DMI32

Deep Memory 32-bit Digital I/O

PCI-DMI32

Software Development Kit
SDK 6.0.0 Reference Manual

Manual Revision: October 29, 2007

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

Preface

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

General Standards Corporation
8302A Whitesburg Dr.
Huntsville, Alabama 35802
Phone: (256) 880-8787
FAX: (256) 880-8788
URL: http://www.generalstandards.com
E-mail: sales@generalstandards.com

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

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

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

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

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

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

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

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

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

Table of Contents

1. Introduction 8
1.1. Purpose 8
1.2. Acronyms 8
1.3. Definitions 8
1.4. Installation 8
1.5. Application Programming Interface 9
1.6. Software Overview 9
1.6.1. Software Architecture 9
1.7. Hardware Overview 10
1.8. Code Samples 10
1.9. Reference Material 10
2. Using the SDK 12
2.1. Compatibility 12
2.2. Limitations and Restrictions 12
2.3. Multithreaded 12
2.4. Compile Time Use 12
2.5. Link Time Use 12
2.6. Run Time Use 12
3. Macros 13
3.1. API Version Number 13
3.2. Common Parameter Assignment Values 13
3.3. Discrete Data Type Options 14
3.4. I/O Status Fields 15
3.5. Maximum Number of Open Handles 16
3.6. Parameter Access “Which” Bits 16
3.7. Registers 17
3.7.1. GSC Registers 17
3.7.2. PCI9080 PCI Configuration Registers 17
3.7.3. PLX PCI9080 Feature Set Registers 18
3.8. Version Data Selectors 20
4. Data Types 21
4.1. Discrete Data Types 21
4.2. dmi32_callback_func_t 21
4.3. Status Values 21

5. Functions 23
5.1. dmi32_api_status() 23
5.2. dmi32_board_count() 24
5.3. dmi32_close() 25
5.4. dmi32_config() 25
5.5. dmi32_init() 27
5.6. dmi32_io_wait() 28
5.7. dmi32_irq_wait() 29
5.8. dmi32_open() 30
5.9. dmi32_read() 31
5.10. dmi32_reg_mod() 32
5.11. dmi32_reg_read() 33
5.12. dmi32_reg_write() 34
5.13. dmi32_reset() 35
5.14. dmi32_seek() 36
5.15. dmi32_status_text() 37
5.16. dmi32_tell() 38
5.17. dmi32_version_get() 39
5.18. dmi32_write() 40
5.19. dmi32_xfer_buf_get() 42
5.20. dmi32_xfer_buf_set() 43

6. Configuration Parameters 45
6.1. Parameter Macros 45
  6.1.1. Parameter Definitions 45
  6.1.2. Value Definitions 45
  6.1.3. Service Definitions 45
    6.1.3.1. Device Handle: h 45
    6.1.3.2. Which Bits: w 46
    6.1.3.3. Set Value: s 46
    6.1.3.4. Get Value: g 46
6.2. Input/Output Parameters 46
  6.2.1. I/O Parameter: Abort 46
  6.2.2. I/O Parameter: Aborted 47
  6.2.3. I/O Parameter: Buffer Pointer 47
  6.2.4. I/O Parameter: Buffer Size 48
  6.2.5. I/O Parameter: Callback Argument 48
  6.2.6. I/O Parameter: Callback Function 49
  6.2.7. I/O Parameter: Mode 49
  6.2.8. I/O Parameter: Offset 50
  6.2.9. I/O Parameter: Overlap Enable 50
  6.2.10. I/O Parameter: Status 51
  6.2.11. I/O Parameter: Timeout 51

6.3. Interrupt Parameters....52
6.3.1. Interrupt Parameter: Callback Argument....52
6.3.2. Interrupt Parameter: Callback Function....53
6.3.3. Interrupt Parameter: Enable....53
6.3.4. Interrupt Parameter: State....54

6.4. Miscellaneous Parameters....54
6.4.1. Miscellaneous Parameter: Board Jumpers....55
6.4.2. Miscellaneous Parameter: DRAM Capacity MB....55
6.4.3. Miscellaneous Parameter: DRAM Size MB....55
6.4.4. Miscellaneous Parameter: Features....56
6.4.5. Miscellaneous Parameter: GSC Register Mapping....56
6.4.6. Miscellaneous Parameter: PLX Register Mapping....57
6.4.7. Miscellaneous Parameter: Strict Arguments....57
6.4.8. Miscellaneous Parameter: Strict Configuration....58
6.4.9. Miscellaneous Parameter: Strict DRAM Limits....58

6.5. Transfer Parameters....59
6.5.1. Transfer Parameter: CLKIN Test....59
6.5.2. Transfer Parameter: Direction....60
6.5.3. Transfer Parameter: Enable....60
6.5.4. Transfer Parameter: Internal Clock Speed....61
6.5.5. Transfer Parameter: Mode....61
6.5.6. Transfer Parameter: Offset....62
6.5.7. Transfer Parameter: Size....62
6.5.8. Transfer Parameter: Start Word....63
6.5.9. Transfer Parameter: Start Word Direction....63
6.5.10. Transfer Parameter: State....64
6.5.11. Transfer Parameter: State Rx....64
6.5.12. Transfer Parameter: State Tx....64
6.5.13. Transfer Parameter: Test Mode....64

6.6. User I/O Parameters....65
6.6.1. User I/O Parameter: Direction....65
6.6.2. User I/O Parameter: Input....66
6.6.3. User I/O Parameter: Output....66

7. Operation....68

7.1. Overview....68

7.2. Data Reception....68
7.2.1. Overview....68
7.2.2. Details....68

7.3. Data Transmission....69
7.3.1. Overview....69
7.3.2. Details....70

7.4. Data Transfer Issues....71
7.4.1. I/O Abort Requests....71
7.4.2. I/O Data Buffers....71
7.4.3. I/O Timeout....71
7.4.4. Blocking/Overlapped I/O....72
7.4.5. I/O Modes....72
7.4.6. DMA Based I/O Requests....72

7.5. Event Notification....72
7.5.1. Event Callback....73

7.5.1.1. Interrupt Notification Callback 73
7.5.1.2. I/O Completion Notification Callback 73
7.5.2. Event Waiting 73

7.6. Clock Checks 73

Document History 75

1. Introduction

This reference manual applies to release version 6.0.0 of the DMI32 SDK.

