

HPDI32

High Performance 32-bit Digital I/O

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

Linux Device Driver 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

E-mail: 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

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

1. Introduction.....	6
1.1. Purpose	6
1.2. Acronyms.....	6
1.3. Definitions	6
1.4. Software Overview	6
1.5. Hardware Overview	6
1.6. Reference Material.....	7
2. Installation.....	8
2.1. CPU and Kernel Support	8
2.2. The /proc File System	8
2.3. File List.....	9
2.4. Installation	9
2.5. Removal.....	9
2.6. Overall Make Script.....	10
2.7. The Driver.....	10
2.7.1. Build	10
2.7.2. Startup	10
2.7.3. Verification.....	12
2.7.4. Version	12
2.7.5. Shutdown.....	12
2.8. Document Source Code Examples.....	12
2.8.1. Build.....	13
2.8.2. Use.....	13
2.9. Sample Applications	13
2.9.1. hpdi32Test	13
2.9.2. irq	14
2.9.3. Transmit.....	16
3. Driver Interface.....	18
3.1. Macros	18
3.1.1. Interrupt Identification Bits: GSC Specific	18
3.1.2. Interrupt Identification Bits: Non-GSC Specific	19
3.1.3. IOCTL	19
3.1.4. I/O PIO Threshold Options.....	19
3.1.5. I/O Data Sizes.....	19
3.1.6. I/O Transfer Modes	20
3.1.7. I/O Transfer Buffer Size	20
3.1.8. I/O Timeouts.....	20
3.1.9. mmap() Register Decoding.....	21
3.1.10. Registers	21
3.2. Data Types	24
3.2.1. hpdi32_debug_data_t	25
3.2.2. hpdi32_driver_info_t.....	25

3.2.3. hpdi32_interrupt_t	25
3.2.4. hpdi32_io_config_t	26
3.2.5. hpdi32_mmap_mem_t	27
3.2.6. hpdi32_mmap_t	27
3.2.7. hpdi32_reg_t	28
3.3. Functions	28
3.3.1. close()	28
3.3.2. ioctl()	29
3.3.3. mmap()	30
3.3.4. munmap()	31
3.3.5. open()	32
3.3.6. read()	33
3.3.7. write()	34
3.4. IOCTL Services	35
3.4.1. HPDI32_IOCTL_DEBUG_DATA_GET	35
3.4.2. HPDI32_IOCTL_DRIVER_INFO_GET	36
3.4.3. HPDI32_IOCTL_INT_NOTIFY	37
3.4.4. HPDI32_IOCTL_INT_STATUS	38
3.4.5. HPDI32_IOCTL_IO_CONFIG	39
3.4.6. HPDI32_IOCTL_MMAP_INFO	40
3.4.7. HPDI32_IOCTL_NO_COMMAND	40
3.4.8. HPDI32_IOCTL_REG_MOD	41
3.4.9. HPDI32_IOCTL_REG_READ	42
3.4.10. HPDI32_IOCTL_REG_WRITE	43
4. Operation	44
4.1. Read and Write Operations	44
4.2. Data Transmission	44
4.3. Repetitious Data Sequence Transmission	45
4.4. Data Reception	45
4.5. Data Transfer Options	45
4.5.1. PIO	45
4.5.2. Standard DMA	45
4.5.3. Demand Mode DMA	47
4.6. Interrupt Notification	47
4.7. Memory Mapped Resources	49
Document History	51

1. Introduction

This user manual applies to driver version 1.19, release 0.

1.1. Purpose

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

1.2. Acronyms

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

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

1.3. Definitions

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

Term	Definition
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. With the driver, user applications are able to open and close a device and, while open, perform read, write and I/O control operations.

1.5. Hardware Overview

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

The HPDI32 offers a half-duplex external I/O interface. The board can either transmit or receive data, but it cannot do both simultaneously. In addition to the 32 synchronous data I/O lines, the external interface includes a set of configurable flow control signals. Some of these can also be configured as discrete I/O. The board accommodates a wide range of applications. This range extends from sending or receive relatively small blocks of data on demand, to sending or receiving large continuous streams of data for an extended period. Once a data link is established, the

data is transferred to/from host memory by simply writing to or reading from the onboard FIFOs. The board has an advanced PCI interface engine, which provides for increased data throughput via DMA.

