is the entry point to updating the current transmitter configuration.

Prototype

```

int hpdi32_dp2_tx_config_set(
    int    fd,
    int    enable,
    int    start,
    int    big_e,
    int    raw,
    __u32  length);

```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
enable	This is the desired enable/disable state. A positive value enables the transmitter. A value of zero (0) disables the transmitter. A negative value leaves the setting unchanged.
start	This is the desired Tx Start setting. A positive value sets Tx Start. A value of zero (0) clears Tx Start. A negative value leaves the setting unchanged.
big_e	This is the desired Endianness setting. A positive value selects Big Endian operation. A value of zero (0) selects Little Endian operation. A negative value leaves the setting unchanged.
raw	This is the desired raw data transmission setting. A positive value selects raw, unencoded data transmission. A value of zero (0) selects encoded data transmission (the default). A negative value leaves the setting unchanged.
length	This is the desired message length in 16-bit words. A positive value is recorded as the desired message length. A value of zero (0) requests variable message lengths. A negative value leaves the setting unchanged.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```

#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

```

```
int hpdi32_dp2_dsl_tx_config_reset(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_tx_config_set(fd, 0, 0, 0, 0, 0);

    if ((verbose) && (err))
    {
        printf( "hpdi32_dp2_tx_config_set() failure, errno = %d\n",
                err);
    }

    return(err);
}
```

### 3.4.22. hpdi32\_dp2\_tx\_fifo\_almost\_get()

This function is the entry point to retrieving the recorded Tx FIFO Almost Full and Almost Empty levels.

#### Prototype

```
int hpdi32_dp2_tx_fifo_almost_get(int fd, int* af, int* ae);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
af	The current Almost Full level is recorded here, if the pointer is non-NULL.
ae	The current Almost Empty level is recorded here, if the pointer is non-NULL.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

#### Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_tx_fifo_almost_show(int fd, int verbose)
{
    int ae;
    int af;
    int err;

    err = hpdi32_dp2_tx_fifo_almost_get(fd, &af, &ae);

    if (verbose == 0)
    {
    }
    else if (err)
    {
        printf("hpdi32_dp2_tx_fifo_almost_get() failure");
        printf(", errno = %d\n", err);
    }
}
```

```

else
{
    printf("Tx FIFO Almost Levels:\n");
    printf("  Almost Full:  %d\n", af);
    printf("  Almost Empty: %d\n", ae);
}

return(err);
}

```

### 3.4.23. hpdi32\_dp2\_tx\_fifo\_almost\_set()

This function is the entry point to updating the Tx FIFO Almost Full and Almost Empty levels. The Tx FIFO is reset when one or both values are modified, and the current Tx FIFO content is lost. If neither value is set, then the Tx FIFO is not reset.

#### Prototype

```
int hpdi32_dp2_tx_fifo_almost_set(int fd, int af, int ae);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
af	This is the desired Almost Full level. If the value is negative, then the setting is not changed. If the value is over 0xFFFF, then an error reported and no changes are applied. Any value from zero (0) to 0xFFFF is applied and the Tx FIFO reset.
ae	This is the desired Almost Empty level. If the value is negative, then the setting is not changed. If the value is over 0xFFFF, then an error reported and no changes are applied. Any value from zero (0) to 0xFFFF is applied and the Tx FIFO reset.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the errno variable.

#### Example

```

#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_tx_fifo_almost_init(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_tx_fifo_almost_set(fd, 16, 16);

    if ((verbose) && (err))
    {
        printf("hpdi32_dp2_tx_fifo_almost_set() failure");
        printf(", errno = %d\n", err);
    }

    return(err);
}

```

### 3.4.24. hpdi32\_dp2\_tx\_fifo\_reset()

This function is the entry point to resetting the Tx FIFO. The current Tx FIFO content is lost when the FIFO is reset.

