# HPDI32

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

# All Form Factors
# …-HPDI32/A/AL/ALT/B

# API Library
# Reference Manual

**Manual Revision: July 12, 2023**
**Driver Release Version 3.12.104.x.x**

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

# Preface



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

# Table of Contents

# Table of Figures

# 1. Introduction

## 1.1. Purpose

The purpose of this document is to describe the interface to the HPDI32 API Library and, to a lesser extent, the underlying device driver. The API Library software provides the interface between "Application Software" and the device driver. The driver software provides the interface between the API Library and the actual HPDI32 hardware. The API Library and device driver interfaces are primarily IOCTL based.

## 1.2. Acronyms

The following is a list of commonly occurring acronyms which may appear throughout this document.

| Acronyms | Description |
|---|---|
| API | Application Programming Interface |
| BMDMA | Block Mode DMA |
| DIL | Driver Interface Library |
| DLL | Dynamic Link Library |
| DMA | Direct Memory Access |
| DMDMA | Demand Mode DMA |
| GSC | General Standards Corporation |
| PCI | Peripheral Component Interconnect |
| PCI64 | A PCI form factor device with a 64-bit, 66MHz PCI bus. |
| PCIe | PCI Express |
| PIO | Programmed I/O |
| PMC | PCI Mezzanine Card |
| PMC64 | A PMC form factor device with a 64-bit, 66MHz PCI bus. |

## 1.3. Definitions

The following is a list of commonly occurring terms which may appear throughout this document.

| Term | Definition |
|---|---|
| … | This is a shortcut representation of the HPDI32 installation directory or any of its subdirectories. |
| API Library | This is a library that provides application-level access to HPDI32 hardware. |
| Application | This is a user mode process, which runs in user space with user mode privileges. |
| Driver | This is the HPDI32 device driver, which runs in kernel space with kernel mode privileges. |
| HPDI32 | This is used as a general reference to any device supported by this driver. |
| Library | This is usually a general reference to the API Library. |
| Linux | This refers to the Linux operating system. Refer to the *HPDI32 Linux Driver User Manual*. |
| Windows | This refers to the Windows operating system. Refer to the *HPDI32 Windows Driver User Manual*. |

## 1.4. Software Overview

### 1.4.1. Basic Software Architecture

This section describes the general architecture for the basic components that comprise HPDI32 applications. The overall architecture is illustrated in Figure 1 below.

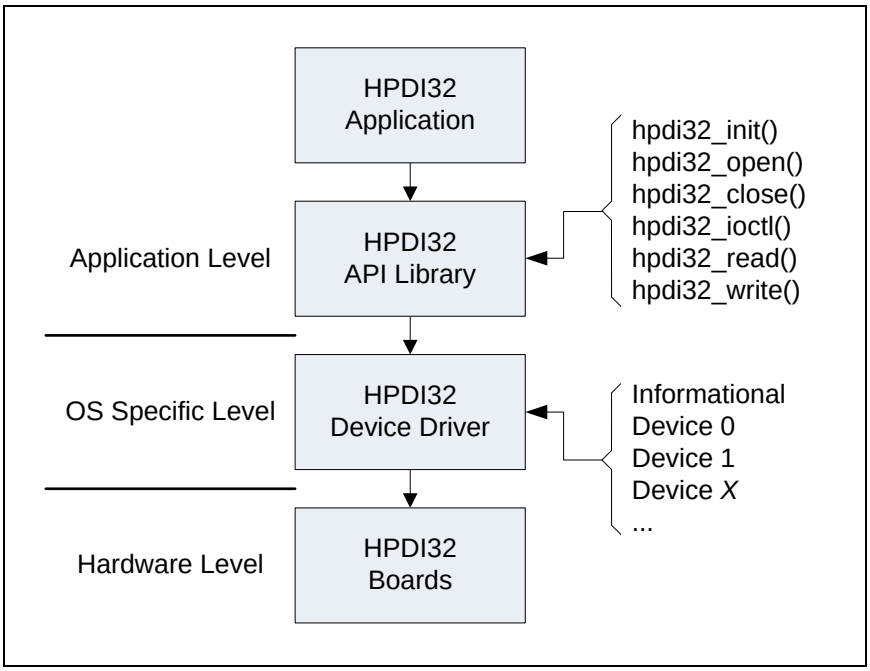


**Figure 1** Basic architectural representation.

### 1.4.2. API Library

The primary means of accessing HPDI32 boards is via the HPDI32 API Library. This library forms a layer between the application and the driver. Additional information is given in section 4 (page 14). With the library, applications are able to open and close a device and, while open, perform I/O control and read and write operations.

### 1.4.3. Device Driver

The device driver is the host software that provides a means of communicating directly with HPDI32 hardware. Depending on the OS, the driver may be a user space application, a kernel mode process, or something in between. The software interface to the device driver is analogous to that of the API Library.

## 1.5. Hardware Overview

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

The HPDI32 offers a half-duplex external I/O interface. The board can either transmit or receive data, but it cannot do both simultaneously. In addition to the 32 synchronous data, I/O lines, the external interface includes a set of configurable flow control signals. Some of these can also be configured as discrete I/O. The board accommodates a wide range of applications. This extends from transferring small blocks of data on demand, to transferring 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/PMC interface can be used interchangeably in 64-bit PCI /PMC slots. However, the performance improvements associated with the 64-bit PCI interface can be achieved only when a 64-bit device is used in a 64-bit slot.

## 1.6. Reference Material

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

- The applicable *HPDI32 Device Driver User Manual* for your operating system from General Standards Corporation.
- 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

## 1.7. Licensing

For licensing information please refer to the text file `LICENSE.txt` in the root installation directory.

# 2. Installation

For additional information on driver installation refer to this same section number in the OS specific HPDI32 driver user manual.

## 2.1. Host and Environment Support

For information on host and environment support refer to this same section number in the OS specific HPDI32 driver user manual.

## 2.2. Driver and Device Information

Each driver implements an OS specific means of obtaining generic, high-level information about the driver and the installed devices. The information is given in textual format. Each line of text begins with an entry name, which is followed immediately by a colon, a space character, and an entry value. Below is an example of what is provided, followed by descriptions of each entry. This information is accessed by passing a device index value of `-1` to the API open service (section 4.6.4, page 18).

```
version: 3.12.104.47
32-bit support: yes
boards: 1
models: HPDI32
ids: 0x3
```

| Entry | Description |
|---|---|
| `version` | This gives the driver version number in the form `x.x.x.x`. |
| `32-bit support` | This reports the driver's support for 32-bit applications. This will be either "`yes`" or "`no`" for 64-bit driver builds and "`yes (native)`" for 32-bit builds. |
| `boards` | This identifies the total number of boards the driver detected. |
| `models` | This gives a comma separated list of the basic model number for each board the driver detected. The model numbers are listed in the same order that the boards are accessed via the API Library's open function. |
| `ids` | This is a list identifying the values read from each board's user jumpers. The id numbers are listed in the same order that the boards are accessed via the API Library's open function. |

The API's source for the text provided is as follows.

| OS | Source |
|---|---|
| Linux | The file "`/proc/hpdi32`". |
| Windows | The Driver Interface Library DLL. |

## 2.3. File List

For the list of primary files included with each release refer to this same section number in the OS specific HPDI32 driver user manual.

## 2.4. Directory Structure

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

**NOTE:** Additional or alternate directories may be installed, depending on the OS. For additional information refer to this same section number in the OS specific HPDI32 driver user manual.

| Directory | Description |
|---|---|
| `hpdi32/` | This is the driver root directory. It contains the documentation, the Overall Make Script (section 2.7, page 12) and the below listed subdirectories. |
| `…/api/` | This directory contains the API Library source files (section 4, page 14). |
| `…/docsrc/` | This directory contains the source files for the code samples given in this document (section 6, page 49). |
| `…/driver/` | This directory contains the driver and any related files (section 5, page 48). |
| `…/include/` | This directory contains the header files for the various libraries. |
| `…/lib/` | This directory contains all of the libraries built from the installed sources. |
| `…/samples/` | This directory contains the sample application subdirectories and all of their corresponding source files (section 9, page 67). |
| `…/utils/` | This directory contains the source files for the utility libraries used by the sample applications (section 7, page 50). |

## 2.5. Installation

For installation instructions refer to this same section number in the OS specific HPDI32 driver user manual.

## 2.6. Removal

For removal instructions refer to this same section number in the OS specific HPDI32 driver user manual.

## 2.7. Overall Make Script

Each HPDI32 installation includes an OS specific means of building all of the build targets included in the installation. For additional information refer to this same section number in the OS specific HPDI32 driver user manual.

## 2.8. Environment Variables

For environment variable information refer to this same section number in the OS specific HPDI32 driver user manual.

# 3. Main Interface Files

This section gives general information on the suggested device interface files to use when developing HPDI32 based applications.

## 3.1. Main Header File

Throughout the remainder of this document references are made to various header files included as part of the HPDI32 driver installation. For ease of use it is suggested that applications include only the single header file shown below rather than individually including those headers identified separately later in this document. Including this header file pulls in all other pertinent HPDI32 specific header files. Therefore, sources may include only this one HPDI32 header and make files may reference only this one HPDI32 include directory.

| Description | File | Location | OS |
|---|---|---|---|
| Header File | `hpdi32_main.h` | `…/include/` | Linux |
| | | `…\include\` | Windows |

## 3.2. Main Library File

Throughout the remainder of this document references are made to various statically linkable libraries included as part of the HPDI32 driver installation. For ease of use it is suggested that applications link only the single library file shown below rather than individually linking those libraries identified separately later in this document. Linking this library file pulls in all other pertinent HPDI32 specific static libraries. Therefore, make files may reference only this one HPDI32 static library and only this one HPDI32 library directory.

| Description | File | Location | OS |
|---|---|---|---|
| Library File | `hpdi32_main.a`<br>`hpdi32_multi.a` | `…/lib/` | Linux |
| | `hpdi32_main.lib`<br>`hpdi32_multi.lib` | `…\lib\…` | Windows |

> **NOTE**: For applications using the HPDI32 and no other GSC devices, link the `hpdi32_main.xxx` library. For applications using multiple GSC device types, link the `xxxx_main.xxx` library for one of the devices and the `xxxx_multi.xxx` library for the others. Linking multiple `xxxx_main.xxx` libraries may likely produce link errors due to duplicate symbols being defined. While it may make little or no difference, it is recommended that one choose the `xxxx_main.xxx` library from the driver with the largest number in positions three (x.x.X.x.x) and/or four (x.x.x.X.x) in the driver release version number.
>
> **NOTE**: The HPDI32 API Library is implemented as a shared or dynamically loaded library and is thus not linked with the HPDI32 Main Library. The API Library must be linked with applications by an OS specific means. For additional information refer to this same section number in the OS specific HPDI32 driver user manual.

### 3.2.1. Build

For information on building the Main Library refer to this same section number in the OS specific HPDI32 driver user manual.

### 3.2.2. Additional Libraries

For information on any additional required libraries refer to this same section number in the OS specific HPDI32 driver user manual.

# 4. API Library

The HPDI32 API Library is the software interface between user applications and the HPDI32 device driver. The interface is accessed by including the header file `hpdi32_api.h`.

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

## 4.1. Files

The API Library is built into a library linkable with HPDI32 applications. The pertinent files are identified in the following table. Some source files are specific only to the HPDI32, some are specific only to the OS and some are HPDI32 and OS independent.

| Description | Files | Location | OS |
|---|---|---|---|
| Source Files | `*.c`, `*.h` | `…/api/` | Linux |
| | | `…\api\` | Windows |
| Header File | `hpdi32_api.h` | `…/include/` | Linux |
| | | `…\include\` | Windows |
| Library File | `libhpdi32_api.so` † | `…/lib/` `/usr/lib/` | Linux |
| | `hpdi32_api.lib` `hpdi32_api.dll` ‡ | `…\lib\…` | Windows |

† The Linux run time executable is implemented as a shared object file.
‡ The Windows run time executable is implemented as a DLL.

## 4.2. Build

For build instructions refer to this same section number in the OS specific HPDI32 driver user manual.

## 4.3. Library Use

For Library usage information refer to this same section number in the OS specific HPDI32 driver user manual.

## 4.4. Macros

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

### 4.4.1. IOCTL Codes

The IOCTL macros are documented in section 4.7 (page 21).

### 4.4.2. Registers

The following gives the complete set of HPDI32 registers.

#### 4.4.2.1. GSC Registers

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

**NOTE**: Refer to the output of the "id" sample application (see section 9 of the OS specific HPDI32 driver user manual) for a complete list of the registers supported by the device being accessed.

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

#### 4.4.2.2. PCI Configuration Registers

Access to the PCI registers is seldom required so these registers are not listed here. For the complete list of the PCI register identifiers refer to header files `gsc_pci9080.h` and `gsc_pci9656.h`, which are automatically included via `hpdi32_api.h`.

#### 4.4.2.3. PLX Feature Set Registers

Access to the PLX registers is seldom required so these registers are not listed here. For the complete list of the PLX register identifiers refer to header files `gsc_pci9080.h` and `gsc_pci9656.h`, which are automatically included via `hpdi32_api.h`.

## 4.5. Data Types

The data types used by the API Library are described with the IOCTL services with which they are used. For additional information refer to section 4.7 (page 21).

## 4.6. Functions

The Library interface includes the functions given in the following subsections. The function return values reflect the completion status of the requested operation. A return value less than zero always reflects an error condition. The table below summarizes the error status values. For the I/O functions, read and write, non-negative return values reflect the number of bytes transferred from or to the device, respectively. A value equal to the requested transfer size indicates complete success. Return values less than the requested transfer size indicate that the I/O timeout expired. For the other API function calls a return value of zero indicates success.

| Return Value | Description | OS |
|---|---|---|
| -1 to -499 | This is the value "(-errno)" (see errno.h). | All |
| -500 to -999 | This is the value returned from the Driver Interface Library. † | Windows |
| <= -1000 | This is "(int)(GetLastError()+1000)" forced to a negative value. | |

† Applicable error codes, if any, are defined in the header os_common.h.

### 4.6.1. hpdi32_close()

This function is the entry point to close a connection made via the API's open call (section 4.6.4, page 18). The device is put in an initialized state before this call returns.