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

1.6. Reference Material

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

- The applicable *HPDI32 User Manual* from General Standards Corporation.
- The *PCI9080 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc. *
- The *PCI9656 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc. *

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

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

2. Installation

2.1. CPU and Kernel Support

The driver is designed to operate with Linux kernel versions 2.6, 2.4 and 2.2 running on a PC system with one or more 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 driver has not been tested with a non-versioned kernel.

NOTE: The driver 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. 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: 64
models: *
```

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 “*” identifies a generic HPDI32. Other values identify other HPDI32 model variations. For example, “DIPHASE2” identified an HPDI32A-DIPHASE2 board.
--------	--

2.3. File List

This release consists of the below listed files. The archives are described in detail in following subsections.

File	Description
hpdi32.tar.gz	This archive contains the driver and all related sources.
hpdi32_linux_driver_user_manual.pdf	This is a PDF version of this user manual.

2.4. Installation

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

1. Change the current directory to `/usr/src/linux/drivers`. (The path name may vary among distributions and kernel versions.)
2. Copy the archive file `hpdi32.tar.gz` into the current directory.
3. Issue the following command to decompress and extract the files from the provided archive. This creates the directory `hpdi32` in the current directory, and then copies all of the archive’s files into this new directory.

```
tar -xzf hpdi32.tar.gz
```

2.5. Removal

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

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

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

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

```
rm -rf /dev/hpdi32*
```

5. If the automated startup procedure was adopted (described in following paragraphs), then edit the system startup script `rc.local` and remove the line that invokes the `hpdi32_start` script. The file `rc.local` should be located in the `/etc/rc.d` directory.

2.6. Overall Make Script

A make script is included in the root installation directory. Executing this script will perform a make for all build targets included in the release. The script is named `hpdi32_make`.

2.7. The Driver

This driver and its related files are contained in the archive file `hpdi32.tar.gz`. The archive's device driver files are listed below. The paragraphs that follow give installation, build and startup instructions.

File	Description
<code>*.c</code>	The driver source files.
<code>*.h</code>	The driver header files.
<code>hpdi32.h</code>	A driver header file. This header should be included by HPDI32 applications.
<code>hpdi32.ko*</code>	The driver executable.
<code>hpdi32_start</code>	Shell script to install the driver executable and device nodes.
<code>makefile</code>	The driver make file.
<code>makefile.dep</code>	An automatically generated make dependency file.

* The file name extension for the pre-built driver executable is `.ko`, which is the convention for the 2.6 kernel. The 2.6 build of the driver executable is included in the release as the final release is built using the 2.6 kernel. The convention for the 2.2 and 2.4 kernels is an extension of `.o`. The driver build procedure produces the appropriately named file.

2.7.1. Build

NOTE: Building the driver requires installation of the kernel sources.

Follow the below steps to build the driver.

1. Change to the directory where the driver and its sources were installed. This should be `/usr/src/linux/drivers/hpdi32/driver`.
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 all
```

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

2.7.2. Startup

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

NOTE: If using the most recent drivers, there are no longer any conflicts between the use of HPDI32 and DIO24 boards and their respective drivers.

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

2.7.2.1. Manual Driver Startup Procedures

Start the driver manually by following the below listed steps.

1. Login as root user, as some of the steps require root privileges.
2. Change to the directory where the driver was installed. This should be `/usr/src/linux/drivers/hpdi32/driver`.
3. Install the driver module and create the device nodes by executing the below command. If any errors are encountered then an appropriate error message will be displayed.

```
./hpdi32_start
```

NOTE: The script's default specifies that the driver be installed in the same directory as the script. The script will fail if this is not so.

NOTE: The above step must be repeated each time the host is rebooted.