1.1. Purpose

The purpose of this document is to describe the Application Programming Interface to the DMI32 Software Development Kit. This software provides the interface between "Application Software" and the DMI32 board. The interface provided by the SDK is based on the board's functionality.

1.2. Acronyms

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

Acronyms	Description
API	Application Programming Interface (This is sometimes used synonymously with SDK or API Library.)
DMA	Direct Memory Access
GSC	General Standards Corporation
PCI	Peripheral Component Interconnect
PIO	Programmed I/O
SDK	Software Development Kit (This is sometimes used synonymously with API or API Library.)

1.3. Definitions

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

Term	Definition
API Buffer	A physically contiguous block of memory allocated via the API.
API Library	This refers to the library implementing the application level DMI32 interface. (This is sometimes used synonymously with SDK or API.)
Application	This refers to user mode processes.
Application Buffers	These are memory buffers allocated and maintained entirely by the application which are used for reading data from and writing data to the DMI32's memory.
Device Driver	This refers to the driver executable component of the DMI32 SDK.
DRAM	This is a general reference to the DMI32's onboard memory.
Driver	This refers to the device driver, which runs under control of the operating system.
Rx	For I/O operations this refers to reading data from the DMI32's onboard memory. Otherwise it refers to the reception of data over the cable interface, either into memory or the User I/O ports.
Transfer Buffer	This is what defines a DMI32 memory region to be involved in a cable based data transfer. The buffer is defined by a zero based offset and a size. Both are defined in bytes and are 32-bit aligned.
Tx	For I/O operations this refers to writing data to the DMI32's onboard memory. Otherwise it refers to the transmission of data over the cable interface, either from memory or the User I/O ports.

1.4. Installation

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

1.5. Application Programming Interface

The SDK API is defined in the three header files listed below. These C language headers are C++ compatible. The only header that need be included by DMI32 applications is `dmi32_api.h`. The API consists of macros, data types, function calls and parameter definitions. These are described in other sections of this document. The headers define numerous items in addition to those described in this document. These additional items are provided without documentation. All software components of the API begin with a prefix of `DMI32` or `GSC` (both appear with upper and lower case letters). The table below indicates where to look for any particular item's definition.

File Name	Description
`dmi32_api.h`	This header contains the bulk of the API, including function calls, data types and numerous macros. All items defined here include the prefix "`DMI32`" or "`dmi32`".
`gsc_common.h`	This header contains status definitions, a few data type definitions and a variety of macros. All items defined here have a prefix of "`GSC`" or "`gsc`".
`gsc_pci9080.h`	This header contains register definitions for the PCI9080, which is the PCI interface chip used on DMI32s. All items defined here have a prefix of "`GSC`" or "`gsc`" and include "`9080`".

1.6. Software Overview

The software interface to the DMI32 consists of a Device Driver and an API Library; the primary components of the SDK. The Device Driver operates under control of the operating system and must be loaded and running in order to access any installed DMI32 devices. The interface provided by the API Library is based on the board's functionality and is organized around the DMI32's set of main hardware features. The general categories are as follows and permit access to and manipulation of virtually every feature available on the board.

- General Access Services (API Status, Version Numbers, Board Count, Open, Close, …)
- Data I/O Configuration
- Data Transfer Configuration
- GPIO Configuration
- Other Miscellaneous Configuration
- Register read and write operations

All DMI32 features are individually accessible via a generalized configuration service. For each parameter, as appropriate, the API includes a set of support macros. These include setting options (i.e. defaults and acceptable values), quick access retrieval macros, and quick access manipulation macros. All are described later in this document.

1.6.1. Software Architecture

An application communicates with a DMI32 using the driver and library described briefly above. Any number of applications may make simultaneous use of the library and each use is totally independent, unless specifically designed to do otherwise. Each instance provides access to at most 32 different DMI32 devices. The diagram below describes the components and how they fit together.

Application	This is any application written to communicate with one or more DMI32 devices using the driver and library provided in the SDK.
API Library	This library presents a DMI32 feature based interface to applications wishing to communicate with DMI32 devices.
Device Driver	The driver provides access to DMI32 devices.
DMI32	This refers to any number of installed DMI32 devices.

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

1.7. Hardware Overview

The DMI32 is a high-capacity memory board with a 32-bit parallel digital I/O interface. The host side connection is 32-bit PCI based. 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 the external I/O interface, depending on the model ordered. Onboard memory of up to 4GB (or 2GB, depending on the model) permits large volumes of data to be transmitted from or recorded by the board. The DMI32 can transmit or receive single blocks up to the size of its memory. Alternatively, it can transmit or receive these blocks continuously without host intervention.

The DMI32 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 User I/O signals that control initiation of data transfer. They may also be used as limited general purpose I/O. The board accommodates a wide range of applications. This range extends from sending or receiving relatively small blocks of data on demand, to sending or receiving large continuous streams of data indefinitely.