Prototype

```
int hpdi32_close(int fd);
```

| Argument | Description |
|---|---|
| Fd | This is the file descriptor obtained from the open service (section 4.6.4, page 18). |

| Return Value | Description |
|---|---|
| 0 | The operation succeeded. |
| < 0 | An error occurred. See error value descriptions above. |

Example

```
#include <stdio.h>

#include "hpdi32_dsl.h"

int hpdi32_close_dsl(int fd)
{
    int errs;
    int ret;

    ret = hpdi32_close(fd);

    if (ret)
        printf("ERROR: hpdi32_close() returned %d\n", ret);

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

### 4.6.2. hpdi32_init()

This function is the entry point to initializing the HPDI32 API Library and must be the first call into the Library. This function may be called more than once, but only the first successful call actually initializes the library. Subsequent calls perform no operation at all. All other API calls return a failure status when the API Library is not initialized.

Prototype

```
int hpdi32_init(void);
```

| Return Value | Description |
|---|---|
| 0 | The operation succeeded. |
| < 0 | An error occurred. See error value descriptions above. |

Example

```
#include <stdio.h>

#include "hpdi32_dsl.h"

int hpdi32_init_dsl(void)
{
    int errs;
    int ret;

    ret = hpdi32_init();

    if (ret)
        printf("ERROR: hpdi32_init() returned %d\n", ret);

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

### 4.6.3. hpdi32_ioctl()

This function is the entry point to performing setup and control operations on a 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 the `arg` argument. The set of supported options for the `request` argument consists of the IOCTL services supported by the driver, which are defined in section 4.7 (page 21).

**NOTE:** IOCTL operations are not supported for an open on device index `-1`.

Prototype

```
int hpdi32_ioctl(int fd, int request, void* arg);
```

| Argument | Description |
|---|---|
| fd | This is the file descriptor obtained from the open service (section 4.6.4, page 18). |
| request | This specifies the desired operation to be performed (section 4.7, page 21). |
| arg | This is specific to the IOCTL operation specified by the `request` argument. |

| Return Value | Description |
|---|---|
| 0 | The operation succeeded. |
| < 0 | An error occurred. See error value description above. |

Example

```
#include <stdio.h>

#include "hpdi32_dsl.h"

int hpdi32_ioctl_dsl(int fd, int request, void* arg)
```

```
{
    int errs;
    int ret;

    ret = hpdi32_ioctl(fd, request, arg);

    if (ret)
        printf("ERROR: hpdi32_ioctl() returned %d\n", ret);

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

### 4.6.4. hpdi32_open()

This function is the entry point to open a connection to a HPDI32 board. Before returning, the initialize IOCTL service is called to reset all hardware and software settings to their defaults.

Prototype

```
int hpdi32_open(int device, int share, int* fd);
```

| Argument | Description |
|---|---|
| device | This is the zero-based index of the HPDI32 to access. † |
| share | Open the device in Shared Access Mode? If non-zero the device is opened in Shared Access Mode (see below). If zero the device is opened in Exclusive Access Mode (see below). |
| fd | The device handle is returned here. The pointer cannot be NULL. Values returned are as follows. |

| Value | Description |
|---|---|
| >= 0 | This is the handle to use to access the device in subsequent calls. |
| -1 | There was an error. The device is not accessible. |

† The index value -1 can also be given to acquire driver information (section 2.2, page 11).

| Return Value | Description |
|---|---|
| 0 | The operation succeeded. |
| < 0 | An error occurred. See error value descriptions above. |

Example

```
#include <stdio.h>

#include "hpdi32_dsl.h"

int hpdi32_open_dsl(int device, int share, int* fd)
{
    int errs;
    int ret;

    ret = hpdi32_open(device, share, fd);

    if (ret)
        printf("ERROR: hpdi32_open() returned %d\n", ret);
```

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

#### 4.6.4.1. Access Modes

The value of the `share` argument determines the device access mode, as follows.

Shared Access Mode:

Shared Access Mode allows multiple applications to access the same device simultaneously. In this mode, the first successful open request returns with the device in an initialized state. Subsequent successful Shared Access Mode open requests do not affect the state of the device. Once opened in Shared Access Mode, the device access remains in this mode until all Shared Access Mode accesses release the device with a close request.

Exclusive Access Mode:

Exclusive Access Mode allows a single application to acquire exclusive access to a device. In this mode, a successful open request returns with the device in an initialized state. While open in this mode all subsequent open requests will fail regardless of the access mode requested. Once opened in Exclusive Access Mode, the device access remains in this mode until released by the application with a close request.

### 4.6.5. hpdi32_read()

This function is the entry point to reading data from an open connection. The function reads up to `bytes` bytes from the device.

> **NOTE:** If an open was performed using an index of `-1`, then read requests will acquire information from the driver (section 2.2, page 11) rather than data from a device.
>
> **NOTE:** For additional information refer to the Data Transfer Modes section (section 8.1.3, page 51).
>
> **NOTE:** The check for an overflow or an underflow is performed upon entry to the read service. The read service does not check for these conditions that occur while the read is in progress. For in-progress overflows or underflows an application must perform the check manually or wait for the check performed by a subsequent read request.

Prototype

```
int hpdi32_read(int fd, void* dst, size_t bytes);
```

| Argument | Description |
|---|---|
| `fd` | This is the file descriptor obtained from the open service (section 4.6.4, page 18). |
| `dst` | The data read is put here. |
| `bytes` | This is the desired number of bytes to read. When reading from a device, this must be a multiple of the configured Rx data size (section 4.7.22, page 31). |

| Return Value | Description |
|---|---|
| `0` to `bytes` | The operation succeeded. When reading from a device, a value less than `bytes` indicates that the I/O timeout period (section 4.7.26, page 33) lapsed before the entire request could be satisfied. |
| `< 0` | An error occurred. See error value description above. |

Example

```
#include <stdio.h>

#include "hpdi32_dsl.h"

int hpdi32_read_dsl(int fd, void* dst, size_t bytes, size_t* qty)
{
    int errs;
    int ret;

    ret = hpdi32_read(fd, dst, bytes);

    if (ret < 0)
        printf("ERROR: hpdi32_read() returned %d\n", ret);

    if (qty)
        qty[0]  = (ret < 0) ? 0 : (size_t) ret;

    errs    = (ret < 0) ? 1 : 0;
    return(errs);
}
```

### 4.6.6. hpdi32_write()

This function is the entry point to reading data from an open connection. The function writes up to `bytes` bytes to the device.

> **NOTE:** For additional information refer to the Data Transfer Modes section (section 8.1.3, page 51).

> **NOTE:** The check for an overflow is performed upon entry to the write service. The write service does not check for this condition that occurs while the write is in progress. For in-progress overflows an application must perform the check manually or wait for the check performed by a subsequent write request.

> **NOTE:** Write operations are not supported for an open on device index `-1`.

Prototype

```
int hpdi32_write(int fd, const void* src, size_t bytes);
```

| Argument | Description |
|---|---|
| `fd` | This is the file descriptor obtained from the open service (section 4.6.4, page 18). |
| `src` | The data to write is taken from this pointer. |
| `bytes` | This is the desired number of bytes to write. This must be a multiple of the configured Tx data size (section 4.7.43, page 39). |

| Return Value | Description |
|---|---|
| `0` to `bytes` | The operation succeeded. A value less than `bytes` indicates that the I/O timeout period (section 4.7.47, page 40) lapsed before the entire request could be satisfied. |
| `< 0` | An error occurred. See error value description above. |

Example

```
#include <stdio.h>

#include "hpdi32_dsl.h"

int hpdi32_write_dsl(int fd, const void* src, size_t bytes, size_t*
qty)
{
    int errs;
    int ret;

    ret = hpdi32_write(fd, src, bytes);

    if (ret < 0)
        printf("ERROR: hpdi32_write() returned %d\n", ret);

    if (qty)
        qty[0]  = (ret < 0) ? 0 : (size_t) ret;

    errs    = (ret < 0) ? 1 : 0;
    return(errs);
}
```

## 4.7. IOCTL Services

The HPDI32 API Library and device driver implement the following IOCTL services. Each service is described along with the applicable `hpdi32_ioctl()` function arguments.

### Transmitter Operation and Settings

For general information on the transmitter, the transmitter cable signals and the transmitter I/O settings refer to sections 8.3 (page 52) and 8.4 (page 53). These include concise tables and descriptions along with references to the respective IOCTL services.

### Receiver Operation and Settings

For general information on the receiver, the receiver cable signals and the receiver I/O settings refer to sections 8.5 (page 60) and 8.6 (page 60). These include concise tables and descriptions along with references to the respective IOCTL services.

#### 4.7.1. HPDI32_IOCTL_CABLE_CMD_MODE_*n*

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

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_CABLE_CMD_MODE_*n* |
| arg | s32* |

Valid argument values are as follows.

| Value | Description |
|---|---|
| -1 | Retrieve the current setting. |
| HPDI32_CABLE_CMD_MODE_FC | This refers to the Flow Control option. If the GPIO feature is not supported, then this is the option returned. |
| HPDI32_CABLE_CMD_MODE_IN | This refers to the GPIO Input option. |
| HPDI32_CABLE_CMD_MODE_OUT_LO | This refers to the GPIO Output Low option. |
| HPDI32_CABLE_CMD_MODE_OUT_HI | This refers to the GPIO Output High option. |

### 4.7.2. HPDI32_IOCTL_CABLE_CMD_STATE_*n*

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

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_CABLE_CMD_STATE_*n* |
| arg | s32* |

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

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

### 4.7.3. HPDI32_IOCTL_GPIO_D32_OUTPUT

This service configures the GPIO D32 Output feature. This service refers only to the output portion of GPIO D32 feature, as the input portion is always active.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_GPIO_D32_OUTPUT |
| arg | s32* |

Valid argument values include the options listed below.

| Value | Description |
|---|---|
| -1 | Retrieve the current setting. This value is returned if the feature is unsupported. |
| HPDI32_GPIO_D32_OUTPUT_DISABLE | This option disables the feature so the GPIO Output Register content is not driven to the cable interface. |
| HPDI32_GPIO_D32_OUTPUT_ENABLE | This option enables the feature so the GPIO Output Register content is driven to the cable interface. |

### 4.7.4. HPDI32_IOCTL_INITIALIZE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_INITIALIZE |
| arg | Not used. |

### 4.7.5. HPDI32_IOCTL_IRQ_CONFIG_EDGE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_IRQ_CONFIG_EDGE |
| arg | s32* |

Valid argument values include any bitwise combination of the bits defined for the HPDI32_IOCTL_IRQ_ENABLE service (section 4.7.7, page 24), or -1 to retrieve the current configurations. The argument value -1 is returned if the device feature is unsupported.

### 4.7.6. HPDI32_IOCTL_IRQ_CONFIG_HIGH

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_IRQ_CONFIG_HIGH |
| arg | s32* |

Valid argument values include any bitwise combination of the bits defined for the HPDI32_IOCTL_IRQ_ENABLE service (section 4.7.7, page 24), or -1 to retrieve the current configurations. The argument value -1 is returned if the device feature is unsupported.

### 4.7.7. HPDI32_IOCTL_IRQ_ENABLE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_IRQ_ENABLE |
| arg | s32* |

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

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

### 4.7.8. HPDI32_IOCTL_QUERY

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_QUERY |
| arg | s32* |

Valid argument values are as follows.

| Value | Description |
|---|---|
| HPDI32_QUERY_BSR_D18_XCVR | This indicates if BSR bit D18 reports the transceiver selection. |
| HPDI32_QUERY_BUS_WIDTH | This returns the device's PCI interface bus width in bits, which is either 32 or 64. |
| HPDI32_QUERY_CLOCK_MAX | This returns the maximum cable interface clock rate in hertz, which is either 25,000,000 or 50,000,000. |
| HPDI32_QUERY_COUNT | This returns the number of query options supported by the IOCTL service. |
| HPDI32_QUERY_DEVICE_TYPE | This returns the identifier value for the device's type. This should be GSC_DEV_TYPE_HPDI32. |
| HPDI32_QUERY_DMDMA_1 | This indicates if Demand Mode DMA is supported on DMA channel 1. |
| HPDI32_QUERY_FIFO_SIZE_RX | This indicates the depth of the Rx FIFO in 32-bit words. A value of zero indicates the FIFO size in not known. |
| HPDI32_QUERY_FIFO_SIZE_TX | This indicates the depth of the Tx FIFO in 32-bit words. A value of zero indicates the FIFO size in not known. |
| HPDI32_QUERY_FORM_FACTOR | This indicates the device's native form factor. The options are listed below. |
| HPDI32_QUERY_GPIO_0_5 | This indicates if the GPIO 0 to 5 configuration options are available for Cable Command signals 1 through 6, respectively. |
| HPDI32_QUERY_GPIO_6 | This indicates if the GPIO 6 configuration option is available for Cable Command signal 0. |
| HPDI32_QUERY_GPIO_D32 | This indicates if the GPIO D32 feature is supported. |
| HPDI32_QUERY_IRQ_CONFIG_REGS | This indicates if the firmware supports the Interrupt Edge/Level Register and the Interrupt High/Low Register. |
| HPDI32_QUERY_MODEL | This indicates the device's basic model number. The options are listed below. |
| HPDI32_QUERY_OVER_UNDER_RUN | This indicates if the firmware supports the Rx FIFO Overrun bit (BSR D21), the Rx FIFO Underrun bit (BSR D22), and the Tx FIFO Overrun bit (BSR D23). |
| HPDI32_QUERY_REG_FSR | This indicates if the firmware supports the Feature Set Register. |
| HPDI32_QUERY_REG_IELR | This indicates if the firmware supports the Interrupt High/Low Register. |
| HPDI32_QUERY_REG_IHLR | This indicates if the firmware supports the Interrupt Edge/Level Register. |
| HPDI32_QUERY_REG_RFSR | This indicates if the firmware supports the Rx FIFO Size Register. |
| HPDI32_QUERY_REG_RFWR | This indicates if the firmware supports the Rx FIFO Words Register. |
| HPDI32_QUERY_REG_TCDR | This indicates if the firmware supports the Tx Clock Divider Register. |
| HPDI32_QUERY_REG_TFSR | This indicates if the firmware supports the Tx FIFO Size Register. |
| HPDI32_QUERY_REG_TFWR | This indicates if the firmware supports the Tx FIFO Words Register. |
| HPDI32_QUERY_SINGLE_CYC_DIS | This indicates if the firmware supports the Single Cycle Disable Bit (BCR D11). |
| HPDI32_QUERY_TRISTATE_TE_RE | This indicates if the firmware supports the Tristate Tx Enabled/Rx Enabled feature (BCR D31). |
| HPDI32_QUERY_TX_AUTO_STOP | This indicates if the firmware supports the Tx Start Auto-Clear Disable bit (BCR D6). |

| | |
|---|---|
| HPDI32_QUERY_TX_CLOCK_DIV_MAX | This indicates the maximum supported value for the Tx Clock Divider. |
| HPDI32_QUERY_TX_CLOCK_DIV_MIN | This indicates the minimum supported value for the Tx Clock Divider. |
| HPDI32_QUERY_USER_JUMPERS | This indicates if the firmware supports the User Jumper status bits (BSR D16 and D17). |
| HPDI32_QUERY_XCVR_TYPE | This indicates the device's transceiver type. The options are listed below. |

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

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

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

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

Valid return values for the Model query are as follows.

| Value | Description |
|---|---|
| HPDI32_QUERY_MODEL_HPDI32 | The device is an HPDI32. |
| HPDI32_QUERY_MODEL_HPDI32A | The device is an HPDI32A. |
| HPDI32_QUERY_MODEL_HPDI32AL | The device is an HPDI32AL. |
| HPDI32_QUERY_MODEL_HPDI32ALT | The device is an HPDI32ALT. |
| HPDI32_QUERY_MODEL_HPDI32B | The device is an HPDI32B. |

Valid return values for the Transceiver query are as follows.

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

### 4.7.9. HPDI32_IOCTL_REG_MOD

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_REG_MOD |
| arg | gsc_reg_t* |

Definition