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

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

```
lsmod
```

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

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

2.7.2.2. Automatic Driver Startup Procedures

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

1. Locate and edit the system startup script `rc.local`, which should be in the `/etc/rc.d` directory. Modify the file by adding the below line so that it is executed with every reboot.

```
/usr/src/linux/drivers/hpdi32/driver/hpdi32_start
```

NOTE: The script's default specifies that the driver be installed in the same directory as the script. The startup script will fail if this is not so.

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

2.7.3. Verification

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 being exercised in its default configuration. Damage may result because the sample application uses the board's default configuration, which drives various external output signals. No damage will result if no cable at all is attached.

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

1. Install the sample application as described in subsequent paragraphs.
2. Change to the directory where the sample application `hpdi32Test.o` was installed.
3. Start the sample application by issuing the below command. The argument identifies which board to access and is required only if multiple boards are installed. The argument is the zero based index of the board to access.

```
./hpdi32Test.o <board>
```

2.7.4. Version

The driver version number can be obtained in a variety of ways. It is reported by the driver both when the driver is loaded and when it is unloaded (depending on kernel configuration options, this may be visible only in `/var/log/messages`). It is recorded in the text file `/proc/hpdi32`. It can also be read by an application via the `HPDI32_IOCTL_DRIVER_INFO_GET` IOCTL service.

2.7.5. Shutdown

Shutdown the driver following the below listed steps.

1. Login as root user, as some of the steps require root privileges.
2. If the driver is currently loaded then issue the below command to unload the driver.

```
rmmmod hpdi32
```

3. Verify that the driver module has been unloaded by issuing the below command. The module name `hpdi32` should not be in the list.

```
lsmod
```

2.8. Document Source Code Examples

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

File	Description
<code>*.c</code>	These are the C source files.
<code>HPDI32DocSrcLib.a</code>	This is a pre-built, linkable version of the library.

HPDI32DocSrcLib.h	This is the library header file.
makefile	This is the library make file.
makeile.dep	This is an automatically generated make dependency file.

2.8.1. Build

Follow the below steps to compile the example files.

1. Change to the directory where the source code example files were installed.
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 all
```

NOTE: The build procedure will fail if the driver sources are not installed in the directory documented in the driver installation procedures.

2.8.2. 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 HPDI32DocSrcLib.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/docsrc. Link time use also has two requires. First, include the static library HPDI32DocSrcLib.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/docsrc.

2.9. Sample Applications

2.9.1. hpdi32Test

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 that tests the functionality of the driver and a user specified HPDI32 board. It can be used as the starting point for application development on top of the HPDI32 Linux device driver. The application performs an automated test of the driver features. The application includes the below listed files.

File	Description
*.c	These are the application's source files.
hpdi32Test	This is the pre-built sample application.
main.h	This is the application's header file.
makeapp.sh	This is the script that initiates a build of the sample application.
makefile	This is a make file.
makefiel.dep	This is a make dependency file.
../utils/*.c	These are utility sources used by the application.
../utils/*.h	These are the headers for the utility sources used by the application.

2.9.1.1. 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/test`.

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 all
```

2.9.1.2. 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. Once started the application will automatically execute a series of tests to verify the operation of the driver and the board. The application will repeat the test cycle as called for by the arguments and the accumulated test results. A single test cycle should take less than one minute to complete. The command line arguments are described in the table below.

```
./hpdi32Test <-c> <-C> <-m#> <-n#> <board>
```

Argument	Description
-c	This will cause the operation to repeat until an error is encountered.
-C	This will cause the operation to repeat even after an error is encountered.
-m#	This will cause the operation to repeat for at most “#” minutes, where “#” is a decimal number.
-n#	This will cause the operation to repeat at most “#” times, where “#” is a decimal number.
board	This is the index of the board to access and is required only if multiple boards are installed.

2.9.2. irq

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 automated interrupt testing of a user specified HPDI32 board. It can be used as the starting point for application development on top of the HPDI32 Linux device driver. The application includes the below listed files.

File	Description
main.c	This is the main source file.
main.h	This is the application's header file.
makefile	This is a make file.
makefiel.dep	This is a make dependency file.
irq	This is the pre-built sample application.
irq*.c	These are the interrupt based source files.
rx.c	This is an additional source file.
../utils/*.c	These are utility sources used by the application.
../utils/*.h	These are the headers for the utility sources used by the application.