1.8. Code Samples

All of the code samples in this manual are included in the `dmi32_dsl` library along with their C source files. The examples given are notably simplistic, but are provided to illustrate use rather than accomplishment of broader tasks.

1.9. Reference Material

The following reference material may be of particular benefit in using the DMI32 and this SDK. The specifications provide the information necessary for an in-depth understanding of the specialized features implemented on this board.

- The applicable *DMI32 Setup Guide* from General Standards Corporation. These are OS specific.
- The applicable *DMI32 User Manual* from General Standards Corporation.
- The *PCI9080 PCI Bus Master Interface Chip* data handbook from PLX Technology, Inc.

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

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

2. Using the SDK

2.1. Compatibility

The API Library interface presented in this version of the SDK is identical for identical for those operating systems for which this version of the SDK has been ported. Compatibility is based on the individual numbers in the overall version number. The first number changes when there are dramatic changes in the interface that will necessitate application porting. The second number changes when there are minor changes to the interface, including additional services. Applications should be able to use the updated driver as is. If not, the application may have to be recompiled. Changes in the third number refer to changes in the release that have to do with the support file or other issues.

2.2. Limitations and Restrictions

The SDK release for each supported operating systems may place limitations or restrictions on that implementation of the SDK. Refer to the Setup Guide for the operation system in question.

2.3. Multithreaded

The API Library is multithreaded. This may require that applications and libraries using the API also be built for multithreaded operation. Difficult to identify bugs may appear when the API Library is used with applications built for single threaded only operation.

2.4. Compile Time Use

Using the SDK places compile time requirements on DMI32 application build procedures. The requirements are generally the same for each operating system specific release, though the details vary from one OS to another and among different tool sets. Refer to the documentation for the build tools you are using. Compile time use has three requirements. First, include the header file `dmi32_api.h` in each C module referencing an API component. Second, configure the compiler for multithreaded operation, as applicable. Third, expand the include file search path to search the directories where the library header and its included files are located. Refer the SDK Setup Guide for your operating system for additional information.

2.5. Link Time Use

Using the SDK places link time requirements on DMI32 application build procedures. The requirements are generally the same for each operating system specific release, though the details vary from one OS to another and among different tool sets. Refer to the documentation for the build tools you are using. Link time use has two requirements. The first requirement is to insure that the API Library has been installed. This permits the library to be automatically located by the linker. The second argument is to link the applications with both the API Library. Refer the SDK Setup Guide for your operating system for additional information.

2.6. Run Time Use

Using the SDK places run time requirements on DMI32 applications. The requirements are generally the same for each operating system specific release, though the details vary from one OS to another and the corresponding DSK release. There are two run time requirements. The first is to insure that the driver has been built and loaded. The second requirement is to insure that the API Library has been installed. Refer the SDK Setup Guide for your operating system for additional information.

3. Macros

The DMI32 API includes the following macros. The headers also contain various other utility type macros, which are provided without documentation. Parameter support macros are not presented in this subsection. These macros are described in section 6 beginning on page 45.

3.1. API Version Number

This macro defines the version number of the API's executable interface. It does not refer to the SDK version number, the API Library version number or the Device Driver version number. Applications pass this value to the function `dmi32_api_status()` (page 23), which is used to verify that the application and the library are compatible.

Macros	Description
`DMI32_API_VERSION`	This is the API's executable interface version number.

3.2. Common Parameter Assignment Values

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

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_tx_timeout_reset(void* handle, int verbose)
{
    U32 status;

    // Reset the Tx I/O timeout period to its default.
    status  = dmi32_config( handle,
                            DMI32_IO_TIMEOUT,
                            DMI32_WHICH_TX,
                            GSC_DEFAULT,
                            NULL);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_config() failure: %ld\n", (long) status);
    }
    else
    {
        printf("I/O Tx Timeout Reset\n");
    }

    return(status);
}
```

Example

```
    #include <stdio.h>

    #include "dmi32_api.h"
    #include "dmi32_dsl.h"

    U32 dmi32_dsl_tx_timeout_get(
        void*   handle,
        U32*    timeout_secs,
        int     verbose)
    {
        U32             status;
        unsigned long   value;

        // Retrieve the Tx I/O timeout period without changing it.
        status  = dmi32_config( handle,
                                DMI32_IO_TIMEOUT,
                                DMI32_WHICH_TX,
                                GSC_NO_CHANGE,
                                &value);
        timeout_secs[0] = (U32) value;

        if (!verbose)
        {
        }
        else if (status != GSC_SUCCESS)
        {
            printf("dmi32_config() failure: %ld\n", (long) status);
        }
        else
        {
            printf( "I/O Tx Timeout Set: %ld seconds\n",
                    (long) timeout_secs);
        }

        return(status);
    }
```

3.3. Discrete Data Type Options

The below macros are defined by application code as needed to enable or disable declarations for and size validation for the data types `S8`, `U8`, `S16`, `U16`, `S32` and `U32` (see page 21).

Macros	**Description**
`GSC_DATA_TYPES_CHECK`	If the API declares the data types and the application defines this macro, then the data type sizes will be validated during the application's build process. This macro should only be defined if the compiler in use supports the `sizeof()` macro during preprocessing.
`GSC_DATA_TYPES_NOT_NEEDED`	Applications should define this macro before including `dmi32_api.h`

	to disable the declarations for these data types.

3.4. I/O Status Fields

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

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

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

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

Example