```
typedef struct
{
    u32 reg;
    u32 value;
    u32 mask;
} gsc_reg_t;
```

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

### 4.7.10. HPDI32_IOCTL_REG_READ

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_REG_READ |
| arg | gsc_reg_t* |

Definition

```
typedef struct
{
    u32 reg;
    u32 value;
    u32 mask;
} gsc_reg_t;
```

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

### 4.7.11. HPDI32_IOCTL_REG_WRITE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_REG_WRITE |
| arg | gsc_reg_t* |

Definition

```
typedef struct
```

```
{
    u32 reg;
    u32 value;
    u32 mask;
} gsc_reg_t;
```

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

### 4.7.12. HPDI32_IOCTL_RX_AUTO_START

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_AUTO_START |
| arg | s32* |

Valid argument values include the options listed below.

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

### 4.7.13. HPDI32_IOCTL_RX_ENABLE

This service enables or disables the receiver.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_ENABLE |
| arg | s32* |

Valid argument values include the options listed below.

| Value | Description |
|---|---|
| -1 | Retrieve the current setting. |
| HPDI32_RX_ENABLE_NO | This disables the receiver. |
| HPDI32_RX_ENABLE_YES | This enables the receiver. |

### 4.7.14. HPDI32_IOCTL_RX_FIFO_AE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_FIFO_AE |
| arg | s32* |

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

> **NOTE**: The Almost Empty level should never be set to a value less than 15. Doing so could result is corrupted data being retrieved from the device.

#### 4.7.15. HPDI32_IOCTL_RX_FIFO_AF

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_FIFO_AF |
| arg | s32* |

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

#### 4.7.16. HPDI32_IOCTL_RX_FIFO_OVERRUN

This service operates on the Rx FIFO overrun status.

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_FIFO_OVERRUN |
| arg | s32* |

Valid argument values supplied to the service are as follows.

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

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

| Value | Description |
|---|---|
| -1 | The feature is not supported. |
| HPDI32_FIFO_ERROR_NO | The FIFO has not experienced an overrun condition. |
| HPDI32_FIFO_ERROR_YES | The FIFO has experienced an overrun condition. |

### 4.7.17. HPDI32_IOCTL_RX_FIFO_RESET

This service resets the Rx FIFO, which clears the content and applies any pending Almost Empty or Almost Full settings. It also clears the associated overrun and underrun status bits.

> **NOTE:** When Almost Empty and Almost Full settings are applied via their respective IOCTL services, the driver automatically performs a FIFO reset to apply the changes. The Almost Empty level is adjusted via the service `HPDI32_IOCTL_RX_FIFO_AE` (section 4.7.14, page 28). The Almost Full level is adjusted via the service `HPDI32_IOCTL_RX_FIFO_AF` (section 4.7.15, page 29).

Usage

| Argument | Description |
|---|---|
| `request` | `HPDI32_IOCTL_RX_FIFO_RESET` |
| `arg` | Not used. |

### 4.7.18. HPDI32_IOCTL_RX_FIFO_STATUS

This service retrieves the current Rx FIFO fill level status.

Usage

| Argument | Description |
|---|---|
| `request` | `HPDI32_IOCTL_RX_FIFO_STATUS` |
| `arg` | `s32*` |

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

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

### 4.7.19. HPDI32_IOCTL_RX_FIFO_UNDERRUN

This service operates on the Rx FIFO underrun status.

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

Usage

| Argument | Description |
|---|---|
| `request` | `HPDI32_IOCTL_RX_FIFO_UNDERRUN` |
| `arg` | `s32*` |

Valid argument values supplied to the service are as follows.

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

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

| Value | Description |
|---|---|
| -1 | The device feature is unsupported. |
| HPDI32_FIFO_ERROR_NO | The FIFO has not experienced an underrun condition. |
| HPDI32_FIFO_ERROR_YES | The FIFO has experienced an underrun condition. |

### 4.7.20. HPDI32_IOCTL_RX_IO_ABORT

This service aborts an ongoing `hpdi32_read()` request.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_ABORT |
| arg | s32* |

The results are reported as one of the following values.

| Value | Description |
|---|---|
| HPDI32_IO_ABORT_NO | An `hpdi32_read()` request was not aborted as none were ongoing. |
| HPDI32_IO_ABORT_YES | An ongoing `hpdi32_read()` request was aborted. |

### 4.7.21. HPDI32_IOCTL_RX_IO_BMDMA_THRESH

This service sets the minimum DMA transfer size used during Block Mode DMA based read requests. As such read requests may consist of multiple smaller DMA transfers, this setting limits the smallest size of those individual transfers. This setting does not apply to the last DMA transfer of the read request. The unit of measure for this setting is bytes.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_BMDMA_THRESH |
| arg | s32* |

Valid argument values are from zero to the depth of the Rx FIFO in 32-bit words, or `-1` to retrieve the current setting. The default is `132` bytes.

### 4.7.22. HPDI32_IOCTL_RX_IO_DATA_SIZE

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

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_DATA_SIZE |
| arg | s32* |

Valid argument values supplied to the service are as follows.

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

### 4.7.23. HPDI32_IOCTL_RX_IO_MODE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_MODE |
| arg | s32* |

Valid argument values supplied to the service are as follows.

| Value | Description |
|---|---|
| -1 | Retrieve the current setting. |
| GSC_IO_MODE_BMDMA | Data is retrieved using Block Mode DMA. In this mode the data must be present in the FIFO before the transfer is initiated. |
| GSC_IO_MODE_DMDMA | Data is retrieved using Demand Mode DMA. In this mode the request can exceed the FIFO size as data is retrieved from the FIFO as it becomes available. This is the default. † |
| GSC_IO_MODE_PIO | Data is retrieved using repetitive register accesses. |

† On 64-bit HPDI32 devices the DMA engine performs DMDMA transfers in 64-bit increments. On 32-bit devices DMDMA is performed in 32-bit increments. (The increment sizing is not enforced at the end of DMDMA transfers that are not sized on such boundaries.) To insure DMDMA data integrity the HPDI32 pauses DMDMA transfers when the Rx FIFO fill level falls to the Almost Empty level and resumes the transfer when the fill level rises above the Almost Empty level. For non-continuous data streams applications may need to accommodate the pause by retrieving the last Almost Empty number of data values using PIO or DMA.

### 4.7.24. HPDI32_IOCTL_RX_IO_OVERRUN

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

**NOTE:** This setting is ignored if the device doesn't support the Rx FIFO Overrun feature.

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_OVERRUN |
| arg | s32* |

Valid argument values are as follows.

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

### 4.7.25. HPDI32_IOCTL_RX_IO_PIO_THRESHOLD

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_PIO_THRESHOLD |
| arg | s32* |

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

### 4.7.26. HPDI32_IOCTL_RX_IO_TIMEOUT

This service sets the timeout limit for read requests. The value is expressed in seconds. The timeout limit is the total amount of time allowed for a single hpdi32_read() request. When this time limit has expired the service terminates. When this occurs the hpdi32_read() return value will be less than the number of bytes requested, and possibly zero.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_TIMEOUT |
| arg | s32* |

Valid argument values are in the range from zero to 3600, -1, and HPDI32_IO_TIMEOUT_INFINITE. A value of zero tells the driver not to sleep in order to wait for more data, and should only be used with PIO mode reads. A value of -1 is used to retrieve the current setting. If the option HPDI32_IO_TIMEOUT_INFINITE is used, then the driver waits indefinitely rather than timing out. The default is 10 seconds.

### 4.7.27. HPDI32_IOCTL_RX_IO_UNDERRUN

This service operates on the Rx FIFO underrun status.

**NOTE:** This setting is ignored if the device doesn't support the Rx FIFO Underrun feature.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_IO_UNDERRUN |
| arg | s32* |

Valid argument values supplied to the service are as follows.

| Value | Description |
|---|---|
| -1 | Retrieve the current state. |
| HPDI32_IO_ERROR_CHECK | Check the underrun status. This is the default. |
| HPDI32_IO_ERROR_IGNORE | Ignore the current status. |

### 4.7.28. HPDI32_IOCTL_RX_LINE_COUNT

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_LINE_COUNT |
| arg | s32* |

The count returned is from zero to 0xFFFFFFFF.

### 4.7.29. HPDI32_IOCTL_RX_STATUS_COUNT

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_RX_STATUS_COUNT |
| arg | s32* |

The count returned is from zero to 0xFFFFFFFF.

### 4.7.30. HPDI32_IOCTL_TRISTATE_TE_RE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TRISTATE_TE_RE |
| arg | s32* |

Valid argument values include the options listed below.

| Value | Description |
|---|---|
| -1 | Retrieve the current setting. Returned if the feature is unsupported. |
| HPDI32_TRISTATE_TE_RE_NO | The signals are driven on the cable interface. |
| HPDI32_TRISTATE_TE_RE_YES | The signals are tristated. |

### 4.7.31. HPDI32_IOCTL_TX_AUTO_START

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_AUTO_START |
| arg | s32* |

Valid argument values include the options listed below.

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

### 4.7.32. HPDI32_IOCTL_TX_AUTO_STOP

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

> **NOTE:** This service is operable only if it is supported by firmware. Refer to the HPDI32_IOCTL_QUERY IOCTL service option HPDI32_QUERY_TX_AUTO_STOP (section 4.7.8, page 24) to determine if this feature is supported by firmware.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_AUTO_STOP |
| arg | s32* |

Valid argument values include the options listed below.

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

### 4.7.33. HPDI32_IOCTL_TX_CLOCK_DIVIDER

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_CLOCK_DIVIDER |
| arg | s32* |

Valid argument values are in the range from zero to0xFFFF, and -1. A value of -1 is used to retrieve the current setting. The argument value -1 is returned if the feature is unsupported.

> **NOTE:** If the Clock Divider is zero, then the Tx Clock frequency equals the on-board oscillator frequency. Otherwise, the Tx Clock frequency is governed by the formula given below. In the formula, $F_{TxC}$ is the Tx Clock frequency, $F_{Osc}$ is the on-board oscillator frequency, and Divider is the Tx Clock Divider value.

$$F_{TxC} = F_{Osc} / (Divider * 2)$$

### 4.7.34. HPDI32_IOCTL_TX_ENABLE

This service enables or disables the transmitter.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_ENABLE |
| arg | s32* |

Valid argument values include the options listed below.

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

### 4.7.35. HPDI32_IOCTL_TX_FIFO_AE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_FIFO_AE |
| arg | s32* |

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

### 4.7.36. HPDI32_IOCTL_TX_FIFO_AF

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_FIFO_AF |
| arg | s32* |

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

> **NOTE**: The Almost Full value should never be set to a value less than 16. Doing so could result is data loss while writing to the device.

### 4.7.37. HPDI32_IOCTL_TX_FIFO_OVERRUN

This service operates on the Tx FIFO overrun status.

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_FIFO_OVERRUN |
| arg | s32* |

Valid argument values supplied to the service are as follows.

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

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

| Value | Description |
|---|---|
| -1 | The device feature is unsupported. |
| HPDI32_FIFO_ERROR_NO | The FIFO has not experienced an overrun condition. |
| HPDI32_FIFO_ERROR_YES | The FIFO has experienced an overrun condition. |

### 4.7.38. HPDI32_IOCTL_TX_FIFO_RESET

This service resets the Tx FIFO, which clears the content and applies any pending Almost Empty or Almost Full settings. It also clears the associated overrun status bit.

> **NOTE:** When Almost Empty or Almost Full settings are applied via their respective IOCTL services, the driver automatically performs a FIFO reset to apply the changes. The Almost Empty level is adjusted via the service HPDI32_IOCTL_TX_FIFO_AE (section 4.7.35, page 36). The Almost Full level is adjusted via the service HPDI32_IOCTL_TX_FIFO_AF (section 4.7.36, page 36).

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_FIFO_RESET |
| arg | Not used. |

### 4.7.39. HPDI32_IOCTL_TX_FIFO_STATUS

This service retrieves the current Tx FIFO fill level status.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_FIFO_STATUS |
| arg | s32* |

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

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

### 4.7.40. HPDI32_IOCTL_TX_FLOW_CONTROL

This service starts and stops data transmission. The transmitter must be enabled for data transmission to be started (HPDI32_IOCTL_TX_ENABLE, section 4.7.34, page 36).

> **NOTE:** If this service is used to stop data transmission, then the desired operation may be negated if the Auto Start feature is enabled (see section 4.7.31, page 35).

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_FLOW_CONTROL |
| arg | s32* |

Valid argument values include the options listed below.

| Value | Description |
|---|---|
| -1 | Retrieve the current setting. |
| HPDI32_TX_FLOW_CONTROL_START | Data transmission is enabled. |
| HPDI32_TX_FLOW_CONTROL_STOP | Data transmission is disabled. |

### 4.7.41. HPDI32_IOCTL_TX_IO_ABORT

This service aborts an ongoing hpdi32_write() request.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_IO_ABORT |
| arg | s32* |

The results are reported as one of the following values.

| Value | Description |
|---|---|
| HPDI32_IO_ABORT_NO | An hpdi32_write() request was not aborted as none were ongoing. |
| HPDI32_IO_ABORT_YES | An ongoing hpdi32_write() request was aborted. |

### 4.7.42. HPDI32_IOCTL_TX_IO_BMDMA_THRESH

This service sets the minimum DMA transfer size used during Block Mode DMA based write requests. As such write requests may consist of multiple smaller DMA transfers, this setting limits the smallest size of those individual transfers. This setting does not apply to the last DMA transfer of the write request. The unit of measure for this setting is bytes.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_IO_BMDMA_THRESH |
| arg | s32* |

Valid argument values are from zero to the depth of the Tx FIFO in 32-bit words, or -1 to retrieve the current setting. The default is 132 bytes.

### 4.7.43. HPDI32_IOCTL_TX_IO_DATA_SIZE

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

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_IO_DATA_SIZE |
| arg | s32* |

Valid argument values supplied to the service are as follows.

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

### 4.7.44. HPDI32_IOCTL_TX_IO_MODE

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_IO_MODE |
| arg | s32* |

Valid argument values supplied to the service are as follows.

| Value | Description |
|---|---|
| -1 | Retrieve the current setting. |
| GSC_IO_MODE_BMDMA | Data is moved using Block Mode DMA. In this mode the data written must fit in the FIFO before the transfer is initiated. |
| GSC_IO_MODE_DMDMA | Data is moved using Demand Mode DMA. In this mode the request can exceed the FIFO size as data is moved to the FIFO as space becomes available. This is the default. |
| GSC_IO_MODE_PIO | Data is written using repetitive register accesses. |

### 4.7.45. HPDI32_IOCTL_TX_IO_OVERRUN

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

**NOTE:** This setting is ignored if the device doesn't support the Tx FIFO Overrun feature.

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_IO_OVERRUN |
| arg | s32* |

Valid argument values are as follows.

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

### 4.7.46. HPDI32_IOCTL_TX_IO_PIO_THRESHOLD

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_IO_PIO_THRESHOLD |
| arg | s32* |

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

### 4.7.47. HPDI32_IOCTL_TX_IO_TIMEOUT

This service sets the timeout limit for write requests. The value is expressed in seconds. The timeout limit is the total amount of time allowed for a single hpdi32_write() request. When this time limit has expired the service

terminates. When this occurs the `hpdi32_write()` return value will be less than the number of bytes requested, and possibly zero.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_IO_TIMEOUT |
| arg | s32* |

Valid argument values are in the range from zero to `3600`, `-1`, and `HPDI32_IO_TIMEOUT_INFINITE`. A value of zero tells the driver not to sleep in order to wait for more space, and should only be used with PIO mode reads. A value of `-1` is used to retrieve the current setting. If the option `HPDI32_IO_TIMEOUT_INFINITE` is used, then the driver waits indefinitely rather than timing out. The default is `10` seconds.

### 4.7.48. HPDI32_IOCTL_TX_LINE_VAL_OFF_CNT

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

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_LINE_VAL_OFF_CNT |
| arg | s32* |

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

### 4.7.49. HPDI32_IOCTL_TX_LINE_VAL_ON_CNT

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

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_TX_LINE_VAL_ON_CNT |
| arg | s32* |

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

### 4.7.50. HPDI32_IOCTL_TX_REMOTE_THROTTLE

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

**NOTE:** The device's Tx Flow Control feature (`HPDI32_IOCTL_TX_FLOW_CONTROL`, section 4.7.40, page 38) operates in parallel with the Remote Throttle feature. When using Remote Throttling the Tx Flow Control feature should be set to stop data flow and the Tx Auto Start feature (`HPDI32_IOCTL_TX_AUTO_START`, section 4.7.31, page 35) should be disabled.

Usage

| Argument | Description |
|---|---|
| `request` | `HPDI32_IOCTL_TX_REMOTE_THROTTLE` |
| `arg` | `s32*` |

Valid argument values are as follows.

| Value | Description |
|---|---|
| `-1` | Retrieve the current setting. |
| `HPDI32_TX_REMOTE_THROTTLE_NO` | This option disables remote throttling. |
| `HPDI32_TX_REMOTE_THROTTLE_YES` | This option enables remote throttling. |

### 4.7.51. HPDI32_IOCTL_TX_STATUS_VAL_CNT

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

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

Usage

| Argument | Description |
|---|---|
| `request` | `HPDI32_IOCTL_TX_STATUS_VAL_CNT` |
| `arg` | `s32*` |

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

### 4.7.52. HPDI32_IOCTL_TX_STATUS_VAL_MIR

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

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

Usage

| Argument | Description |
|---|---|
| `request` | `HPDI32_IOCTL_TX_STATUS_VAL_MIR` |
| `arg` | `s32*` |

Valid argument values are as follows.

| Value | Description |
|---|---|
| `-1` | Retrieve the current setting. |

| | |
|---|---|
| HPDI32_TX_STATUS_VAL_MIR_NO | While asserted, the Status Valid signal is not mirrored onto the Line Valid signal. The Line Valid signal remains negated while the Status Valid signal is asserted. |
| HPDI32_TX_STATUS_VAL_MIR_YES | While asserted, the Status Valid signal is mirrored onto the Line Valid signal. The Line Valid signal is asserted while the Status Valid signal is asserted. |

### 4.7.53. HPDI32_IOCTL_USER_JUMPERS

This service retrieves the status for the user installable jumpers.

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_USER_JUMPERS |
| arg | s32* |

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

| Value | Description |
|---|---|
| 0x0 | Either neither jumper is installed or the device doesn't support the jumper feature. † |
| 0x1 | Only jumper one is installed |
| 0x2 | Only jumper two is installed |
| 0x3 | Both jumpers are installed |

† Refer to the HPDI32_IOCTL_QUERY IOCTL service's HPDI32_QUERY_USER_JUMPERS option to determine if the device supports the user jumper feature (section 4.7.8, page 24).

### 4.7.54. HPDI32_IOCTL_WAIT_CANCEL

This service resumes all threads blocked via HPDI32_IOCTL_WAIT_EVENT IOCTL calls (section 4.7.55, page 44), according to the provided criteria. When a blocked thread is waiting for any event specified in the structure, then the thread is resumed.

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_WAIT_CANCEL |
| arg | gsc_wait_t* |

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

| Fields | Description |
|---|---|
| flags | This is unused by wait cancel operations. |
| main | This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be cancelled. Refer to section 4.7.55.2 (page 45). |
| gsc | This specifies the set of HPDI32_WAIT_GSC_* events whose wait requests are to be cancelled. Refer to section 4.7.55.3 (page 45). |
| alt | This is unused by the HPDI32 driver and should be zero. |
| io | This specifies the set of HPDI32_WAIT_IO_* events whose wait requests are to be cancelled. Refer to section 4.7.55.4 (page 46). |
| timeout_ms | This is unused by wait cancel operations. |
| count | Upon return this indicates the number of waits that were cancelled. |

### 4.7.55. HPDI32_IOCTL_WAIT_EVENT

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

**NOTE**: The service waits only for the first of the specified events, not for all specified events.

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

Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_WAIT_EVENT |
| arg | gsc_wait_t* |

Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
} gsc_wait_t;
```