Prototype

```
int hpdi32_dp2_tx_fifo_reset(int fd);
```

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

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpdi32_dp2_dsl.h"

int hpdi32_dp2_dsl_tx_fifo_clear(int fd, int verbose)
{
    int err;

    err = hpdi32_dp2_tx_fifo_reset(fd);

    if ((verbose) && (err))
    {
        printf( "hpdi32_dp2_tx_fifo_reset() failure, errno = %d\n",
                err);
    }

    return(err);
}
```

### 3.4.25. hpdi32\_dp2\_tx\_last\_msg\_size()

This function is the entry point to retrieving the recorded number of bytes transmitted as part of the last/current transmit message. Refer to the hardware user manual for information on when this count stops and when it is reset.

Prototype

```
int hpdi32_dp2_tx_last_msg_size(int fd, __u32* size);
```

Argument	Description
fd	This is the file descriptor of the device to be accessed.
size	The current count is recorded here, if the pointer is non-NULL.

Return Value	Description
0	The operation succeeded.
else	An error occurred. The value returned is the value of the <code>errno</code> variable.

Example

```
#include <stdio.h>

#include "hpd32_dp2_dsl.h"

int hpd32_dp2_dsl_tx_last_msg_size_show(int fd, int verbose)
{
    int    err;
    __u32  size;

    err = hpd32_dp2_tx_last_msg_size(fd, &size);

    if (verbose == 0)
    {
    }
    else if (err)
    {
        printf( "hpd32_dp2_tx_last_msg_size() failure, errno = %d\n",
                err);
    }
    else
    {
        printf("Last Tx Message Size: %lu Bytes\n", (long) size);
    }

    return(err);
}
```

**3.4.26. hpd32\_dp2\_lib\_version\_get()**

This function is the entry point to retrieving the library’s version number and build time stamp.

Prototype

```
void hpd32_dp2_lib_version_get(
    char*    version,
    size_t   vsize,
    char*    built,
    size_t   bsize);
```

Argument	Description
version	This is a pointer to the buffer to receive the version number. This should be at least five bytes long.
vsize	This is the size of the above buffer.
built	This is a pointer to the buffer to build time stamp. This is reported as if by <code>sprintf(built, "%s, %s", __DATA__, __TIME__);</code>
bsize	This is the size of the above buffer.

Example

```
#include <stdio.h>

#include "hpd32_dp2_dsl.h"
```

```

void hpdi32_dp2_dsl_version_show(int verbose)
{
    char    built[64];
    char    version[8];

    hpdi32_dp2_lib_version_get( version,
                                sizeof(version),
                                built,
                                sizeof(built));

    if (verbose)
    {
        printf("HPDI32 DiPhase2 Library:\n");
        printf("  Version: %s\n", version);
        printf("  Built:   %s\n", built);
    }
}

```

### 3.5. 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`.

#### 3.5.1. HPDI32\_IOCTL\_DEBUG\_DATA\_GET

This service retrieves debug data produced by temporary code included in the driver only during driver debugging and development.

##### Usage

<code>ioctl()</code> Argument	Description
<code>request</code>	<code>HPDI32_IOCTL_DEBUG_DATA_GET</code>
<code>arg</code>	<code>hpdi32_debug_data_t*</code>

##### Example

```

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

#include "HPDI32DocSrcLib.h"

int hpdi32_debug_read(int fd, int verbose)
{
    hpdi32_debug_data_t debug;
    int                i;
    int                status;

    status = ioctl(fd, HPDI32_IOCTL_DEBUG_DATA_GET, &debug);

    if (!verbose)
    {

```

```

    }
    else if (status == -1)
    {
        printf("ioctl() failure, errno = %d\n", errno);
    }
    else
    {
        printf("Debug Data: device\n");

        for (i = 0; i < HPDI32_DEBUG_DATA_VALUES; i++)
        {
            printf( " %-2d. 0x%08lX\n",
                    i,
                    (unsigned long) debug.device[i]);
        }

        printf("Debug Data: driver\n");

        for (i = 0; i < HPDI32_DEBUG_DATA_VALUES; i++)
        {
            printf( " %-2d. 0x%08lX\n",
                    i,
                    (unsigned long) debug.driver[i]);
        }
    }

    return(status);
}

```

### 3.5.2. HPDI32\_IOCTL\_DRIVER\_INFO\_GET

This service retrieves information about the driver itself.

#### Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_DRIVER_INFO_GET
arg	hpdi32_driver_info_t*

#### Example