```
#include "dmi32_api.h"
#include "dmi32_dsl.h"

long dmi32_dsl_io_status_evaluate(U32 io_status)
{
    long    bytes;
    U32     status  = io_status & GSC_IO_STATUS_MASK;

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

```
        else if (status == GSC_IO_STATUS_SUCCESS)
        {
            // The operation has completed.
            bytes   = (long) (io_status & GSC_IO_STATUS_COUNT_MASK);
        }
        else if (status == GSC_IO_STATUS_TIMEOUT)
        {
            // The timeout period lapsed.
            // The count may not be accurate.
            bytes   = -1;
        }
        else if (status == GSC_IO_STATUS_ERROR)
        {
            // There was an error.
            // The count may not be accurate.
            bytes   = -1;
        }
        else if (status == GSC_IO_STATUS_ABORTED)
        {
            // The operation was aborted.
            // The count may not be accurate.
            bytes   = -1;
        }
        else
        {
            // Unknown status.
            bytes   = -1;
        }

        return(bytes);
    }
```

3.5. Maximum Number of Open Handles

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

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

3.6. Parameter Access “Which” Bits

The table below lists the set of selection bits that may be set when a configuration parameter is modified or accessed. They are referred to as “which” bits in that they specify the object or objects which the parameter is to access. When appropriate, bits within the same category may be bitwise or’d in order to apply the action to multiple objects.

Macros	Description
`DMI32_WHICH_IRQ_ALL`	This refers to all of the interrupts.
`DMI32_WHICH_IRQ_BE`	This specifies the Buffer End interrupt. *
`DMI32_WHICH_IRQ_BS`	This specifies the Buffer Start interrupt. *
`DMI32_WHICH_IRQ_MBUR`	This specifies the Multi-Buffer Under Run interrupt. *
`DMI32_WHICH_IRQ_UIOA`	This specifies the User I/O A Input interrupt. *
`DMI32_WHICH_IRQ_UIOB`	This specifies the User I/O B Input interrupt. *
`DMI32_WHICH_RX`	This refers to I/O based read parameters. *

`DMI32_WHICH_TX`	This refers to I/O based write parameters. *
`DMI32_WHICH_USER_IO_A`	This refers to the User I/O A port. *
`DMI32_WHICH_USER_IO_B`	This refers to the User I/O B port. *
`DMI32_WHICH_USER_IO_CURRENT`	This refers to the User I/O port currently configured as an input or output, as applicable to the respective parameters.

* Other macros are also defined that include various logical combinations of some bits.

3.7. Registers

The following tables give the complete set of DMI32 registers. The tables are divided by register categories. There are PCI registers, PLX feature set registers, and there are GSC firmware based registers. The PCI registers and the PLX registers are provided by the PCI interface chips used on the DMI32. Applications have read access to all registers, but write access only to the GSC firmware registers.

3.7.1. GSC Registers

The following table gives the complete set of GSC specific DMI32 registers. For detailed definitions of these registers refer to the applicable *DMI32 User Manual*.

Macros	Description
`DMI32_BCR`	Board Control Register (BCR)
`DMI32_BSR`	Board Status Register (BSR)
`DMI32_FR`	Features Register (FR)
`DMI32_FRR`	Firmware Revision Register (FRR)
`DMI32_ICR`	Interrupt Control Register (ICR)
`DMI32_ISR`	Interrupt Status Register (ISR)
`DMI32_LMBR`	Local Memory Bank Register (LMBR)
`DMI32_TBBAR`	Transfer Buffer Base Address Register (TBBAR)
`DMI32_TBSR`	Transfer Buffer Size Register (TBSR)
`DMI32_TSWR`	Transfer Start Word Register (TSWR)

3.7.2. PCI9080 PCI Configuration Registers

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

Macros	Description
`GSC_PCI_9080_BAR0`	PCI Base Address Register for Memory Accesses to Local, Runtime, and DMA Registers (PCIBAR0)
`GSC_PCI_9080_BAR1`	PCI Base Address Register for I/O Accesses to Local, Runtime, and DMA Registers (PCIBAR1)
`GSC_PCI_9080_BAR2`	PCI Base Address Register for Memory Accesses to Local Address Space 0 (PCIBAR2)
`GSC_PCI_9080_BAR3`	PCI Base Address Register for Memory Accesses to Local Address Space 1 (PCIBAR3)
`GSC_PCI_9080_BAR4`	Unused Base Address Register (PCIBAR4)
`GSC_PCI_9080_BAR5`	Unused Base Address Register (PCIBAR5)
`GSC_PCI_9080_BISTR`	PCI Built-In Self Test Register (PCIBISTR)
`GSC_PCI_9080_CCR`	PCI Class Code Register (PCICCR)
`GSC_PCI_9080_CIS`	PCI Cardbus CIS Pointer Register (PCICIS)
`GSC_PCI_9080_CLSR`	PCI Cache Line Size Register (PCICLSR)
`GSC_PCI_9080_CR`	PCI Command Register (PCICR)
`GSC_PCI_9080_DIDR`	PCI Device ID Register (PCIDIDR)

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

NOTE: The following table gives register identification information for DMI32 boards.

Register	Value	Description
`GSC_PCI_9080_VIDR`	`0x10B5`	The PCI interface chip is a PLX device.
`GSC_PCI_9080_DIDR`	`0x9080`	The PCI interface chip is a PLX PCI9080.
`GSC_PCI_9080_SVID`	`0x10B5`	The below register value has been assigned by PLX.
`GSC_PCI_9080_SID`	`0x2707`	The device is a DMI32 family product.