| Fields | Description |
|---|---|
| flags | This must initially be zero. Upon return this indicates the reason that the thread was resumed. Refer to section 4.7.55.1 (page 45). |
| main | This specifies any number of GSC_WAIT_MAIN_* events that the thread is to wait for. Refer to section 4.7.55.2 (page 45). |
| gsc | This specifies any number of HPDI32_WAIT_GSC_* events that the thread is to wait for. Refer to section 4.7.55.3 (page 45). |
| alt | This is unused by the HPDI32 driver and must be zero. |
| io | This specifies any number of HPDI32_WAIT_IO_* events that the thread is to wait for. Refer to section 4.7.55.4 (page 46). |

| | |
|---|---|
| timeout_ms | This specified the maximum amount of time, in milliseconds, that the thread is to wait for any of the referenced events. A value of zero means do not timeout at all. If non-zero, then upon return the value is the approximate amount of time actually waited. |
| count | This is unused by wait event operations and must be zero. |

#### 4.7.55.1. gsc_wait_t.flags Options

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

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

#### 4.7.55.2. gsc_wait_t.main Options

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

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

#### 4.7.55.3. gsc_wait_t.gsc Options

The wait structure's gsc field may specify any combination of the below interrupt options. These are the interrupt options referenced in the Interrupt Control Register. Applications are responsible for enabling the desired interrupt options. Refer to HPDI32_IOCTL_IRQ_ENABLE (section 4.7.7, page 24). If a device supports the Interrupt Edge/Level Register then interrupts configured for level triggering are disabled by the driver when they occur. If a device does not support this register, then all firmware interrupts are disabled by the driver when they occur.

| Value | Description |
|---|---|
| HPDI32_WAIT_GSC_CC0_FV_E_GPIO6 | This refers to the Cable Command 0 signal, whose default configuration is to trigger on a falling edge. This signal may be configured as Frame Valid or GPIO 6. |
| HPDI32_WAIT_GSC_CC0_FV_S_GPIO6 | This refers to the Cable Command 0 signal, whose default configuration is to trigger on a rising edge. This signal may be configured as Frame Valid or GPIO 6. |
| HPDI32_WAIT_GSC_CC1_LV_GPIO0 | This refers to the Cable Command 1 signal. This signal may be configured as Line Valid or GPIO 0. |
| HPDI32_WAIT_GSC_CC2_SV_GPIO1 | This refers to the Cable Command 2 signal. This signal may be configured as Status Valid or GPIO 1. |
| HPDI32_WAIT_GSC_CC3_RR_GPIO2 | This refers to the Cable Command 3 signal. This signal may be configured as Rx Ready or GPIO 2. |
| HPDI32_WAIT_GSC_CC4_TR_GPIO3 | This refers to the Cable Command 4 signal. This signal may be configured as Tx Data Ready or GPIO 3. |

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

#### 4.7.55.4. gsc_wait_t.io Options

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

| Fields | Description |
|---|---|
| HPDI32_WAIT_IO_RX_ABORT | This refers to read requests which have been aborted. |
| HPDI32_WAIT_IO_RX_DONE | This refers to read requests which have been satisfied. |
| HPDI32_WAIT_IO_RX_ERROR | This refers to read requests which end due to an error. |
| HPDI32_WAIT_IO_RX_TIMEOUT | This refers to read requests which end due to the timeout period lapse. |
| HPDI32_WAIT_IO_TX_ABORT | This refers to write requests which have been aborted. |
| HPDI32_WAIT_IO_TX_DONE | This refers to write requests which have been satisfied. |
| HPDI32_WAIT_IO_TX_ERROR | This refers to write requests which end due to an error. |
| HPDI32_WAIT_IO_TX_TIMEOUT | This refers to write requests which end due to the timeout period lapse. |

### 4.7.56. HPDI32_IOCTL_WAIT_STATUS

This service counts all threads blocked via the HPDI32_IOCTL_WAIT_EVENT IOCTL service (section 4.7.55, page 44), according to the provided criteria. A match is made when a waiting thread's wait criteria matches any of the criteria specified in the structure passed to this service.

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

#### Usage

| Argument | Description |
|---|---|
| request | HPDI32_IOCTL_WAIT_STATUS |
| arg | gsc_wait_t* |

#### Definition

```
typedef struct
{
    u32  flags;
    u32  main;
    u32  gsc;
    u32  alt;
    u32  io;
    u32  timeout_ms;
    u32  count;
```

```
} gsc_wait_t;
```

| Fields | Description |
|---|---|
| flags | This is unused by wait status operations. |
| main | This specifies the set of GSC_WAIT_MAIN_* events whose wait requests are to be counted. Refer to section 4.7.55.2 (page 45). |
| gsc | This specifies the set of HPDI32_WAIT_GSC_* events whose wait requests are to be counted. Refer to section 4.7.55.3 (page 45). |
| alt | This is unused by the HPDI32 driver and should be zero. |
| io | This specifies the set of HPDI32_WAIT_IO_* events whose wait requests are to be counted. Refer to section 4.7.55.4 (page 46). |
| timeout_ms | This is unused by wait status operations. |
| count | Upon return this indicates the number of waits that met any of the specified criteria. |

# 5. The Driver

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

## 5.1. Files

The driver is built into an OS specific executable. The pertinent files are identified in the following table. Some source files are specific only to the HPDI32, some are specific only to the OS and some are HPDI32 and OS independent.

**NOTE:** The driver source files are not included with the Windows driver.

| Description | Files | Location | OS |
|---|---|---|---|
| Source Files | `*.c`, `*.h` | `…/driver/` | Linux |
| Header File | `hpdi32.h` | `…/driver/` | Linux |
| | | `…\include\` | Windows |
| Driver File | `hpdi32.ko` † | `…/driver/` | Linux (kernels version 2.6 and later) |
| | `hpdi32.o` † | `…/driver/` | Linux (kernels version 2.4 and earlier) |
| | `hpdi32_dil.lib`<br>`hpdi32_dil.dll` | `…\lib\…` | Windows |
| | `hpdi32_9080.sys` ‡<br>`hpdi32_9656.sys` ‡ | `…\driver\…` | |

† The Linux run time executable is implemented as a loadable kernel module.
‡ The Windows run time executables are implemented as driver `.sys` files.

## 5.2. Build

For instructions on building the driver refer to this same section number in the OS specific HPDI32 driver user manual.

## 5.3. Startup

For instructions on starting the driver executable refer to this same section number in the OS specific HPDI32 driver user manual.

## 5.4. Verification

For instructions on verifying that the driver has been loaded and is running refer to this same section number in the OS specific HPDI32 driver user manual.

## 5.5. Version

For instructions on obtaining the driver version number refer to this same section number in the OS specific HPDI32 driver user manual.