2.9.2.1. 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/irq`
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 all
```

2.9.2.2. 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. A single iteration should take just a few seconds. The command line arguments are described in the table below.

```
./irq <-c> <-C> <-fc#> <-gpio#> <-irq#> <-m#> <-n#> <board>
```

Argument	Description
-c	This will cause the operation to repeat until an error is encountered.
-C	This will cause the operation to repeat even after an error is encountered.
-fc#	This specifies a hexadecimal mask of the Flow Control interrupts to test. The default is to test all Flow Control interrupts.
-gpio#	This specifies a hexadecimal mask of the GPIO interrupts to test. The default is to test all GPIO interrupts.
-irq#	This specifies a hexadecimal mask of the interrupts to test. The default is to test all interrupts.
-m#	This will cause the operation to repeat for at most “#” minutes, where “#” is a decimal number.

-n#	This will cause the operation to repeat at most “#” times, where “#” is a decimal number.
board	This is the index of the board to access and is required only if multiple boards are installed.

2.9.3. Transmit

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 data transmission test of a user specified HPDI32 board. It can be used as the starting point for application development on top of the HPDI32 Linux device driver. The application performs data transmission per the parameters specified by macros in `_config_board()` routine of the source `tx.c`. The application includes the below listed files.

File	Description
<code>main.c</code>	This is the main source file.
<code>main.h</code>	This is the application’s header file.
<code>makefile</code>	This is a make file.
<code>makefiel.dep</code>	This is a make dependency file.
<code>tx</code>	This is the pre-built sample application.
<code>tx.c</code>	This is the source for the data transmission portion of the application.
<code>../utils/*.c</code>	These are utility sources used by the application.
<code>../utils/*.h</code>	These are the headers for the utility sources used by the application.

2.9.3.1. 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/tx`.
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 all
```

2.9.3.2. 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 configure and board and transmit data as coded in `tx.c`. The application will repeat the transmission as called for by the arguments and the accumulated test results. A single iteration should take about four seconds with the default configuration settings. The command line arguments are described in the table below.

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

Argument	Description
-c	This will cause the operation to repeat until an error is encountered.
-C	This will cause the operation to repeat even after an error is encountered.
-m#	This will cause the operation to repeat for at most “#” minutes, where “#” is a decimal number.
-n#	This will cause the operation to repeat at most “#” times, where “#” is a decimal number.
board	This is the index of the board to access and is required only if multiple boards are installed.

3. Driver Interface

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

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

3.1. Macros

The driver interface includes the following macros, which are defined in `hpdi32.h`. The header 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_FRAME_VALID_E</code>	This refers to the end of the Frame Valid assertion pulse.
<code>HPDI32_INT_GSC_FRAME_VALID_S</code>	This refers to the start of the Frame Valid assertion pulse.
<code>HPDI32_INT_GSC_LV_GPIO0</code>	This refers to the Line Valid/GPIO0 signal.
<code>HPDI32_INT_GSC_RE_GPIO5</code>	This refers to the Receive Enable/GPIO5 signal.
<code>HPDI32_INT_GSC_RR_GPIO2</code>	This refers to the Receive Ready/GPIO2 signal.
<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_SV_GPIO1</code>	This refers to the Status Valid/GPIO1 signal.
<code>HPDI32_INT_GSC_TE_GPIO4</code>	This refers to the Transmit Enable/GPIO4 signal.
<code>HPDI32_INT_GSC_TR_GPIO3</code>	This refers to the Transmit Ready/GPIO3 signal.
<code>HPDI32_INT_GSC_TX_FIFO_AE</code>	This refers to the Tx FIFO Almost Empty status.
<code>HPDI32_INT_GSC_TX_FIFO_AF</code>	This refers to the Tx FIFO Almost Full status.
<code>HPDI32_INT_GSC_TX_FIFO_EMPTY</code>	This refers to the Tx FIFO Empty status.
<code>HPDI32_INT_GSC_TX_FIFO_FULL</code>	This refers to the Tx FIFO Full status.

This set of utility macros defines masks associated with groups of interrupt sources.