3.7.3. PLX PCI9080 Feature Set Registers

The following tables give the set of PLX feature set registers.

Local Configuration Registers

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

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

Runtime Registers

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

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

DMA Registers

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

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

Message Queue Registers

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

Macros	Description
`GSC_PLX_9080_IFHPR`	Inbound Free Head Pointer Register (IFHPR)
`GSC_PLX_9080_IFTPR`	Inbound Free Tail Pointer Register (IFTPR)
`GSC_PLX_9080_IPHPR`	Inbound Post Head Pointer Register (IPHPR)
`GSC_PLX_9080_IPTPR`	Inbound Post Tail Pointer Register (IPTPR)
`GSC_PLX_9080_IQP`	Inbound Queue Port Register (IQP)
`GSC_PLX_9080_MQCR`	Messaging Queue Configuration Register (MQCR)
`GSC_PLX_9080_OFHPR`	Outbound Free Head Pointer Register (OFHPR)
`GSC_PLX_9080_OFTPR`	Outbound Free Tail Pointer Register (OFTPR)
`GSC_PLX_9080_OPHPR`	Outbound Post Head Pointer Register (OPHPR)
`GSC_PLX_9080_OPLFIM`	Outbound Post List FIFO Interrupt Mask Register (OPLFIM)

`GSC_PLX_9080_OPLFIS`	Outbound Post List FIFO Interrupt Status Register (OPLFIS)
`GSC_PLX_9080_OPTPR`	Outbound Post Tail Pointer Register (OPTPR)
`GSC_PLX_9080_OQP`	Outbound Queue Port Register (OQP)
`GSC_PLX_9080_QBAR`	Queue Base Address Register (QBAR)
`GSC_PLX_9080_QSR`	Queue Status/Control Register (QSR)

3.8. Version Data Selectors

This set of macros is used when requesting a version number and indicates which version number is desired. The macros are passed as the `id` argument to the `dmi32_version_get()` function (see page 39). The second table below lists utility macros used to retrieve each of the respective version numbers. In the second table, the argument `h` refers to the handle used to access the device, the `b` refers to an application buffer where the version string is recorded, and the `s` is the size of that buffer.

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

Macro (Services)	**Description**
`DMI32_VERSION_GET_LIBRARY(h,b,s)`	This requests the version number for the API Library.
`DMI32_VERSION_GET_DRIVER(h,b,s)`	This requests the version number for the Device Driver.

4. Data Types

The interface includes the following data types.

4.1. Discrete Data Types

The following discrete data types are defined and used by the API. If a DMI32 application includes other headers which also define these types, then the API can be directed to omit these definitions. This is done by defining the macro `GSC_DATA_TYPES_NOT_NEEDED` before including the API header. The alternate definitions must however define these types as listed in the below table.

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

4.2. dmi32_callback_func_t

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

Definition

```
typedef void (*dmi32_callback_func_t)(U32 arg1, U32 arg2, U32 arg3);
```

Arguments	Description
`arg1`	This is the device handle cast to a `U32` data type.
`arg2`	For Interrupt Notification this is the `DMI32_WHICH_XXX` bit for the respective interrupt. For I/O Completion Notification this is the applicable `GSC_IO_STATUS_XXX` status components.
`arg3`	This is any arbitrary application supplied data value.

4.3. Status Values

This unnamed enumerated data type lists all possible status values returnable from API service calls. The enumerated values represent common definitions used by this and other GSC SDKs. Many values will never be encountered when using the DMI32 SDK. The table below gives brief descriptions for many values and omits those that should never be seen. The most common value encountered is `GSC_SUCCESS` and indicates that the request was completed successfully.

Definition

```
typedef enum
{
    …
};
```

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

`GSC_ACCESS_DENIED`	The operation failed because access to a device, service or system resource or service was denied.
`GSC_FAILED`	An operation failed in a non-specific manner.
`GSC_INIT_FAILURE`	API Library initialization failed.
`GSC_INSUFFICIENT_RESOURCES`	An operation failed because insufficient OS resources were available.
`GSC_INVALID_API_HANDLE`	An operation failed because the application supplied an invalid device handle. API device handles are API specific resources and are of no meaning to the OS.
`GSC_INVALID_DATA`	An operation failed because invalid data was provided.
`GSC_INVALID_VERSION_API`	API Library initialization failed because the API Library version was incompatible. This refers either to the API's version number or the GSC revision level. The version data can still be retrieved when this status is seen.
`GSC_INVALID_VERSION_DRIVER`	API Library initialization failed because the Device Driver version was incompatible. This refers either to the driver's version number or the GSC revision level. The version data can still be retrieved when this status is seen.
`GSC_NULL_PARAM`	An operation failed because an argument was NULL.
`GSC_SUCCESS`	An operation completed successfully.
`GSC_THREAD_FAILURE`	An operation (`dmi32_open()`) failed because a support thread could not be started.
`GSC_TOO_MANY_OPEN_HANDLES`	An operation (`dmi32_open()`) failed because the application attempted too many simultaneous device accesses.
`GSC_UNSUPPORTED_FUNCTION`	An operation failed because the application requested a service that is unsupported or unimplemented.
`GSC_WAIT_TIMEOUT`	An operation completed because a timeout period lapsed.
`GSC_WAIT_CANCELED`	An operation waiting for an event ended prematurely. This usually means the application was terminated while waiting for the event.

5. Functions

The DMI32 API includes the following functions. The SDK interface also includes a number of function style macro definitions. These macros are described in section 6 beginning on page 45.