## 5.6. Shutdown

For instructions on terminating the driver executable refer to this same section number in the OS specific HPDI32 driver user manual.

# 6. Document Source Code Examples

The source code examples included in this document are built into a statically linkable library usable with console applications. 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.

## 6.1. Files

The library files are summarized in the table below.

| Description | Files | Location | OS |
|---|---|---|---|
| Source Files | `*.c`, `*.h` … | `…/docsrc/` | Linux |
| | | `…\docsrc\` | Windows |
| Header File | `hpdi32_dsl.h` | `…/include/` | Linux |
| | | `…\include\` | Windows |
| Library File | `hpdi32_dsl.a` | `…/lib/` | Linux |
| | `hpdi32_dsl.lib` | `…\lib\…` | Windows |

## 6.2. Build

For library build instructions refer to this same section number in the OS specific HPDI32 driver user manual.

## 6.3. Library Use

For library usage information refer to this same section number in the OS specific HPDI32 driver user manual.

# 7. Utilities Source Code

The API Library installation includes a body of utility source code designed to aid in the understanding and use of the interface calls and IOCTL services. Utility sources are also included for device independent and common, general-purpose services. Most of the utilities are implemented as visual wrappers around the corresponding services to facilitate structured console output for the sample applications. The utility sources are compiled and linked into static libraries to simplify their use. An additional purpose of these utility services is to provide a library of working sample code to assist in a user's learning curve and application development effort.

For each API function there is a corresponding utility source file with a corresponding utility service. As an example, for the API function `hpdi32_open()` there is the utility file `open.c` containing the utility function `hpdi32_open_util()`. The naming pattern is as follows: API function `hpdi32_xxxx()`, utility file name `xxxx.c`, utility function `hpdi32_xxxx_util()`. Additionally, for each IOCTL code there is a corresponding utility source file with a corresponding utility service. As an example, for IOCTL code `HPDI32_IOCTL_QUERY` there is the utility file `util_query.c` containing the utility function `hpdi32_query()`. The naming pattern is as follows: IOCTL code `HPDI32_IOCTL_XXXX`, utility file name `util_xxxx.c`, utility function `hpdi32_xxxx()`.

## 7.1. Files

The library files are summarized in the table below.

| Description | Files | Location | OS |
|---|---|---|---|
| Source Files | `*.c, *,h …` | `…/utils/` | Linux |
| | | `…\utils\` | Windows |
| Header File | `hpdi32_utils.h` | `…/include/` | Linux |
| | | `…\include\` | Windows |
| Library File | `hpdi32_utils.a`<br>`gsc_utils.a`<br>`os_utils.a`<br>`plx_utils.a` | `…/lib/` | Linux |
| | `hpdi32_utils.lib`<br>`gsc_utils.lib`<br>`os_utils.lib`<br>`plx_utils.lib` | `…\lib\…` | Windows |

## 7.2. Build

For library build instruction refer to this same section number in the OS specific HPDI32 driver user manual.

## 7.3. Library Use

For library usage information refer to this same section number in the OS specific HPDI32 driver user manual.

# 8. Operating Information

This section explains some basic operational procedures for using the HPDI32. This is in no way intended to be a comprehensive guide. This is simply to address a very few issues relating to their use.

## 8.1. Debugging Aids

The driver package includes the following items useful for development and/or debugging aids.

### 8.1.1. Device Identification

When communicating with technical support complete device identification is virtually always necessary. The *id* example application is provided for this specific purpose. This is a text only console application. The output can be piped to a file, which can then be emailed to GSC technical support when requested. Locate the application as follows.

| Description | File | Location | OS |
|---|---|---|---|
| Application | `id` | `…/id/` | Linux |
| | `id.exe` | `…\id\…` | Windows |

### 8.1.2. API Listing

Among the utility services provided is a function to generate a listing to the console of all API settings and all register contents. When used, the function is typically used to verify the device configuration. In these cases, the function should be called just prior to the first read or write operation. The function arguments are as follows. The utility location is given in the subsequent table.

| Argument | Description |
|---|---|
| `fd` | This is the file descriptor used to access the device. |

| Description | File/Name | Location | OS |
|---|---|---|---|
| Function | `hpdi32_api_listing()` | Source File | All |
| Source File | `util_api_listing.c` | `…/utils/` | All |
| Header File | `hpdi32_utils.h` | `…/include/` | All |
| Library File | `hpdi32_utils.a` | `…/lib/` | Linux |
| | `hpdi32_utils.lib` | `…\lib\…` | Windows |

### 8.1.3. Detailed Register Dump

Among the utility services provided is a function to generate a detailed listing of device registers to the console. When used, the function is typically used to verify device configuration. In these cases, the function should be called after complete device configuration and before the first I/O call. When intended for sending to GSC tech support, please set the *detail* arguments to `1`. The function arguments are as follows. The utility location is given in the subsequent table.

| Argument | Description |
|---|---|
| `fd` | This is the file descriptor used to access the device. |
| `detail` | If non-zero the register dump will include details of each register field. |

| Description | File/Name | Location | OS |
|---|---|---|---|
| Function | `hpdi32_reg_list()` | Source File | All |
| Source File | `util_reg.c` | `…/utils/` | Linux |
| | | `…\utils\` | Windows |

| | | | |
|---|---|---|---|
| Header File | hpdi32_utils.h | …/include/ | Linux |
| | | …\include\ | Windows |
| Library File | hpdi32_utils.a | …/lib/ | Linux |
| | hpdi32_utils.lib | …\lib\ | Windows |

## 8.2. I/O Modes

> **NOTE**: Transfer requests below a configurable limit are automatically performed using PIO mode. This is because PIO is more efficient for very small transfers. The default driver defined limits are 32 samples for both reads and write. However, the transition points are configurable via the HPDI32_IOCTL_TX_IO_PIO_THRESHOLD (section 4.7.46, page 40) and HPDI32_IOCTL_RX_IO_PIO_THRESHOLD services (section 4.7.25, page 33).

### 8.2.1. PIO - Programmed I/O

This mode transfers data by repetitive register accesses. Of the modes supported this is the least efficient. However, it is the only mode that can be used with an I/O Timeout setting of zero.

### 8.2.2. BMDMA - Block Mode DMA

This option uses block mode DMA transfers to fulfill application I/O requests. In this mode, movement of data by the DMA engine is initiated for reads only after the data is already present in the Rx FIFO or for writes when space is available in Tx FIFO. In this mode the driver subdivides requests, as needed, based on the current state of the respective FIFO. Consequently, each I/O request may consist of a single DMA transfer, a few DMA transfers or of many DMA transfers. Typically, the lower the transfer clock relative to bus activity between the card and the host, the smaller and more numerous the number of transfers.

> **NOTE**: The Tx and Rx Block Mode DMA Threshold settings are used to limit the smallest size of individual DMA transfers. This is done to help improve efficiency. The BMDMA default exceeds the equivalent PIO Threshold default to limit the use of PIO to those instances where it is required to complete I/O requests. This too is done to help improve efficiency. For additional information refer to section 4.7.42 (page 39, Tx BMDMA Threshold), section 4.7.21 (page 31, Rx BMDMA Threshold), section 4.7.46 (page 40, Tx PIO Threshold) and section 4.7.25 (page 33, Rx PIO Threshold).

### 8.2.3. DMDMA - Demand Mode DMA

In this DMA mode the transfers are subdivided and started differently. First, subdividing is done only as needed to limit individual transfers to the size of the Tx and Rx transfer buffers (8MB each) maintained by the driver for each device. Second, the transfers are started without waiting for data to arrive in the Rx FIFO or for space to become available in the Tx FIFO. In this mode the DMA engine moves Rx data as it appears in the Rx FIFO and moves Tx data as space appears in the Tx FIFO.

## 8.3. Basic Transmit Configuration

The basic steps needed for transmitter configuration are illustrated in the utility function noted below. The table also gives the location of the source file, the header file and the corresponding library containing the executable code.

| Description | File/Name | Location | OS |
|---|---|---|---|
| Function | hpdi32_config_tx() | Source File | All |
| Source File | util_config_tx.c | …/utils/ | Linux |
| | | …\utils\ | Windows |
| Header File | hpdi32_utils.h | …/include/ | Linux |
| | | …\include\ | Windows |

| | | | |
|---|---|---|---|
| Library File | hpdi32_utils.a | …/lib/ | Linux |
| | hpdi32_utils.lib | …\lib\… | Windows |

## 8.4. Basic Transmitter Operation

A basic illustration of the transmitter and the transmitter cable signals is given in Figure 2 below.

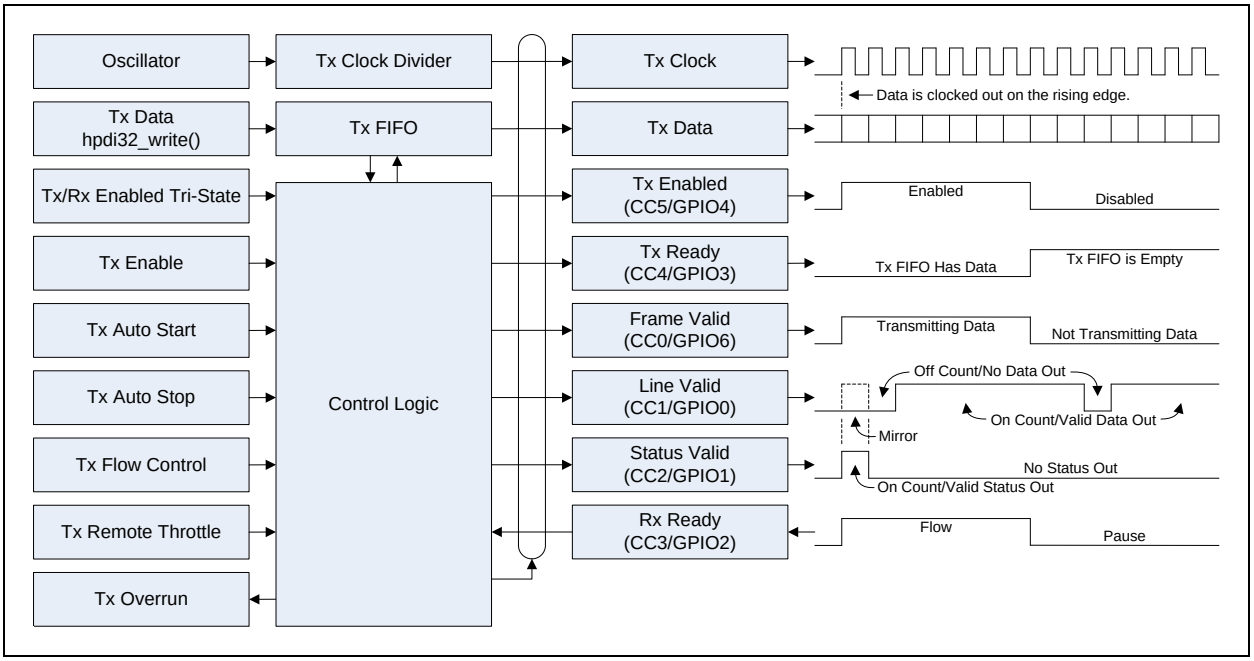


**Figure 2** A depiction of the HPDI32 transmitter and the transmitter cable signals.

### 8.4.1. Transmitter Cable Signals

The transmitter cable interface operates as described in the table below. The signals are described according to their flow control functionality. If a cable signal is not configured for flow control operation (it may be GPIO) then the corresponding flow control feature is disabled. While the transmitter is disabled the transmitter cable signal functionalities are disabled. In this case, the output signals are tristated and the inputs are ignored. The exception is the Tx Enabled signal, which is active even when the transmitter is disabled. While the transmitter is enabled, the signals are driven according to how they are configured, with some also depending on the presence of transmit data in the Tx FIFO. The cable signals are configured via the Cable Command Mode IOCTL services (section 4.7.1, page 21). The signal states, high or low, are readable via the Cable Command State IOCTL services (section 4.7.2, page 22) regardless of how the signals are configured.

| Signal | Description |
|---|---|
| Tx Clock | The Tx Clock signal is driven at all times while the transmitter is enabled. For information on the Tx Clock frequency refer to section 4.7.33, page 35. |
| Tx Data | These are the 32 signals representing the data being transmitted. The signals are driven when the transmitter is enabled and tristated when the transmitter is disabled. When driven, all 32 signals are active, even if the I/O data size is set for eight-bits or 16-bits. Transmit data is posted to the cable interface on the rising edge of the Tx Clock signal. When data is not being clock out the data content is indeterminate. The source of the transmit data is the Tx FIFO, which receives data via the `hpdi32_write()` API Library call. |

| | |
|---|---|
| Tx Enabled | This signal is driven high when the transmitter is enabled and is driven low when it is disabled. If the Tx/Rx Enabled Tristate setting is enabled, then this signal is tristated rather than being driven. This is primarily for cases where two HPDI32 devices are connected back-to-back. |
| Tx Ready | This signal reflects the availability of transmit data. The signal is driven low when data is present in the Tx FIFO and it is driven high when the Tx FIFO is empty. The signal is driven when the transmitter is enabled and tristated when the transmitter is disabled. |
| Frame Valid | This is the primary "data valid" signal. The signal is driven high at the Tx Clock rising edge when data is being clocked onto the cable interface. The signal is driven low at the Tx Clock rising edge when data is not being clocked onto the cable interface. The signal is driven when the transmitter is enabled and tristated when the transmitter is disabled. † |
| Line Valid | This is a secondary "data valid" signal intended for data organized into frames. The signal is driven low at the Tx Clock rising edge for *Line Valid Off* consecutive clocks. Then, the signal is driven high at the Tx Clock rising edge for *Line Valid On* clocks as line data is being clocked onto the cable interface. This off/on pattern appears immediately after the *Status Valid On* period and is repeated until the end of the frame. This feature is disabled if either of the *Line Valid Off* or *Line Valid On* count is zero. The signal is driven when the transmitter is enabled and tristated when the transmitter is disabled. † |
| Status Valid | This is a secondary "data valid" signal intended for data organized into frames. The signal is driven high at the Tx Clock rising edge for Status Valid On clocks as status data is being clocked onto the cable interface. The signal is then driven low for the remainder of the frame. The feature is disabled if the Status Valid On count is set to zero. The signal is driven when the transmitter is enabled and tristated when the transmitter is disabled. † |
| Rx Ready | This is a flow control signal driven by the receiving device. The receiving device drives the signal high to enable data transmission and drives it low to inhibit data transmission. This transmitter feature is enabled by enabling the Tx Remote Throttle setting. The signal is usable when the transmitter is enabled and ignored when the transmitter is disabled. |

† If this signal is not enabled, then it is not used to quality transmission of data. If none of these signals are enabled, then there is no means of distinguishing between valid data and indeterminate data.