```

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

#include "HPDI32DocSrcLib.h"

int hpdi32_driver_info(int fd, hpdi32_driver_info_t* info, int verbose)
{
    int status;

    status = ioctl(fd, HPDI32_IOCTL_DRIVER_INFO_GET, info);

    if ((verbose) && (status == -1))
        printf("ioctl() failure, errno = %d\n", errno);
}

```

```

        return(status);
    }

```

### 3.5.3. HPDI32\_IOCTL\_INT\_NOTIFY

This service specifies the set of HPDI32 interrupts for which interrupt notification is desired. The `notify` fields of the referenced structure are used to specify those interrupts for which notification is desired. If the request is successful, then the `status` fields identify those interrupts which the application can configure and control. For these interrupts, the application is responsible for configuring and enabling them. The fields named `gsc` refer to GSC specific interrupt sources that are implemented in DIPHASE firmware. The fields named `other` refer to all other interrupt sources. At times an application may request notification of one or more GSC interrupts that are in use by the driver. When this occurs the respective bits in the `status.gsc` field will be clear. This can occur at times while the driver performs read and write operations. Since applications have direct access only to the GSC specific interrupt sources, the `status.other` field will always be zero. When a GSC interrupt occurs for which notification is requested, the driver will log, clear and disable the interrupt, then notify the application via a SIGIO signal. The application must then request the interrupt status (via the `HPDI32_IOCTL_INT_STATUS` IOCTL service) to determine which interrupt occurred. To continue further notification, the application must re-enable the respective interrupt. To end notification for an interrupt the application must submit another `HPDI32_IOCTL_INT_NOTIFY` IOCTL request with the interrupt bit clear.

**WARNING:** If an application modifies a driver configured interrupt, then data loss or corruption is probable and the application may cease to function properly.

**WARNING:** Applications should use the `HPDI32_IOCTL_REG_MOD` IOCTL service when manipulating the interrupt registers (ICR, IELR and IHLR) and must manipulate only those interrupts identified in the `status.gsc` field.

**WARNING:** The `HPDI32_INT_ANY_OTHER` interrupt bit was implemented for driver development purposes and should never occur. This effectively refers to any unexpected non-GSC specific interrupt. If such an interrupt does occur however, all interrupt on the board will be disabled. Interrupts can be enabled either by closing and reopening the device, or by any subsequent `HPDI32_IOCTL_INT_NOTIFY` IOCTL.

**NOTE:** An interrupt referenced in the `status.gsc` field becomes unavailable for use by the driver. When this occurs the driver will resort to less efficient means to accomplish the desired task. This can occur at times while the driver performs read and write operations.

**NOTE:** Requests can be made with any combination of interrupt options, including the `HPDI32_INT_PCI` and any or all others. When this particular bit is set with one or more others and one of the other respective interrupts occur, then only one SIGIO signal is posted even though two log bits are recorded.

#### Usage

<code>ioctl()</code> Argument	Description
<code>request</code>	<code>HPDI32_IOCTL_INT_NOTIFY</code>
<code>arg</code>	<code>hpdi32_interrupt_t*</code>

#### Example

```

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

```

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

#include "HPDI32DocSrcLib.h"

int hpdi32_int_notify(int fd, hpdi32_interrupt_t* arg, int verbose)
{
    int status;

    status = ioctl(fd, HPDI32_IOCTL_INT_NOTIFY, arg);

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

    return(status);
}
```

### 3.5.4. HPDI32\_IOCTL\_INT\_STATUS

This service requests the driver's accumulated interrupt notification status. If successful the `notify` fields of the referenced structure report the current notification settings while the `status` fields report the driver's accumulated notification status. The driver's accumulated status is then cleared.