Macros	Description
<code>HPDI32_INT_GSC_RX_FIFO_MASK</code>	This applies to the set of Rx FIFO interrupts.
<code>HPDI32_INT_GSC_TX_FIFO_MASK</code>	This applies to the set of Tx FIFO interrupts.

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: Reads and writes using the DMDMA option must be performed with the Transmit Enable and Receive Enable bits set in the Board Control Register, respectively.

NOTE: The driver will automatically set the DMA Channel 0 DMDMA Direction bit in the Board Control Register as needed. An ongoing DMDMA may be abruptly halted if this bit is altered by an application prior to completion of the data transfer.

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 Linux device drivers. 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
HPDI32_IO_TIMEOUT_DEFAULT	This is the default timeout assigned each time the device is opened.
HPDI32_IO_TIMEOUT_MAX	This specifies the maximum timeout period.
HPDI32_IO_TIMEOUT_MIN	This specifies the minimum timeout period.
HPDI32_IO_TIMEOUT_NO_CHANGE	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 HPDI32 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 *HPDI32 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_FSR</code>	Feature Set Register (FSR)
<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_RFSR</code>	Rx FIFO Size Register (RFSR)
<code>HPDI32_GSC_RFWR</code>	Rx FIFO Words Register (RFWR)
<code>HPDI32_GSC_RLCR</code>	Rx Line Counter Register (RLCR)
<code>HPDI32_GSC_RSCR</code>	Rx Status Counter Register (RSCR)
<code>HPDI32_GSC_TAR</code>	Tx Almost Register (TAR)
<code>HPDI32_GSC_TCDR</code>	Tx Clock Divider Register (TCDR)
<code>HPDI32_GSC_TFSR</code>	Tx FIFO Size Register (TFSR)
<code>HPDI32_GSC_TFWR</code>	Tx FIFO Words Register (TFWR)
<code>HPDI32_GSC_TLILCR</code>	Tx Line Invalid Length Count Register (TLILCR)
<code>HPDI32_GSC_TLVLCR</code>	Tx Line Valid Length Count Register (TLVLCR)
<code>HPDI32_GSC_TSVLCR</code>	Tx Status Valid Length Count Register (TSVLCR)

3.1.10.2. PCI Configuration Registers

The following table gives the set of PCI configuration registers available on both 32-bit and 64-bit boards. For detailed definitions of these registers refer to the *PCI9080 Data Book* if you have a 32-bit board or to the *PCI9656 Data Book* if you have a 64-bit board.

Macros	Description
<code>HPDI32_PCI_BAR0</code>	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 PCIIDR value of 0x965610B5 identifies the PCI interface chip as a PLX PCI9656, which is a 64-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 and a value of 0x2705 identifies 64-bit versions of the HPDI32. Refer to the GSC Firmware Revision Register for additional identification information.

The following table gives the set of additional PCI configuration registers available on 64-bit boards. For detailed definitions of these registers refer to the *PCI9656 Data Book*.

Macros	Description
HPDI32_PCI_CAP_PTR	New Capability Pointer Register (CAP_PTR)
HPDI32_PCI_HS_CNTL	Hot Swap Control Registers (HS_CNTL)
HPDI32_PCI_HS_CSR	Hot Swap Control/Status Register (HS_CSR)
HPDI32_PCI_HS_NEXT	Hot Swap Next Capability Pointer Register (HS_NEXT)
HPDI32_PCI_PMC	Power Management Capabilities Register (PMC)
HPDI32_PCI_PMCAPID	Power Management Capability ID Register (PMCAPID)
HPDI32_PCI_PMCSR	Power Management Control/Status Register (PMCSR)
HPDI32_PCI_PMCSR_BSE	PMCSR Bridge Support Expansions Register (PMCSR_BSE)
HPDI32_PCI_PMDATA	Power Management Data Register (PMDATA)
HPDI32_PCI_PMNEXT	Power Management Next Capability Pointer Register (PMNEXT)
HPDI32_PCI_VPD_NEXT	PCI Vital Product Data Next Capability Pointer Register (VVPD_NEXT)
HPDI32_PCI_VPDAD	PCI Vital Product Data Address Register (VVPDAD)
HPDI32_PCI_VPDATA	PCI VPD Data Register (VVPDATA)
HPDI32_PCI_VPDCNTL	PCI Vital Product Data Control Register (VVPDCNTL)

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* if you have a 32-bit board. Because the PLX PCI9656 is register compatible with the PCI9080, the below tables apply to both PCI interface chips. Refer to the *PCI9656 Data Book* if you have a 64-bit board.