### 8.4.2. Transmitter Cable Signals - continuous unstructured data stream

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

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

The simplest configuration usable for a continuous unstructured data stream is illustrated in Figure 3. This configuration uses the Tx Clock, Tx Data and Frame Valid signals, while all of the other transmitter signals are unused. Even in this simplest configuration the unused signals must be configured, even if simply configured so that they are unused by the transmitter. The easiest way to do this is to configure the unused signals as general-purpose inputs.

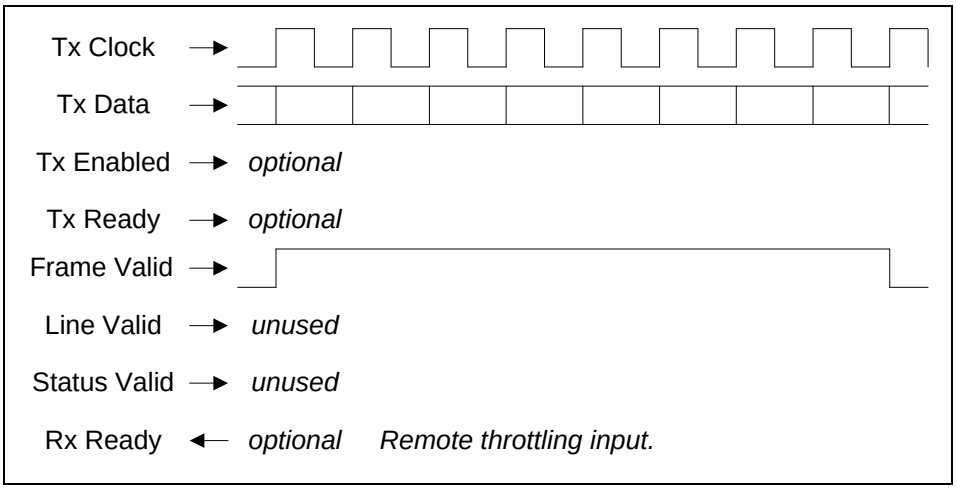


**Figure 3** A simple continuous unstructured data stream cable configuration.

#### 8.4.2.1. Tx Clock

The Tx Clock output signal is the clock that synchronizes the transmitter logic and which clocks transmit data out the cable interface. The signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled, the signal is not driven by the HPDI32. This signal is required for data transmission and cannot be disabled or reconfigured. The Tx Clock is derived from the on-board oscillator, which is routed through the Tx Clock Divider. If the divider is zero, then the Tx Clock frequency equals the on-board oscillator frequency. Otherwise, the Tx Clock frequency is governed by the formula $F_{TxC} = F_{Osc} / (Div * 2)$. In the formula, $F_{TxC}$ is the Tx Clock frequency, $F_{Osc}$ is the on-board oscillator frequency, and `Div` is the Tx Clock Divider value (see section 4.7.33, page 35).

#### 8.4.2.2. Tx Data

The Tx Data output signals are synchronized with the Tx Clock to transmit 32-bits of parallel data out the cable interface. The signals are driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled, the signals are not driven by the HPDI32. These signals are required for transmission of data. The transmitter clocks out the data on Tx Clock's rising edge. See Figure 4. The transmitter hardware has a 32-bit data path, including the FIFOs and the cable transceivers, though the source data width is configurable (section 4.7.43, page 39). When the source data is less than 32-bits wide, it is aligned with the D0 bit and passed through the transmitter as 32-bit data words. When the source data is 8-bits wide the data appears on cable signals D0 through D7. The upper 24 data signals can be ignored, though they are driven by the transmitter. When the source data is 16-bits wide the data appears on cable signals D0 through D15. The upper 16 data signals can be ignored, though they are driven by the transmitter.

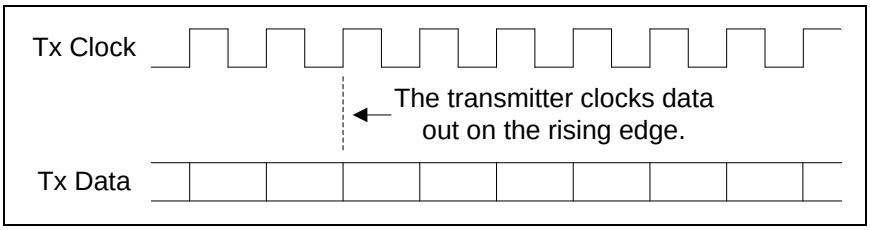


**Figure 4** Tx Data is synchronized with Tx Clock.

When not used for data transmission, the data lines can be used as a 32-bit wide discrete digital output port (see section 4.7.3, page 22), when supported by firmware (see `HPDI32_IOCTL_GPIO_D32_OUTPUT`, section 4.7.8, page 24). When configured as discrete output, the port output is updated by writing to the GPIO Output Register.

#### 8.4.2.3. Tx Enabled

The Tx Enabled output signal reflects the enabled state of the transmitter. The signal is normally driven on the cable interface to indicate that the transmitter is enabled or disabled, but can be tristated when the transmitter is disabled. This signal is not required for Flow Control of continuous, unstructured data streams. The Tx Enabled signal is driven high when the transmitter is enabled and is driven low when disabled. See Figure 5. However, to avoid possible contention with other devices, the signal can be configured to become tristated when the transmitter is enabled (section 4.7.30, page 34). Signal state changes are not synchronized with Tx Clock.

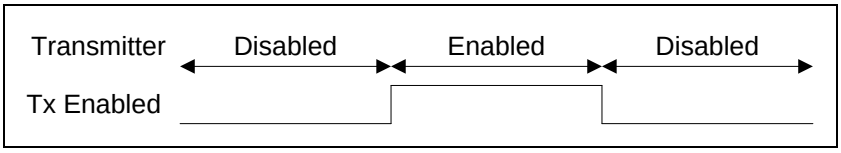


**Figure 5** The Tx Enabled signal reflects the transmitter enable state.

The Tx Enabled signal is the default functionality of the cable pin identified in the board user manual as Cable Command 5 (CC5). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 4. When the pin is not used it is recommended that it be configured as a GPIO input.

#### 8.4.2.4. Tx Ready

The Tx Ready output signal reflects the availability of data from the transmitter. The signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled, the signal is not driven by the HPDI32. This signal is not required for Flow Control of continuous, unstructured data streams. When it is used, it is driven high when the Tx FIFO is empty and is driven low when the Tx FIFO has data. See Figure 6. The signal state changes are not synchronized with Tx Clock.

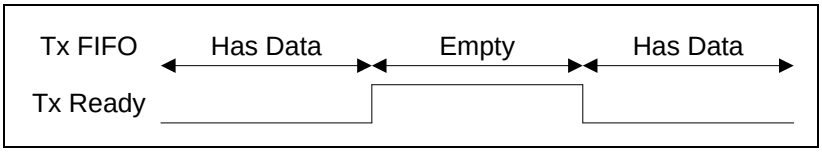


**Figure 6** The Tx Ready signal reflects the Tx FIFO empty state.

The Tx Ready signal is the default functionality of the cable pin identified in the board user manual as Cable Command 4 (CC4). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 3. When the pin is not used it is recommended that it be configured as a GPIO input.

#### 8.4.2.5. Frame Valid

The Frame Valid output signal indicates when data is being transmitted out the cable interface. The signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled, the signal is not driven by the HPDI32. This signal is virtually always used for Flow Control purposes when data is to be transmitted. The signal is driven high when data is being transmitted and is driven low otherwise. See Figure 7. The signal is synchronized with Tx Clock and changes state on the clock's rising edge.

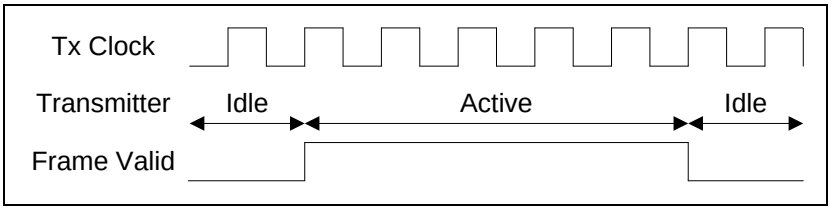


**Figure 7** The Frame Valid signal reflects the data transmission process.

The Frame Valid signal is the default functionality of the cable pin identified in the board user manual as Cable Command 0 (CC0). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 6. When the pin is not used it is recommended that it be configured as a GPIO input.

> **NOTE**: Technically, it is possible to transmit data without using the Frame Valid signal. In this case transmission is governed by whatever other flow control signals are configured for use. If none are, then data is transmitted at the present Tx Clock rate without any flow control whatever. This is a way to achieve discrete output functionality when the D32 GPIO feature is not supported by firmware. This is also a means to performs tests at the board's maximum transmission rate.

#### 8.4.2.6. Line Valid

The Line Valid output signal reflects valid transmit data being presented at the cable interface. The signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled, the signal is not driven by the HPDI32. This signal is not required for Flow Control of continuous, unstructured data streams. When it is used, it is driven high to indicate that data is being clocked out the cable interface and is driven low otherwise. See Figure 8. The signal is synchronized with Tx Clock and changes state on the clock's rising edge.

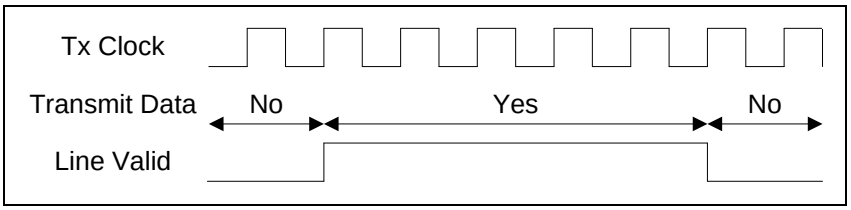


**Figure 8** The Line Valid signal reflects valid transmit data being presented at the cable interface.

The Line Valid signal is the default functionality of the cable pin identified in the board user manual as Cable Command 1 (CC1). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 0. When the pin is not used it is recommended that it be configured as a GPIO input. An additional means of negating the transmitter's Line Valid functionality is by setting the Line Valid On Count and Line Valid Off Count values to zero (see section 4.7.49, page 41 and section 4.7.48, page 41, respectively).

#### 8.4.2.7. Status Valid

The Status Valid output signal reflects valid status data being transmitted out the cable interface. The signal is driven on the cable interface only when the transmitter is enabled. When the transmitter is disabled, the signal is not driven by the HPDI32. This signal is not required for Flow Control of continuous, unstructured data streams. When it is used, it is driven high when status data is being clocked out the cable interface and is driven low otherwise. See Figure 9. The signal is synchronized with Tx Clock and changes state on the clock's rising edge.

> **NOTE**: Status Data is data clocked out while the Status Valid signal is asserted. Refer to the board user manual for additional information.

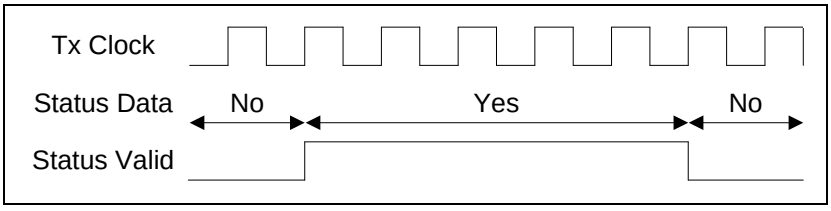


**Figure 9** The Status Valid signal reflects valid status data being presented at the cable interface.

The Status Valid signal is the default functionality of the cable pin identified in the board user manual as Cable Command 2 (CC2). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 1. When the pin is not used it is recommended that it be configured as a GPIO input. An additional means of negating the transmitter's Status Valid functionality is by setting the Status Valid On Count value to zero (see section 4.7.51, page 42).

#### 8.4.2.8. Rx Ready

The Rx Ready input signal may be used by receiving devices to pause the flow of data from the HPDI32 transmitter. As a flow control signal, the transmitter responds to the input from the cable interface only when the transmitter is enabled. When the transmitter is disabled, the transmitter ignores the signal. This signal is not required for Flow Control of continuous, unstructured data streams. The Rx Ready signal is driven high by the remote receiver to permit data flow and is driven low to pause data flow. See Figure 10. The signal is synchronized with Tx Clock such that state changes are clocked in on the clock's rising edge. Data flow neither pauses nor resumes instantaneously. It may take several Tx Clock cycles for transmission to pause or resume due to the firmware's internal pipeline.

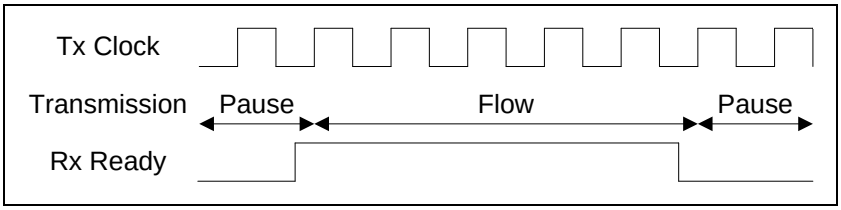


**Figure 10** The remote receiving device can drive the Rx Ready signal to control data flow.

The Rx Ready signal is the default functionality of the cable pin identified in the board user manual as Cable Command 3 (CC3). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 2. When the pin is not used it is recommended that it be configured as a GPIO input.