#### Usage

<code>ioctl()</code> Argument	Description
<code>request</code>	<code>HPDI32_IOCTL_INT_STATUS</code>
<code>arg</code>	<code>hpdi32_interrupt_t*</code>

#### Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_int_status(int fd, hpdi32_interrupt_t* arg, int verbose)
{
    int status;

    status = ioctl(fd, HPDI32_IOCTL_INT_STATUS, arg);

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

    return(status);
}
```

### 3.5.5. HPDI32\_IOCTL\_IO\_CONFIG

This service updates and/or reads the configurable I/O parameters. Prior to the `ioctl()` call the structure fields specify the desired settings, which may include one or more of the `_NO_CHANGE` options when no respective changes are desired. Upon return from a successful call the structure fields reflect the current settings. This service is

synchronized with all active and pending read and write requests and will be performed when all such other requests have been completed.

The structure's `samples` fields specify the desired size of the read and write data transfer buffers. These buffers are required for intermediate data storage during read and write requests. The size of these buffers is minimal when the device is opened. For improved performance though applications can alter the buffer sizes according to application needs. For optimal throughput the buffers should be the size of the respective FIFOs. If the driver's memory allocation request fails, then the service request also fails and the previous settings remain is affect.

**NOTE:** The transfer buffer sizes cannot be changed while an `mmap()` resource is still in use. All `mmap()` calls must be accompanied by corresponding `munmap()` calls in order to change a buffer's size.

**NOTE:** As of version 1.14 the driver's I/O buffer allocation mechanism has been updated. This update produced two changes. The first is that the upper allocation size limit has been increased from 128K bytes to 2M bytes. The second change is that the size of the buffers granted may be larger or smaller than requested, depending on available resources and the kernel's allocation granularity.

### Usage

<b>ioctl()</b> Argument	<b>Description</b>
<code>request</code>	<code>HPDI32_IOCTL_IO_CONFIG</code>
<code>arg</code>	<code>hpdi32_io_config_t*</code>

### Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_io_config(int fd, hpdi32_io_config_t* config, int verbose)
{
    int status;

    status = ioctl(fd, HPDI32_IOCTL_IO_CONFIG, config);

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

    return(status);
}
```

### 3.5.6. HPDI32\_IOCTL\_MMAP\_INFO

This service retrieves the parameters required for subsequent `mmap()` and `munmap()` system calls. In order to use these system services, this IOCTL service must be issued after each `HPDI32_IOCTL_IO_CONFIG` IOCTL service, as that service's data affects data values returned by this service.

Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_MMAP_INFO
arg	hpdi32_mmap_t*

**NOTE:** On some systems access of the GSC registers block via `mmap()` is unsupported. This occurs mostly with embedded motherboards because the BIOS tries to conserve resources when mapping PCI device memory and I/O regions into the processor's address space. Use of the `mmap()` feature to access the GSC registers requires that the memory region be placed on a boundary the size of the processors physical memory page. In cases where this does not occur the driver will set the field `regs.size` to zero (0) to reflect that the registers are not `mmap()` accessible. Attempts to use `mmap()` when this does occur may return a NULL pointer.

Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_mmap_info(int fd, hpdi32_mmap_t* mem, int verbose)
{
    int status;

    status = ioctl(fd, HPDI32_IOCTL_MMAP_INFO, mem);

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

    return(status);
}
```

**3.5.7. HPDI32\_IOCTL\_NO\_COMMAND**

This is an empty driver entry point. This IOCTL may be given to verify that the driver is correctly installed and that an HPDI32 has been successfully opened. If an error status is returned then something isn't working properly.

Usage

<b>ioctl() Argument</b>	<b>Description</b>
request	HPDI32_IOCTL_NO_COMMAND
arg	Not used.

Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_no_command(int fd, int verbose)
```

```

{
    int status;

    status = ioctl(fd, HPDI32_IOCTL_NO_COMMAND);

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

    return(status);
}

```

### 3.5.8. HPDI32\_IOCTL\_REG\_MOD

This service performs a read-modify-write operation on an HPDI32 register. This includes only the GSC specific registers. All PCI and PLX PCI9080 feature set registers are read-only. Refer to `hpdi32.h` for a complete list of the accessible registers.

**NOTE:** Care should be exercised in using this service with registers that have set-to-clear bits. For such bits, the respective `mask` and `value` bits should be clear unless specifically attempting to clear the intended register bit.