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 (IFTPR)
HPDI32_PLX_IPHPR	Inbound Post Head Pointer Register (IPHPR)
HPDI32_PLX_IPTPR	Inbound Post Tail Pointer Register (IPTPR)
HPDI32_PLX_IQP	Inbound Queue Port Register (IQP)
HPDI32_PLX_MQCR	Messaging Queue Configuration Register (MQCR)
HPDI32_PLX_OFHPR	Outbound Free Head Pointer Register (OFHPR)
HPDI32_PLX_OFTPR	Outbound Free Tail Pointer Register (OFTP)
HPDI32_PLX_OPHPR	Outbound Post Head Pointer Register (OPHPR)
HPDI32_PLX_OPLFIM	Outbound Post List FIFO Interrupt Mask Register (OPLFIM)
HPDI32_PLX_OPLFIS	Outbound Post List FIFO Interrupt Status Register (OPLFIS)
HPDI32_PLX_OPTPR	Outbound Post Tail Pointer Register (OPTPR)
HPDI32_PLX_OQP	Outbound Queue Port Register (OQP)
HPDI32_PLX_QBAR	Queue Base Address Register (QBAR)
HPDI32_PLX_QSR	Queue Status/Control Register (QSR)

3.1.10.4. PLX PCI9656 Feature Set Registers

The following table gives the set of additional PLX PCI9656 feature set registers. For the complete set of PCI9656 feature set registers, combine the registers given here with those given above for the PCI9080. For detailed definitions of all PCI9656 registers refer to the *PCI9656 Data Book*.

Local Configuration Registers

Macros	Description
HPDI32_PLX_ABTADR	PCI Abort Address Register (PABTADR)
HPDI32_PLX_ARB	PCI Arbiter Control Register (PCIARB)
HPDI32_PLX_DMDAC	Direct Master PCI Dual Address Cycle Upper Address Register (DMDAC)
HPDI32_PLX_LMISC1	Local Miscellaneous Control 1 Register (LMISC1)
HPDI32_PLX_LMISC2	Local Miscellaneous Control 2 Register (LMISC2)
HPDI32_PLX_PROT_AREA	Serial EEPROM Write-Protected Address Boundary Register (PROT_AREA)

DMA Registers

Macros	Description
HPDI32_PLX_DMADAC0	DMA Channel 0 PCI Dual Address Cycle Upper Address Register (DMADAC0)
HPDI32_PLX_DMADAC1	DMA Channel 1 PCI Dual Address Cycle Upper Address Register (DMADAC1)

3.2. Data Types

This driver interface includes the following data types, which are defined in `hpdi32.h`.

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 mmap parameters required for accessing the GSC specific registers.
regs.offset	This is the value to pass as the mmap() offset parameter.
regs.size	This is the value to pass as the mmap() and munmap() length parameters.
read	This structure defines the mmap parameters required for accessing driver's read transfer buffer.
read.offset	This is the value to pass as the mmap() offset parameter.
read.size	This is the value to pass as the mmap() and munmap() length parameters.
write	This structure defines the mmap parameters required for accessing driver's write transfer buffer.
write.offset	This is the value to pass as the mmap() offset parameter.
write.size	This is the value to pass as the mmap() and munmap() length 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 value field that are to be applied during the read-modify-write IOCTL service. If a bit is set in the mask, then the corresponding value bit is applied to the register. If a mask bit is not set then the corresponding register bit is left unchanged.

3.3. 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 <code>request</code> does not call for any additional arguments, then any additional arguments provided are ignored. The HPDI32 IOCTL services use at most one argument.