### 8.4.3. Transmitter Configuration Settings

The following is a general description of the transmitter configuration settings.

| Setting | Description |
|---|---|
| Tx Auto Start | If this setting is enabled, then the `hpdi32_write()` service initiates data transmission via the *Tx Flow Control - Start* feature. This setting should usually be enabled. (See section 4.7.31, page 35.) |
| Tx Auto Stop | When available in firmware, disabling this setting permits data transmission to continue without interruption even though the Tx FIFO may temporarily become empty. *This setting should virtually always be disabled.* (See section 4.7.32, page 35.) |
| Tx Clock Divider | This setting is the value by which the master clock is divided to generate the Tx Clock. (See section 4.7.33, page 35.) |
| Tx Enable | This is the master setting that enables or disables the transmitter. (See section 4.7.34, page 36.) |
| Tx FIFO Almost Empty | During transmission, the transmitter slows or pauses transmission when the Tx FIFO fill level falls below this level. This is done to prevent the transmission pipeline from reading from an empty Tx FIFO. Applications can adjust this threshold and can use the state as an interrupt source. The level should never be reduced below the default. (See section 4.7.35, page 36.) |

| | |
|---|---|
| Tx FIFO Almost Full | During data transfer, the transmitter slows or pauses DMDMA data movement into the Tx FIFO if its fill level exceeds this setting. This is done to prevent the transmission pipeline from overflowing the Tx FIFO. Applications can adjust this threshold and can use the state as an interrupt source. The level should never be reduced below the default. (See section 4.7.36, page 36.) |
| Tx FIFO Overrun | This setting allows applications to query for, and clear, a Tx FIFO Overrun. During normal operation the driver prevents overruns. (See section 4.7.37, page 37.) |
| Tx Flow Control | This setting is a manual means of starting or stopping data transmission. The *Stop* option should be selected while the Tx Remote Throttle feature is used. (See section 4.7.40, page 38.) † |
| Tx Line Valid Off Count | This setting adjusts the Line Valid signal's *Off* count. If set to zero, the Line Valid feature is disabled. (See section 4.7.48, page 41.) |
| Tx Line Valid On Count | This setting adjusts the Line Valid signal's *On* count. If set to zero, the Line Valid feature is disabled. (See section 4.7.49, page 41.) |
| Tx Remote Throttle | This setting controls the transmitter's use of the Rx Ready signal. If enabled, the remote device can drive the signal for hardware flow control. The *Disable* setting should be selected while the Tx Flow Control feature is used. (See section 4.7.50, page 41.) † |
| Tx Status Valid Count | This setting adjusts the Status Valid On count. If set to zero, the Status Valid feature is disabled. (See section 4.7.51, page 42.) |
| Tx Status Valid Mirror | This setting controls the state of the Line Valid signal while the Status Valid signal is driven high. If the *On* setting is selected, then the Line Valid signal is driven high while the Status Valid signal is driven high. The Status Valid signal otherwise remains low. (See section 4.7.52, page 42.) |

† The Tx Flow Control and Tx Remote Throttle features operate independently and in parallel so should not be used simultaneously.

### 8.4.4. Transmitter I/O Settings

The following is a general description of the transmitter I/O settings, which are the settings affecting the operation of the `hpdi32_write()` service.

| **Setting** | **Description** |
|---|---|
| Tx I/O Data Size | This setting specifies the size of each data item transferred over the cable interface. The options are 8-bits, 16-bits and 32-bits. This setting affects only the transfer of data from the host to the device as the Tx FIFO does not perform data packing and the cable interface always drives all 32 data lines. Data alignment is such that data bit D0 always corresponds to cable data signal D0. (See section 4.7.42, page 39.) |
| Tx I/O Mode | This service selects the method the driver uses to transfer data from the host to the device. The options are PIO, Block Mode DMA and Demand Mode DMA. (See section 4.7.44, page 39, and section 8.1.3, page 51.) |
| Tx I/O Overrun | This setting configures the `hpdi32_write()` service's response to Tx FIFO overruns. If the *check* option is selected, the default, then the service checks for a Tx FIFO overrun before the service starts transferring data to the device. If the *ignore* option is selected, then the check is not performed. During normal operation the driver prevents overruns. (See section 4.7.45, page 40.) |
| Tx I/O DMA Threshold | This setting limits the size of the smallest individual DMA transfers. This is done to help improve efficiency. The BMDMA default exceeds the equivalent PIO Threshold default to limit the use of PIO to those instances where it is required to complete I/O requests. (See section 4.7.42, page 39.) |
| Tx I/O PIO Threshold | For very small data transfers the driver automatically changes DMA requests to PIO. This is done because PIO is more efficient for very small write requests. This service configures the threshold at which that change takes place. The default is 32 data values. (See section 4.7.46, page 40.) |

| | |
|---|---|
| Tx I/O Timeout | This setting specifies the maximum duration of each `hpdi32_write()` request. (See section 4.7.47, page 40.) |
| Tx/Rx Enable Tristate | This setting controls the operation of the Tx Enabled and Rx Enabled cable signals when either the transmitter or receiver is enabled, respectively. With the *No* setting, each signal is driven. With the *Yes* settings each signal is tristated rather than being driven. (See section 4.7.30, page 34.) |

## 8.5. Basic Receive Configuration

The basic steps needed for receiver configuration are illustrated in the utility function noted below. The table also gives the location of the source file, the header file and the corresponding library containing the executable code.

| Description | File/Name | Location | OS |
|---|---|---|---|
| Function | `hpdi32_config_rx()` | Source File | All |
| Source File | `util_config_rx.c` | `…/utils/` | Linux |
| | | `…\utils\` | Windows |
| Header File | `hpdi32_utils.h` | `…/include/` | Linux |
| | | `…\include\` | Windows |
| Library File | `hpdi32_utils.a` | `…/lib/` | Linux |
| | `hpdi32_utils.lib` | `…\lib\…` | Windows |

## 8.6. Basic Receiver Operation

A basic illustration of the receiver and the receiver cable signals is given in Figure 11 below.

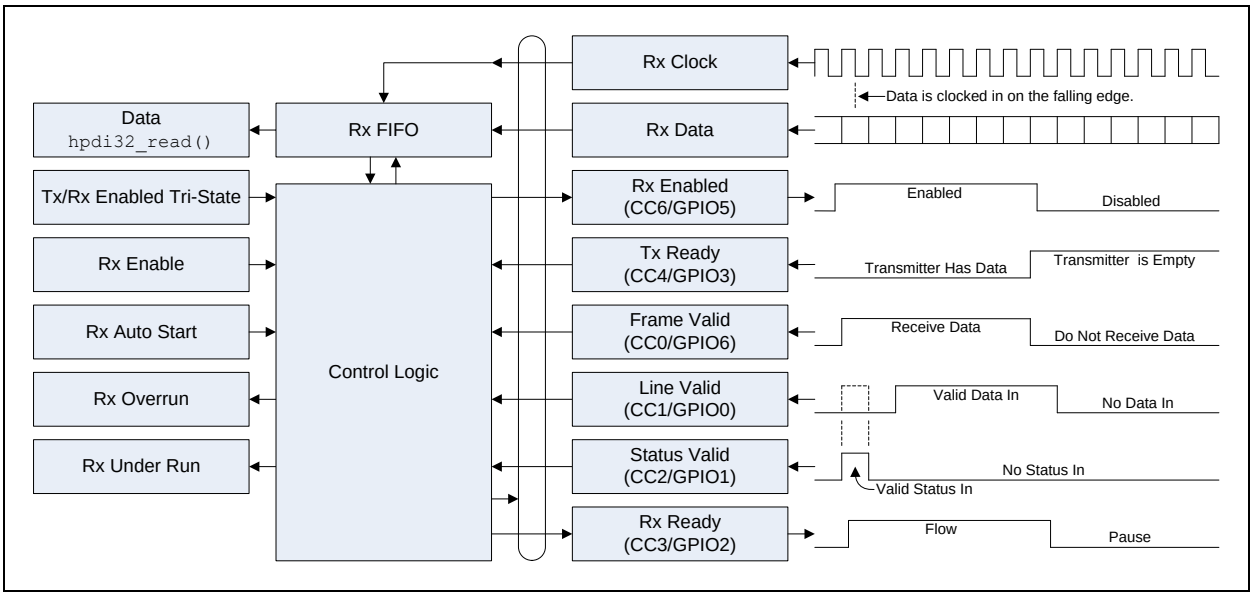


**Figure 11** A depiction of the HPDI32 receiver and the receiver cable signals.

### 8.6.1. Receiver Cable Signals

The receiver cable interface operates as described in the table below. The signals are described according to their flow control functionality. If a cable signal is not configured for flow control operation (it may be GPIO) then the corresponding flow control feature is disabled. While the receiver is disabled the receiver cable signal functionalities are disabled. In this case, the output signals are tristated and the inputs are ignored. The exception is the Rx Enabled signal, which is active even when the receiver is disabled. While the receiver is enabled, the signals are driven according to how they are configured, with some also depending on the volume of data in the Rx FIFO. The cable

signals are configured via the Cable Command Mode IOCTL services (section 4.7.1, page 21). The signal states, high or low, are readable via the Cable Command State IOCTL services (section 4.7.2, page 22) regardless of how the signals are configured.

| Signal | Description |
|---|---|
| Rx Clock | The Rx Clock signal is observed only while the receiver is enabled. The signal is ignored when the receiver is disabled. |
| Rx Data | These are the 32 signals representing the data being received. The signals are observed as data is clocked in and are ignored otherwise. When observed, all 32 signals are used, even if the I/O data size is set for eight-bits or 16-bits. Receive data is taken from the cable interface on the falling edge of the Rx Clock signal. The destination for the receive data is the Rx FIFO, which returns data via the `hpdi32_read()` API Library call. |
| Rx Enabled | This signal is driven high when the receiver is enabled and is driven low when it is disabled. If the Tx/Rx Enabled Tristate setting is enabled, then this signal is tristated rather than being driven. This is primarily for cases where two HPDI32 devices are connected back-to-back. |
| Tx Ready | This signal reflects the availability of transmit data and is driven by the transmitting device. The receiver does not make use of this signal. However, the signal state can be used by software to control operation of an application. |
| Frame Valid | This is the primary "data valid" input signal. The signal is high at the Rx Clock falling edge when data is to be clocked in from the cable interface. The signal is low at the Rx Clock falling edge when data is not to be clocked in from the cable interface. The signal is used when the receiver is enabled and is ignored when the receiver is disabled. † |
| Line Valid | This is a secondary "data valid" signal intended for data organized into frames. The signal is high at the Rx Clock falling edge when data is to be clocked in from the cable interface. The signal is low at the Rx Clock falling edge when data is not to be clocked in from the cable interface. The signal is observed when the receiver is enabled and is ignored when the receiver is disabled. † |
| Status Valid | This is a secondary "data valid" input signal intended for data organized into frames. At the start of a frame, the signal is high at the Rx Clock falling edge when status data is to be clocked in from the cable interface. After the start of a frame, the signal remains low till the end of the frame. The signal is observed when the receiver is enabled and is ignored when the receiver is disabled. † |
| Rx Ready | This is a hardware flow control signal driven by the receiver. The receiver drives the signal high to enable data transmission and drives it low to inhibit data transmission. The source of the signal is the Rx FIFO Almost Full status. The signal is driven when the receiver is enabled and is tristated when the receiver is disabled. |

† If this signal is not enabled, then it is not used to quality reception of data. If none of these signals are enabled, then there is no means of distinguishing between valid data and indeterminate data.

### 8.6.2. Receiver Cable Signals - continuous unstructured data stream

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

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

The simplest configuration usable for a continuous unstructured data stream is illustrated in Figure 12. This configuration uses the Rx Clock, Rx Data and Frame Valid signals, while all of the other receiver signals are unused. Even in this simplest configuration the unused signals must be configured, even if simply configured so that

they are unused by the receiver. The easiest way to do this is to configure the unused signals as general-purpose inputs.

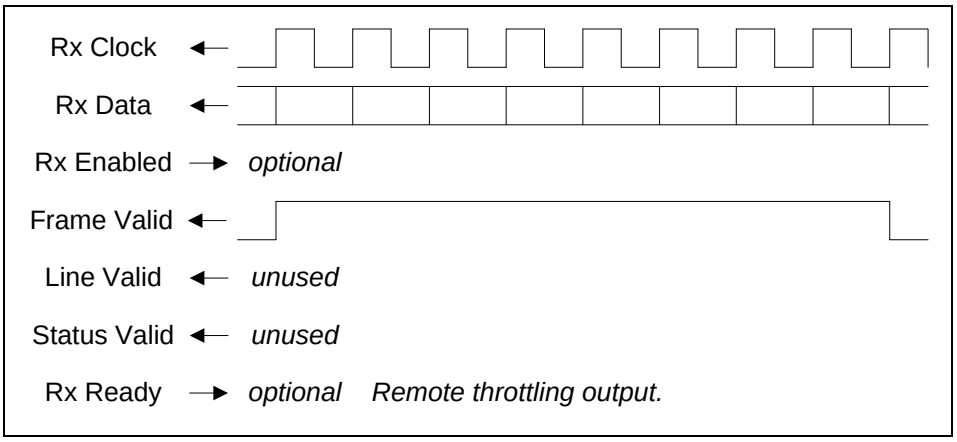


**Figure 12** A simple continuous unstructured data stream cable configuration.

#### 8.6.2.1. Rx Clock

The Rx Clock input signal is the clock that synchronizes the receiver logic and which clocks data in from the cable interface. This signal is required in order for the receiver to capture data presented to the cable interface. This signal must be driven by the transmitting device. If not, then the receiver cannot capture input data. The input is ignored when the receiver is disabled and must be driver when the receiver is enabled.

#### 8.6.2.2. Rx Data

The Rx Data input signals are synchronized with the Rx Clock to capture 32-bits of parallel data. As inputs from the cable interface, the receiver responds to the signals only when the receiver is enabled. When the receiver is disabled, the receiver ignores the signals. These signals must be driven by the transmitting device or the receiver cannot capture input data. The receiver clocks in data on Rx Clock's falling edge. See Figure 13. It is expected that the transmitting device clocks data out on the clock's rising edge. The receiver hardware has a 32-bit data path, including the FIFOs and the cable transceivers, though the source data width is configurable (section 4.7.22, page 31). When the source data is less than 32-bits wide, it is aligned with the D0 bit and passed through the receiver as full 32-bit data words. When the source data is 8-bits wide it appears on cable signals D0 through D7. The upper 24 data signals are recorded, but can be ignored. When the source data is 16-bits wide it appears on cable signals D0 through D15. The upper 16 data signals are recorded, but can be ignored.