#### Usage

<code>ioctl()</code> Argument	Description
<code>request</code>	<code>HPDI32_IOCTL_REG_MOD</code>
<code>arg</code>	<code>hpdi32_reg_t*</code>

#### Example

```

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

#include "HPDI32DocSrcLib.h"

int hpdi32_reg_mod(
    int      fd,
    __u32    reg,
    __u32    value,
    __u32    mask,
    int      verbose)
{
    hpdi32_reg_t    parm;
    int              status;

    parm.reg        = reg;
    parm.value      = value;
    parm.mask       = mask;
    status          = ioctl(fd, HPDI32_IOCTL_REG_MOD, &parm);

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

    return(status);
}

```

### 3.5.9. HPDI32\_IOCTL\_REG\_READ

This service reads the value of an HPDI32 register. This includes all PCI registers, all PLX PCI9080 feature set registers, and all GSC specific registers. Refer to `hpdi32.h` for a complete list of the accessible registers.

#### Usage

<code>ioctl()</code> Argument	Description
request	HPDI32_IOCTL_REG_READ
arg	<code>hpdi32_reg_t*</code>

#### Example

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

#include "HPDI32DocSrcLib.h"

int hpdi32_reg_read(int fd, __u32 reg, __u32* value, int verbose)
{
    hpdi32_reg_t    parm;
    int              status;

    parm.reg        = reg;
    parm.value      = 0xDEADBEEFL;
    parm.mask       = 0;    /* ignored for reads */
    status          = ioctl(fd, HPDI32_IOCTL_REG_READ, &parm);

    if (status >= 0)
        value[0]    = parm.value;
    else if (verbose)
        printf("ioctl() failure, errno = %d\n", errno);

    return(status);
}
```

### 3.5.10. HPDI32\_IOCTL\_REG\_WRITE

This service writes a value to an HPDI32 register. This includes only the GSC specific registers. All PCI and PLX PCI9080 feature set registers are read-only. Refer to `hpdi32.h` for a complete list of the accessible registers.

#### Usage

<code>ioctl()</code> Argument	Description
request	HPDI32_IOCTL_REG_WRITE
arg	<code>hpdi32_reg_t*</code>

#### Example

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

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

#include "HPDI32DocSrcLib.h"

int hpdi32_reg_write(int fd, __u32 reg, __u32 value, int verbose)
{
    hpdi32_reg_t    parm;
    int             status;

    parm.reg        = reg;
    parm.value      = value;
    parm.mask       = 0;    /* ignored for writes */
    status          = ioctl(fd, HPDI32_IOCTL_REG_WRITE, &parm);

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

    return(status);
}
```

## 4. Operation

This section explains some operational procedures using the driver and library. This is in no way intended to be a comprehensive guide on using the HPDI32A-DIPHASE2. This is simply to address a very few issues relating to the board's use.

### 4.1. Read and Write Operations

Before performing `read()` and `write()` requests the device I/O parameters should be configured via the `HPDI32_IOCTL_IO_CONFIG` IOCTL service. In addition, the transmit and receive FIFO Almost settings must be updated and the FIFOs reset, and the transmitter and receiver must be configured.

### 4.2. Data Transmission

The transmitter must be enabled and started for data to be transmitted over the cable interface. These requirements vary when merely writing data to the Tx FIFO however, according to the current I/O mode. When using the PIO or DMA modes, the transmitter need not be enabled or started so long as the Tx FIFO has room to accommodate the write request. In all other cases, either when using DMDMA or when the Tx FIFO is too full to contain the additional data, then the transmitter must be enabled and started before the write request is initiated. After transmission of a message completes, the transmitter must be stopped and disabled, then re-enabled and restarted before another message can be transmitted.

### 4.3. Data Reception

The receiver must be enabled to be able to receive data over the cable interface. Generally speaking, the receiver must be disabled and re-enabled before receiving another message.

### 4.4. Loop Back Operation

For loop back operation, the requirements for successful data transfer depend on the I/O mode in use. For PIO and DMA modes, the transmitter need not be enabled or started before writing data to the Tx FIFO. For DMDMA however, the transmitter must be enabled and started, and the receiver enabled before data can be written to the Tx FIFO. For PIO and DMA, both the transmitter and the receiver must be enabled before the transmitter is started.