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

Example

```

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

#include "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.
<code>prot</code>	This is the desired access and should equal "PROT_READ PROT_WRITE".
<code>flags</code>	This should be the value "MAP_SHARED".
<code>fd</code>	This is the file descriptor returned by the <code>open()</code> function.
<code>offset</code>	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 <code>O_RDWR</code> .

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 mmap() feature, then this must be NULL.
count	This is the desired number of bytes to read. This must be a multiple of four (4). If the buf 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 errno.
0 to count	The operation succeeded. For blocking I/O a return value less than count indicates that the request timed out. For non-blocking I/O a return value less than count 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 count indicates that the request timed out. For non-blocking I/O a return value less than count 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. 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.

3.4.1. 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

ioctl() Argument	Description
request	HPDI32_IOCTL_DEBUG_DATA_GET
arg	hpdi32_debug_data_t*

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);
}

```

3.4.2. HPDI32_IOCTL_DRIVER_INFO_GET

This service retrieves information about the driver itself.

Usage

ioctl() Argument	Description
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);
}
```

3.4.3. 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 HPDI32 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

ioctl() 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);
}
```

3.4.4. 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

ioctl() 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);
}

```

3.4.5. 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. If the driver's memory allocation request fails, then the service request also fails and the previous settings remain in 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

<code>ioctl()</code> Argument	Description
<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);
}

```

3.4.6. 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

ioctl() Argument	Description
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);
}

```

3.4.7. 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

ioctl() Argument	Description
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);
}
```

3.4.8. 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

ioctl() Argument	Description
request	HPDI32_IOCTL_REG_MOD
arg	hpdi32_reg_t*

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);
}

```

3.4.9. 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
<code>request</code>	<code>HPDI32_IOCTL_REG_READ</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_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);
}

```

3.4.10. 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

ioctl() Argument	Description
request	HPDI32_IOCTL_REG_WRITE
arg	hpdi32_reg_t*

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. This is in no way intended to be a comprehensive guide on using the HPDI32. 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. Depending on the features supported by the HPDI32 firmware, the driver may rely upon each board's Almost Empty and Almost Full settings to determine the amount of data that can be written to or read from the device. If these values are not accurate then data may unknowingly be lost during write operations and indeterminate data may unknowingly be obtained during read operations.

WARNING: Failure to reset the FIFOs after programming their Almost Empty and Almost Full values may result in data loss during write operations and indeterminate data being obtained during read operations. This would occur due to the driver's use of the yet to be programmed values while trying to determine the amount of available data/space during an I/O request.

NOTE: Failure to program the Tx and Rx FIFO Almost Empty and Almost Full settings with appropriate values may result in reduced throughput during read and write operations.

4.2. Data Transmission

Data transmission is essentially a three step process; configure the HPDI32, write data to the transmit FIFO and initiate a transfer. A simplified version of this process is illustrated in the steps outlined below.

NOTE: These steps are guidelines only. The actual steps needed may vary significantly from one application to another.

1. Perform a board reset to put the HPDI32 in a known state.
2. Perform the steps required for any desired interrupt notification, including configuration of the desired interrupts. Respond to notifications according to application needs.
3. Set the write operation I/O parameters.
4. Set the transmit FIFO's Almost Empty and Almost Full values. Don't forget to reset the FIFO.
5. Set the Tx Clock Divider.
6. Set the duration of the Tx Status Valid, Line Valid and Line Invalid signal states. If the Line Valid signal is to mirror the Status Valid assertion state, then set the BCR Line Valid High on Status Valid High bit. Otherwise clear the bit.
7. If the remote device is to control data transfer then set the BCR Cable Throttle Enable bit. If the local host is to control data transfer then clear this bit.
8. Enable data transmission by setting the BCR Transmit Enable bit.
9. Write the desired data to the device.
10. If the local host is to control data transmission then set the BCR Start Transmit bit to initiate data transfer.