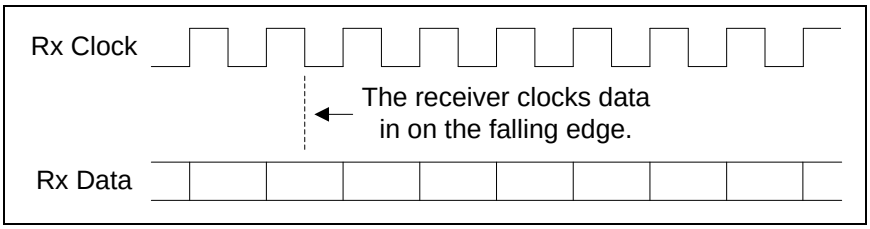


**Figure 13** Rx Data is synchronized with Rx Clock.

When not used for data reception, the data lines can be used as a 32-bit wide discrete digital input port, when supported by firmware (see `HPDI32_IOCTL_GPIO_D32_OUTPUT`, section 4.7.8, page 24). When used in this way, the port input is obtained by reading the GPIO Input Register. No action is required to make use of the discrete input as the receiver and the GPIO input port can be used simultaneously. Thus, an application may choose to designate some lines for clocked input and others for discrete input.

### 8.6.2.3. Rx Enabled

The Rx Enabled output signal reflects the enabled state of the receiver. The signal is normally driven on the cable interface to indicate that the receiver is enabled or disabled, but can be tristated when the receiver is disabled. This signal is not required for Flow Control of continuous, unstructured data streams. The Rx Enabled signal is driven high when the receiver is enabled and is driven low when disabled. See Figure 14. However, to avoid possible contention with other devices, the signal can be configured to become tristated when the receiver is disabled (section 4.7.30, page 34). Signal state changes are not synchronized with the Rx Clock.

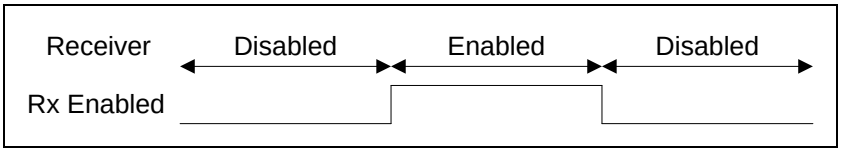


**Figure 14** The Rx Enabled signal reflects the receiver enable state.

The Rx Enabled signal is the default functionality of the cable pin identified in the board user manual as Cable Command 6 (CC6). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 5. When the pin is not used it is recommended that it be configured as a GPIO input.

### 8.6.2.4. Frame Valid

The Frame Valid input signal reflects the availability of data at the cable interface for capture by the receiver. As an input from the cable interface, the receiver responds to the signal only when the receiver is enabled. When the receiver is disabled, the receiver ignores the signal. This signal is not required for Flow Control of continuous, unstructured data streams. The signal is driven high by the transmitting device to indicate that valid data is present to be clocked in at the cable interface and is driven low otherwise. See Figure 15. The signal is synchronized with the Rx Clock such that state changes are clocked in on the clock's falling edge. It is expected that the transmitting device clocks Frame Valid changes out on the clock's rising edge.

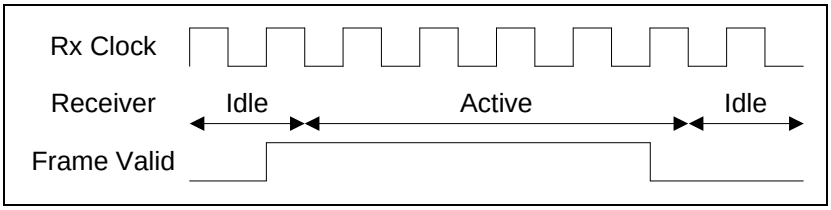


**Figure 15** The Frame Valid signal reflects the data reception process.

The Frame Valid signal is the default functionality of the cable pin identified in the board user manual as Cable Command 0 (CC0). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 6. When the pin is not used it is recommended that it be configured as a GPIO input.

> **NOTE**: Technically, it is possible to receive data without using the Frame Valid signal. In this case reception is governed by whatever other flow control signals are configured for use. If none are, then data is received at the rate of the Rx Clock signal. With no flow control signaling in use, this configuration can only control data reception by gating the Rx Clock. That is, the Rx Clock remains idle when no data is present at the cable interface, and it is active when valid data is present. Using an ungated Rx Clock is a means to performs tests at the board's maximum reception rate.

### 8.6.2.5. Line Valid

The Line Valid input signal reflects the availability of line data at the cable interface for capture by the receiver. As an input from the cable interface, the receiver responds to the signal only when the receiver is enabled. When the receiver is disabled, the receiver ignores the signal. This signal is not required for Flow Control of a continuous unstructured data streams. The signal is driven high by the transmitting device to indicate that valid data is present to be clocked in at the cable interface and is driven low otherwise. See Figure 16. The signal is synchronized with the Rx Clock such that state changes are clocked in on the clock's falling edge. It is expected that the transmitting device clocks Line Valid changes out on the clock's rising edge.

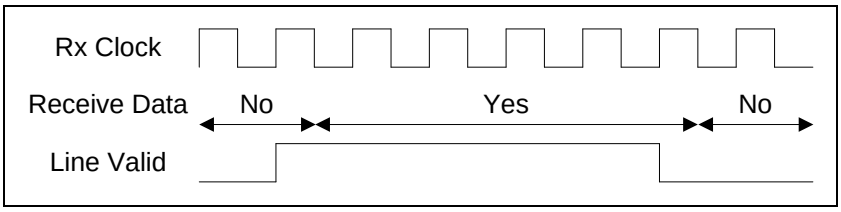


**Figure 16** The Line Valid signal reflects valid transmit data being presented at the cable interface.

The Line Valid signal is the default functionality of the cable pin identified in the board user manual as Cable Command 1 (CC1). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 0. When the pin is not used it is recommended that it be configured as a GPIO input.

### 8.6.2.6. Status Valid

The Status Valid input signal reflects the availability of status data at the cable interface for capture by the receiver. As an input from the cable interface, the receiver responds to the signal only when the receiver is enabled. When the receiver is disabled, the receiver ignores the signal. This signal is not required for Flow Control of a continuous unstructured data streams. The signal is driven high by the transmitting device to indicate that status data is present to be clocked in at the cable interface and is driven low otherwise. See Figure 17. The signal is synchronized with the Rx Clock such that state changes are clocked in on the clock's falling edge. It is expected that the transmitting device clocks Status Valid changes out on the clock's rising edge.

> **NOTE**: Status Data is data clocked in while the Status Valid signal is asserted. Refer to the board user manual for additional information.

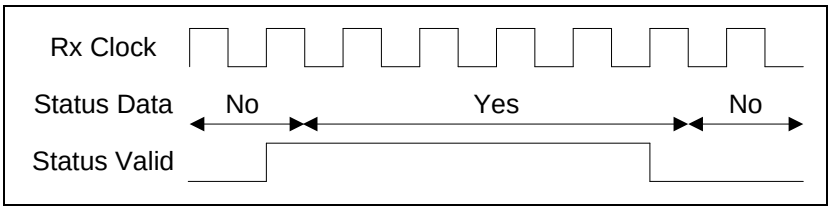


**Figure 17** The Status Valid signal reflects valid status data being presented at the cable interface.

The Status Valid signal is the default functionality of the cable pin identified in the board user manual as Cable Command 2 (CC2). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 1. When the pin is not used it is recommended that it be configured as a GPIO input.

### 8.6.2.7. Rx Ready

The Rx Ready output signal may be used by the receiver to pause the flow of data from the remote transmitting device. The signal is driven on the cable interface only when the receiver is enabled. When the receiver is disabled, the signal is not driven by the HPDI32. This signal is not required for Flow Control of continuous, unstructured data

streams. The Rx Ready signal is driven high by the receiver to permit data flow and is driven low to pause data flow. See Figure 18. The signal reflects the Rx FIFO Almost Full Status. The signal is not synchronized with Rx Clock and changes state as the Rx FIFO fill level changes. Data flow neither pauses nor resumes instantaneously. It may take several Rx Clock cycles for the remote transmit device to pause or resume.

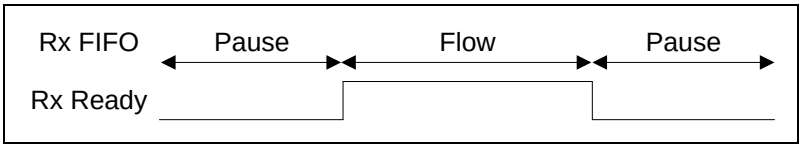


**Figure 18** The receiver drives the Tx Ready signal to control data flow.

The Rx Ready signal is the default functionality of the cable pin identified in the board user manual as Cable Command 3 (CC3). This default behavior can be negated by configuring the pin as a general purpose I/O signal (see section 4.7.1, page 21). In this configuration the signal is identified as GPIO 2. When the pin is not used it is recommended that it be configured as a GPIO input.

### 8.6.3. Receiver Operation Settings

The following is a general description of the receiver operation settings.

| Setting | Description |
|---|---|
| Rx Auto Start | If this setting is enabled, then the `hpdi32_read()` service enables the receiver before initiating data reception. This setting should usually be enabled. (See section 4.7.12, page 28.) |
| Rx Enable | This is the master setting that enables or disables the receiver. (See section 4.7.13, page 28.) |
| Rx FIFO Almost Empty | During data transfer, the receiver slows or pauses DMDMA data movement from the Rx FIFO if its fill level falls to this setting. This is done to prevent the receiver pipeline from reading from an empty Rx FIFO. Applications can adjust this threshold and can use the state as an interrupt source. The level should never be reduced below the default. (See section 4.7.14, page 28.) |
| Rx FIFO Almost Full | During data transfer, the receiver attempts to slow or pause the remote transmitter when the Rx FIFO fill level rises to this fill level. The attempt is made via the Rx Ready cable signal, which is driven via the Rx FIFO Almost Full status. This is the remote transmitter's Remote Throttle input. This is done to prevent the receiver pipeline from writing to a full Rx FIFO. Applications can adjust this threshold and can use the state as an interrupt source. The level should never be reduced below the default. (See section 4.7.15, page 29.) |
| Rx FIFO Overrun | This setting allows applications to query for, and clear, an Rx FIFO Overrun. (See section 4.7.16, page 29.) |
| Rx FIFO Underrun | This setting allows applications to query for, and clear, an Rx FIFO Underrun. During normal operation the driver prevents underruns. (See section 4.7.19, page 30.) |

### 8.6.4. Receiver I/O Settings

The following is a general description of the receiver I/O settings, which are the settings affecting the operation of the `hpdi32_read()` service.

| Setting | Description |
|---|---|
| Rx I/O Data Size | This setting specifies the size of each data item transferred over the cable interface. The options are 8-bits, 16-bits and 32-bits. This setting affects only the transfer of data from the device to the host as the Rx FIFO does not perform data packing and the cable interface always receives all 32 data bits. Data alignment is such that data bit D0 always corresponds to cable data signal D0. (See section 4.7.21, page 31.) |

| | |
|---|---|
| Rx I/O Mode | This service selects the method the driver uses to transfer data from the device to the host. The options are PIO, Block Mode DMA and Demand Mode DMA. For mode descriptions refer to section 8.1.3, page 51. (See section 4.7.23, page 32.) |
| Rx I/O Overrun | This setting configures the `hpdi32_read()` service's response to Rx FIFO overruns. If the *check* option is selected, the default, then the service checks for an Rx FIFO overrun before the service starts transferring data from the device. If the *ignore* option is selected, then the check is not performed. During normal operation the driver prevents overruns. (See section 4.7.24, page 32.) |
| Rx I/O DMA Threshold | This setting limits the size of the smallest individual DMA transfers. This is done to help improve efficiency. The BMDMA default exceeds the equivalent PIO Threshold default to limit the use of PIO to those instances where it is required to complete I/O requests. (See section 4.7.21, page 31.) |
| Rx I/O PIO Threshold | For very small data transfers the driver automatically changes DMA requests to PIO. This is done because PIO is more efficient for very small read requests. This service configures the threshold at which that change takes place. The default is 32 data values. (See section 4.7.25, page 33.) |
| Rx I/O Timeout | This setting specifies the maximum duration of each `hpdi32_read()` request. (See section 4.7.26, page 33.) |
| Rx I/O Underrun | This setting configures the `hpdi32_read()` service's response to Rx FIFO underruns. If the *check* option is selected, the default, then the service checks for an Rx FIFO underrun before the service starts transferring data from the device. If the *ignore* option is selected, then the check is not performed. During normal operation the driver prevents underruns. (See section 4.7.27, page 33.) |
| Tx/Rx Enable Tristate | This setting controls the operation of the Tx Enabled and Rx Enabled cable signals when either the transmitter or the receiver is enabled, respectively. With the *No* setting, each signal is driven when enabled. With the *Yes* settings each signal is tristated rather than being driven. (See section 4.7.30, page 34.) |

# 9. Sample Applications

For information on the sample applications refer to this same section number in the OS specific HPDI32 driver user manual.

# Document History

| Revision | Description |
|---|---|
| July 12, 2023 | Updated to version 3.12.104.x.x. Numerous, minor editorial changes. Updated the Rx and Tx Reset FIFO services. |
| December 20, 2022 | Updated to version 3.11.101.x.x. Minor editorial changes. Corrected the description of the Clock Divider feature. Added details on the operation of the transmitter and receiver cable signals. Expanded Tx and Rx signal information in section 8. Added Tx and Rx DMA Threshold information to section 8. |
| October 5, 2022 | Updated to version 3.10.101.x.x. Updated the information for the open and close calls. Discontinued the SDK. Minor editorial updates. |
| March 25, 2021 | Updated to version 3.9.93.x.x. Various minor editorial changes. Updated the debugging aids content. |
| November 3, 2020 | Updated to version 3.9.91.x.x. Various minor editorial changes. Added support for HPDI32B model boards. |
| October 23, 2020 | Version 3.8.91.x.x. Added the `HPDI32_IOCTL_GPIO_D32_OUTPUT` IOCTL service. |
| October 1, 2020 | Version 3.7.91.x.x. Minor editorial changes. Added error code information. Removed some SDK information. |
| April 22, 2020 | Version 3.7.91.x.x. Minor editorial changes. |
| March 1, 2020 | Initial release. Version 3.7.91.x.0. |