### 4.5. Data Transfer Options

#### 4.5.1. PIO

This mode uses repetitive register accesses in performing data transfers and is most applicable for low throughput requirements. For read requests the driver effectively performs repetitive register reads from the receive FIFO so long as the FIFO is not empty. For write requests the driver effectively performs repetitive register writes to the transmit FIFO so long as the FIFO is not full. The Almost Empty and Almost Full FIFO status levels are not used for PIO based data transfers.

**NOTE:** Applications can make read or write requests of any desired size irrespective of the size of the FIFOs on the HPDI32. For PIO transfers the driver breaks requests into smaller pieces according to the configured size of the transfer buffers and the amount of available data/space. I/O requests using PIO should never return a failure status.

#### 4.5.2. Standard DMA

This mode is intended for data transfers that do not exceed the size of the respective FIFO. In this mode, all data transfer between the PCI interface and the FIFOs is done in burst mode. The FIFO fill level status settings have no effect on data transfers.

**NOTE:** Applications are responsible for ensuring that write requests do not exceed available FIFO space and that read requests do not exceed available FIFO data. If a write request exceeds available FIFO space, then the excess data will be lost. The Tx FIFO Overrun will be recorded in the Board Status Register. If a read request exceeds available FIFO data, then the excess will be indeterminate. The Rx FIFO Under Run will be recorded in the Board Status Register.

**NOTE:** A DMA request that seeks to transfer PIO Threshold or fewer samples will be performed using PIO. This is done because it is more efficient to perform small data transfers using PIO than to use DMA.

### 4.5.3. Demand Mode DMA

This mode is intended for data transfers that exceed the size of the respective FIFO and uses the FIFO fill levels to control data bursting during data transfer. While the FIFOs can hold up to 128K data samples, depending on the FIFO size, Demand Mode DMA reads and writes may typically entail requests for millions of data values in a single call. For write operations, data transfer occurs in burst mode while the transmit FIFO is not Almost Full. While the FIFO is Almost Full data is transferred in non-burst mode. No data transfer occurs while the FIFO is Full. For read operations, data transfer occurs in burst mode while the receive FIFO is not Almost Empty. While the FIFO is Almost Empty data is transferred in non-burst mode. No data transfer occurs while the FIFO is Empty.

**NOTE:** A DMA request that seeks to transfer PIO Threshold or fewer samples will be performed using PIO. This is done because it is more efficient to perform small data transfers using PIO than to use DMA.

## 4.6. Interrupt Notification

Interrupt notification under Linux requires a number of Linux and HPDI32 specific steps. The example below illustrates the steps required.

### Example

```
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "HPDI32DocSrcLib.h"

static int _fd;

static void handle_sigio(int signo)
{
    hpdi32_interrupt_t data;
    int status;

    status = ioctl(_fd, HPDI32_IOCTL_INT_STATUS, &data);

    if (status)
    {
        /* The request failed. */
    }
    else if (data.status.gsc & HPDI32_INT_GSC_TX_FIFO_AE)
    {
        /* Perform actions specific to this interrupt. */
    }
    else if (data.status.gsc & HPDI32_INT_GSC_RX_FIFO_AF)
```

```

    {
        /* Perform actions specific to this interrupt. */
    }
}

int hpdi32_async_setup(int fd)
{
    hpdi32_interrupt_t  data;
    int                 flags;
    pid_t               pid;
    hpdi32_reg_t        reg;
    int                 status;

    _fd = fd;
    signal(SIGIO, handle_sigio);
    pid = getpid();
    fcntl(fd, F_SETOWN, pid);
    flags = fcntl(fd, F_GETFL);
    flags |= FASYNC;
    fcntl(fd, F_SETFL, flags);
    data.notify.gsc      = HPDI32_INT_GSC_TX_FIFO_AE
                          | HPDI32_INT_GSC_RX_FIFO_AF;
    data.notify.other    = 0;
    status               = ioctl(fd, HPDI32_IOCTL_INT_NOTIFY, &data);
    reg.value            = HPDI32_INT_GSC_TX_FIFO_AE
                          | HPDI32_INT_GSC_RX_FIFO_AF;
    reg.mask             = HPDI32_INT_GSC_TX_FIFO_AE
                          | HPDI32_INT_GSC_RX_FIFO_AF;

    if (status == 0)
    {
        reg.reg = HPDI32_GSC_IELR; /* Edge Triggered */
        status = ioctl(fd, HPDI32_IOCTL_REG_MOD, &reg);
    }

    if (status == 0)
    {
        reg.reg = HPDI32_GSC_IHLR; /* Rising Edge */
        ioctl(fd, HPDI32_IOCTL_REG_MOD, &reg);
    }

    if (status == 0)
    {
        reg.reg = HPDI32_GSC_ICR; /* Enable */
        ioctl(fd, HPDI32_IOCTL_REG_MOD, &reg);
    }

    if (status)
    {
        data.notify.gsc      = 0;
        data.notify.other    = 0;
        ioctl(fd, HPDI32_IOCTL_INT_NOTIFY, &data);
    }

    return(status);
}