5.1. dmi32_api_status()

This function is the entry point to determine the status of the API Library. This must be the very first call into the API and determines the usability of the API Library and the Device Driver. If the initial status obtained is other than `GSC_SUCCESS`, then only a limited portion of the API is functional. If not fully usable, then both values returned may be useful in resolving the situation. Thereafter, the status obtained might vary if the API encounters irregular circumstances. The below table lists macros associated with this service.

Macro (Services)	Description
`DMI32_API_STATUS(s,a)`	Retrieve the status information without having to explicitly enter the macro for the current API version.

Prototype

```
U32 dmi32_api_status(U32* stat, U32* arg, U32 api_ver);
```

Argument	Description
`stat`	The API records the current API status here, which can change during use. The pointer must not be NULL. If an error status is returned, then the API must not be used.
`arg`	The API records auxiliary status information here, which can change during use. This value should be related to the above reported status. The pointer must not be NULL.
`api_ver`	This must be the version number of the API the application was written for. The API will evaluate this argument with respect to the library's own version to determine the library's suitability for the applications. If the API differs in any way that would cause problems, then the `stat` argument will reflect an error condition.

Return Value	Description
`GSC_SUCCESS`	The operation succeeded (the status was retrieved).
Otherwise …	A `GSC_XXX` error status reflecting the problem encountered. The API should not be used in this case.

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_api_status(int verbose)
{
    U32 arg;
    U32 stat;
    U32 status;

    status  = dmi32_api_status(&stat, &arg, DMI32_API_VERSION);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_api_status() failure: %ld\n", (long) status);
    }
    else
    {
        printf("API Status:\n");
        printf("  Status:   0x%lX\n", (long) stat);
        printf("  Argument: 0x%lX\n", (long) arg);
    }

    status  = (status == GSC_SUCCESS) ? stat : status;
    return(status);
}
```

5.2. dmi32_board_count()

This function is the entry point to determine the number of DMI32 boards installed in the system and accessible to the API. This service can be called without requiring access to any particular device.

Prototype

```
U32 dmi32_board_count(U8* count);
```

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

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_board_count(U8* count, int verbose)
{
    U32 status;

    status  = dmi32_board_count(count);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_board_count() failure: %ld\n", (long) status);
    }
    else
    {
        printf("DMI32 Board Count: %d\n", (int) count[0]);
    }
    return(status);
}
```

5.3. dmi32_close()

This function is the entry point to close a connection to an open DMI32 board. The function should only be called after a successful open of the respective device via `dmi32_open()` and must not be used after being closed. Before returning, the API returns the device to the same state produced when originally opened.

Prototype

```
U32 dmi32_close(void* handle);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_close(void* handle, int verbose)
{
    U32 status;

    status  = dmi32_close(handle);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_close() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Device Closed: 0x%lX\n", (long) handle);
    }

    return(status);
}
```

5.4. dmi32_config()

This function is the entry point to accessing an individual parameter where all pertinent data is given as separate arguments. This is the API's primary entry point to managing device parameters. The function should only be called after a successful open of the respective device via `dmi32_open()`.

Prototype

```
U32 dmi32_config(
    void*           handle,
    U32             parm,
    U32             which,
    unsigned long   set,
    unsigned long*  get);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`parm`	This specifies the parameter to be accessed.
`which`	This is any number or combination of parameter specific `DMI32_WHICH_XXX` bits that specify the object(s) the parameter is applied to. Many parameters ignore this argument. When it is used, a value of zero is acceptable, and merely specifies to access none of the corresponding objects.
`set`	This is the value to apply to the parameter being accessed. The universal value `GSC_NO_CHANGE` specifies that the parameter not be altered and must be used when the purpose of the access is merely to get the current setting. Some parameters are read-only, in which case this argument is ignored.
`get`	After applying any changes to the parameter, its current setting is recorded here. When the "`which`" argument specifies multiple objects, only the last accessed is recorded here. This argument may be NULL, in which case the current setting is not retrieved.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_tx_timeout_set(
    void*   handle,
    U32     timeout_s,
    int     verbose)
{
    unsigned long   get;
    U32             status;

    status  = dmi32_config( handle,
                            DMI32_IO_TIMEOUT,
                            DMI32_WHICH_TX,
                            timeout_s,
                            &get);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_config() failure: 0x%lX\n", (long) status);
    }
    else
    {
        printf("Tx (write) Timeout:\n");
        printf("  Set: 0x%lX\n", (long) timeout_s);
        printf("  Get: 0x%lX\n", get);
    }

    return(status);
}
```

5.5. dmi32_init()

This function is the entry point to return a device and all parameters to the state produced when the device was first opened. In doing this, any I/O operations in progress are aborted. This function should only be called after a successful open of the respective device via `dmi32_open()`.

Prototype

```
U32 dmi32_init(void* handle);
```

Argument	**Description**
`handle`	This is an API device handle obtained via `dmi32_open()`.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_init(void* handle, int verbose)
{
    U32 status;

    status  = dmi32_init(handle);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_init() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Device Initialized: 0x%lX\n", (long) handle);
    }

    return(status);
}
```

5.6. dmi32_io_wait()

This function is the entry point to pause thread execution until an I/O operation completes. The function should only be called after a successful open of the respective device via `dmi32_open()`. When called, the current thread will block until a referenced I/O operation completes. The waiting will occur if no I/O operations are currently active, or if an I/O operation is active in either blocking or overlapped mode. The call will return as soon as the time period expires, or when the first referenced operation completes, whether by an abort request, a failed I/O request, a timeout or successful data transfer, whichever occurs first. There is no limit to the number of threads that may simultaneously utilize this service or on the combination of operations that may be referenced.

Prototype