4.3. Repetitious Data Sequence Transmission

The driver supports an efficient I/O write operation for circumstances where the data to be transmitted consists of a fixed data sequence that is repeated over and over. This is applicable only in cases where the data for the sequence fits entirely within the Tx FIFO minus the Almost Full value. The operation is initialized by first putting the sequence into the drivers write buffer. This can be done through normal `write()` calls or manually by direct access to the write transfer buffer via the `mmap()` feature. The operation is then carried out by making repetitive `write()` calls with a NULL data pointer and a count argument indicating the size of the sequence in bytes.

4.4. Data Reception

Data reception is essentially a three-step process; configure the HPDI32, initiate data reception and read the received data. A simplified version of this process is illustrated in the steps outlined below.

NOTE: These steps are guidelines only. The actual steps needed may vary significantly from one application to another.

1. Perform a board reset to put the HPDI32 in a known state.
2. Perform the steps required for any desired interrupt notification, including configuration of the desired interrupts. Respond to notifications according to application needs.
3. Set the read operation I/O parameters.
4. Set the receive FIFO's Almost Empty and Almost Full values. Don't forget to reset the FIFO.
5. Enable data reception by setting the BCR Receive Enable bit.
6. Read the data as it is received according to the application's needs.

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 the Tx Overrun feature is supported in firmware. If a read request exceeds available FIFO data, then the excess will contain indeterminate data. The Rx FIFO Under Run will be recorded in the Board Status Register, if the Rx Under Run feature is supported in firmware.

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.

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

1. Use the transmit FIFO's Almost Full status as a stimulus to queue additional data for subsequent write operations. The amount of data that needs to be queued is generally a function of the data transfer rate, the period of time over which the rate is to be maintained, the amount of data to be transmitted in the allotted period, the amount of time needed to make the data available for queuing, and application, driver and system overhead. Since the Almost Full status doesn't affect data transfer into the Tx FIFO, the fill level can be set strictly according to application needs.
2. Use the transmit FIFO's Almost Empty status as a stimulus to perform a write operation. The amount of data submitted in each request should be the size of the transmit buffer, which is described below. Since the Almost Empty status doesn't affect Tx FIFO data transfer, the fill level can be set strictly according to application needs. It is desirable though to set the Almost Empty status level as low as possible, as it affects the optimal size of the transmit buffer. In contrast however, the status level should be set high enough to prevent the FIFO from becoming empty before the write operation, thus preventing a lapse in data transmission due to an empty FIFO.
3. The optimal size of the transmit buffer is the size of the transmit FIFO, minus the Almost Empty value.
4. Write requests that exceed the size of the transmit buffer will result in reduced efficiency. Smaller requests result in more requests being made, but they do not reduce the efficiency of individual write requests. Individual write requests may operate with reduced efficiency if they are not timed with the FIFO's Almost Empty status.
5. Access the driver's write transfer buffer using the `mmap()` feature implemented in the driver.

NOTE: This methodology is most applicable to HPDI32s that support the FIFO size registers. When these registers are unsupported, optimal performance is gained by considerably different means.

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

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

3. Read requests that exceed the size of the receive buffer will result in reduced efficiency. Smaller requests result in more requests being made, but they do not reduce the efficiency of individual read requests. Individual read requests might operate with reduced efficiency if they are not timed with the FIFO's Almost Full status.
4. Access the driver's read transfer buffer using the `mmap()` feature implemented in the driver.

NOTE: This methodology is most applicable to HPDI32s that support the FIFO size registers. When these registers are unsupported, optimal performance is gained by considerably different means.

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 32K data values, 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.

Using Demand Mode DMA for a large transmit operation (larger than the FIFO) requires that the HPDI32 be primed before initiating the write request. Priming the board simply involves putting some data in the FIFO (do not try to exceed the FIFO's size), setting the BCR Start Transmit bit, then initiating the large Demand Mode DMA transfer before the FIFO runs empty. Once started, large transfers can be maintained so long as the transmit FIFO doesn't run empty. If the FIFO does run empty, then the board must be re-primed. This priming action is required because data transmission can only be started while data is in the FIFO and because the BCR's Transmit Start bit automatically clears itself when the FIFO is empty. If this procedure is not followed, then large write requests will timeout with the transmit FIFO being full and no data will have been transferred.

NOTE: A Demand Mode 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 Demand Mode 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

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