```

## 4.7. Memory Mapped Resources

The `mmap()` and `munmap()` system calls are supported by this driver so that various driver resources can be accessed directly by an application. The resources that may be mapped are the GSC specific register block, the `write()` data transfer buffer, and the `read()` data transfer buffer. If the GSC registers are mapped, then an application can read and write the registers without the overhead of the `ioctl()` service. If the data transfer buffers are mapped, then the `read()` and `write()` services operate more efficiently. The driver specific parameters for the `mmap()` and `munmap()` system calls are retrieved using the `HPDI32_IOCTL_MMAP_INFO` IOCTL service. The `hpdi32_mmap_t.regs` structure contains the parameters required for accessing the GSC specific register block. The `hpdi32_mmap_t.read` structure contains the parameters required for accessing the read transfer buffer. The `hpdi32_mmap_t.write` structure contains the parameters required for accessing the read transfer buffer. Below are the general steps that should be followed when using the `mmap` feature.

**WARNING:** All GSC registers must be accessed as 32-bit values on 32-bit boundaries. The results are indeterminate if not accessed in this manner. The pointer returned by `mmap()` for the GSC register should be cast as a “`__u32*`” data type.

**WARNING:** All restrictions and precautions that apply to GSC registers when accessed via the `ioctl()` service also apply when being accessed directly.

**WARNING:** The interrupt registers (ICR, IELR and IHLR) should not be modified directly. They should only be modified via the `HPDI32_IOCTL_REG_MOD` IOCTL service.

**NOTE:** On some systems access of the GSC registers block via `mmap()` is unsupported. This occurs mostly with embedded motherboards because the BIOS tries to conserve resources when mapping PCI device memory and I/O regions into the processor’s address space. Use of the `mmap()` feature to access the GSC registers requires that the memory region be placed on a boundary the size of the processors physical memory page. In cases where this does not occur the driver will set the field `regs.size` to zero (0) to reflect that the registers are not `mmap()` accessible. Attempts to use `mmap()` when this does occur may return a NULL pointer.

1. Determine the desired I/O parameters along with the most efficient data transfer buffer sizes for the desired data transfer modes. Using the `HPDI32_IOCTL_IO_CONFIG` IOCTL service set the read and write sample values to `HPDI32_IO_SAMPLES_MIN`. This will increase the available kernel memory for the next request. Now issue the IOCTL service using the read and write sample values previously calculated.
2. Issue the `HPDI32_IOCTL_MMAP_INFO` IOCTL service to obtain the values required by the driver for the `mmap()` and `munmap()` system calls.
3. Issue the `mmap()` system call for each of the desired memory areas.
4. Perform operations as required by the application.
5. Issue a `munmap()` system call for each corresponding `mmap()` call made above.

## Document History

<b>Revision</b>	<b>Description</b>
July 30, 2007	Updated to driver version 1.19, release 0, library version 1.00.
September 29, 2006	Updated to driver version 1.18, release 0, library version 1.00.
August 23, 2006	Updated to driver version 1.17, release 0, library version 1.00.
May 30, 2006	Updated to driver version 1.16, release 0, library version 1.00.
April 24, 2006	Updated to driver version 1.15, release 1, library version 1.00.
April 18, 2006	Updated to driver version 1.15, release 0, library version 1.00.
February 22, 2006	Initial release of the DIPHASE2 library user manual.