```
U32 dmi32_io_wait(void* handle, U32 which, U32 timeout_ms);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`which`	This is any bitwise or'd combination of `DMI32_WHICH_TX` or `DMI32_WHICH_RX`. Set `DMI32_WHICH_TX` to wait on a write operation. Set `DMI32_WHICH_RX` to wait on a read operation. If neither is set the function returns immediately with `GSC_SUCCESS`.
`timeout_ms`	This is the timeout limit is milliseconds. If an I/O operation does not complete within this time period, then the call returns at the end of the period. The timeout period will be at least the amount of time specified, but may be longer depending on the OS.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_tx_wait(void* handle, U32 timeout_ms, int verbose)
{
    U32 status;

    status  = dmi32_io_wait(handle, DMI32_WHICH_TX, timeout_ms);

    if (!verbose)
    {
    }
    else if (status == GSC_SUCCESS)
    {
        printf("Tx Wait: write operation completed.\n");
    }
    else if (status == GSC_WAIT_TIMEOUT)
    {
        printf( "Tx Wait: timeout after %ld milliseconds\n",
                (long) timeout_ms);
    }
    else
    {
        printf("dmi32_io_wait() failure: %ld\n", (long) status);
    }

    return(status);
}
```

5.7. dmi32_irq_wait()

This function is the entry point to pause thread execution until an interrupt occurs. The function should only be called after a successful open of the respective device via `dmi32_open()`. When called, the current thread will block until any one of a specified set of interrupts occurs. The call will return as soon as the time period expires, or when the first referenced interrupt occurs, whichever occurs first. There is no limit to the number of threads that may simultaneously utilize this service or on the combination of interrupts that may be referenced.

Prototype

```
U32 dmi32_irq_wait(void* handle, U32 which, U32 timeout_ms);
```

Argument	**Description**
`handle`	This is an API device handle obtained via `dmi32_open()`.
`which`	This is any bitwise or'd combination of `DMI32_WHICH_IRQ_XXX` bits. Set the bits according to the interrupt(s) of interest. Unreferenced interrupts will have no impact. If none are set the function returns immediately with `GSC_SUCCESS`.
`timeout_ms`	This is the timeout limit is milliseconds. If an interrupt does not occur within this time period, then the call returns at the end of the period. The timeout period will be at least the amount of time specified, but may be longer depending on the OS.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_multi_buffer_under_run_wait(
    void*   handle,
    U32     timeout_ms,
    int     verbose)
{
    U32 status;

    status  = dmi32_irq_wait(   handle,
                                DMI32_WHICH_IRQ_MBUR,
                                timeout_ms);
    if (!verbose)
    {
    }
    else if (status == GSC_SUCCESS)
    {
        printf("Under Run Wait: interrupt occurred.\n");
    }
    else if (status == GSC_WAIT_TIMEOUT)
    {
        printf( "Under Run Wait: "
                "timeout after %ld milliseconds\n",
                (long) timeout_ms);
    }
    else
    {
        printf("dmi32_irq_wait() failure: %ld\n", (long) status);
    }

    return(status);
}
```

5.8. dmi32_open()

This function is the entry point to open a connection to a DMI32 board. This function must be called before any other device access functions may be called. If successful, the device and all parameters are initialized to default settings. Multiple requests can be made to access the same device, and each can succeed. However, care must be taken when doing this as device access via one handle is likely to interfere with the device state maintained by the other. Additionally, one handle may configure the device in a way that conflicts with the configuration established by the other.

Prototype

```
U32 dmi32_open(U8 index, void** handle);
```

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

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_open(U8 index, void** handle, int verbose)
{
    U32 status;
```

```
        status  = dmi32_open(index, handle);

        if (!verbose)
        {
        }
        else if (status != GSC_SUCCESS)
        {
            printf("dmi32_open() failure: %ld\n", (long) status);
        }
        else
        {
            printf("Device Opened:\n");
            printf("  Index:  0x%lX\n", (long) index);
            printf("  Handle: 0x%lX\n", (long) handle);
        }

        return(status);
    }
```

5.9. dmi32_read()

This function is the entry point to reading DRAM data from a DMI32. The function should only be called after a successful open of the respective device via `dmi32_open()`. The operation will be carried out according to the current set of read side I/O Parameters. If the Overlap Enable option is disabled, then the function will block and return either when the requested amount of data has been read or when the timeout period has lapsed, whichever occurs first. If the Overlap Enable option is enabled, the function will return immediately and the operation will be carried out in the background. In this case the application must either use I/O Completion Notification or query the read side I/O Status parameters to determine when the operation completes and how much data was read. The service reads up to the requested number of bytes, which must be a multiple of four bytes (only full 32-bit data values are retrieved). Only a single read operation can be active at a time. If a request is made while a read operation is in progress, then the new request will fail. If overlapped I/O is requested and the function returns an error status, the overlapped operation may not have been initiated. In most cases, no matter how an I/O operation ends though, even if it could not be started, an I/O completion event will be triggered if at all possible.

> **NOTE:** The API automatically advances the read side I/O Offset parameter according to the amount of data read. The advancement corresponds to the "`count`" amount reported by this service.
>
> **NOTE:** Only a single PIO based I/O request can be active at a time. If a PIO write request is active when a PIO read request is made, then the read request will fail.
>
> **NOTE:** Thoroughly examine the various I/O Parameters to determine the settings required for each application.

Prototype

```
U32 dmi32_read(void* handle, void* buffer, U32 bytes, U32* count);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`buffer`	This is where the retrieved data is stored. It must be large enough to store all of the data requested and it must remain accessible by the API until the operation completes. The pointer must not be NULL. The buffer can be an application allocated buffer or either of the API Buffers.

`bytes`	This is the desired number of bytes to retrieve. The API will limit this to the value specified by the macro `GSC_IO_STATUS_COUNT_MASK` and will round it down to an integral multiple of four bytes.
`count`	The API records the number of bytes actually transferred here. The value recorded may be less than the amount requested due to various factors; request limits, timeout, abort or an error.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_read(
    void*   handle,
    void*   buffer,
    U32     bytes,
    U32*    count,
    int     verbose)
{
    U32 status;

    status  = dmi32_read(handle, buffer, bytes, count);

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

    return(status);
}
```

5.10. dmi32_reg_mod()

This function is the entry point to performing a read-modify-write on a register. The function should only be called after a successful open of the respective device via `dmi32_open()`. Only the DMI32 firmware registers (those defined inside `dmi32_api.h`) may be modified. All PCI and PLX registers are read-only.

Prototype

```
U32 dmi32_reg_mod(void* handle, U32 reg, U32 value, U32 mask);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`reg`	This is the register to access. PCI and PLX registers are read-only.
`value`	This is the desired value to apply. Bits not referenced by the mask are ignored.
`mask`	This specifies the "`value`" bits to modify. If a bit is set here, then the corresponding "`value`" bit will be applied. The remaining "`value`" bits are ignored.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_bcr_mod(
    void*   handle,
    U32     value,
    U32     mask,
    int     verbose)
{
    U32 status;

    status  = dmi32_reg_mod(handle, DMI32_BCR, value, mask);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_reg_mod() failure: %ld\n", (long) status);
    }
    else
    {
        printf("BCR Modify:\n");
        printf("  Value: 0x%lX\n", (long) value);
        printf("  Mask:  0x%lX\n", (long) mask);
    }

    return(status);
}
```

5.11. dmi32_reg_read()

This function is the entry point to reading the value from a DMI32 register. The function should only be called after a successful open of the respective device via `dmi32_open()`. All DMI32 registers may be read.

Prototype

```
U32 dmi32_reg_read(void* handle, U32 reg, U32* value);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`reg`	This is the register to access.
`value`	The value read is recorded here. If this is NULL then no action is taken.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_bcr_read(void* handle, U32* value, int verbose)
{
    U32 status;

    status  = dmi32_reg_read(handle, DMI32_BCR, value);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_reg_read() failure: %ld\n", (long) status);
    }
    else
    {
        printf("BCR Read: 0x%lX\n", (long) value[0]);
    }

    return(status);
}
```

5.12. dmi32_reg_write()

This function is the entry point to writing to a register. The function should only be called after a successful open of the respective device via `dmi32_open()`. Only the DMI32 firmware registers (those defined inside `dmi32_api.h`) may be modified. All PCI and PLX registers are read-only.

Prototype

```
U32 dmi32_reg_write(void* handle, U32 reg, U32 value);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`reg`	This is the register to access.
`value`	The value to write to the register.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_bcr_write(void* handle, U32 value, int verbose)
{
    U32 status;

    status  = dmi32_reg_write(handle, DMI32_BCR, value);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_reg_write() failure: %ld\n", (long) status);
    }
    else
    {
        printf("BCR Write: 0x%lX\n", (long) value);
    }

    return(status);
}
```

5.13. dmi32_reset()

This function is the entry point to perform a device hardware reset. The function should only be called after a successful open of the respective device via `dmi32_open()`. In doing this, any I/O operations in progress are aborted.

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

Prototype

```
U32 dmi32_reset(void* handle);
```

Argument	**Description**
`handle`	This is an API device handle obtained via `dmi32_open()`.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"
```

```
U32 dmi32_dsl_reset(void* handle, int verbose)
{
    U32 status;

    status  = dmi32_reset(handle);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_reset() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Device Reset: 0x%lX\n", (long) handle);
    }

    return(status);
}
```

5.14. dmi32_seek()

This function is the entry point to changing the I/O Offset parameter for subsequent read and/or write service calls. The function should only be called after a successful open of the respective device via `dmi32_open()`. This service is similar to the ANSI `fseek()` call available for files as the offset is similar to a file pointer. The read and write services each maintain their own, independent logical file pointer. The following table lists macros associated with this service.

NOTE: Using this service is equivalent to accessing the I/O Offset parameters.

Macro (Services)	Description
`DMI32_SEEK_RX(h,s)`	This sets the logical file pointer for read operations.
`DMI32_SEEK_TX(h,s)`	This sets the logical file pointer for write operations.
`DMI32_SEEK_TX_RX(h,s)`	This sets the logical file pointer for read and write operations.

Prototype

```
U32 dmi32_seek(void* handle, U32 which, U32 offset);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`which`	This is any bitwise or'd combination of `DMI32_WHICH_IO_XXX` bits. Set the bits according to the read or write side to be affected. If none are set then no action is taken.
`offset`	This is the offset in bytes from the beginning of DRAM. This must be on a four byte boundary and should not go beyond the end of memory. The actual offset applied may differ depending on the request, the amount of installed DRAM and other parameter settings.

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

This service is also accompanied by a small set of utility macros designed to enhance use of this API service. These are given in the table below. In the table, the argument “`h`” refers to the device handle used to access the device, the argument “`s`” refers to the application’s desired offset and the argument “`w`” refers to any bitwise or’d combination of the `DMI32_WHICH_IO_RX` and `DMI32_WHICH_IO_TX` macros.

Macro (Services)	**Description**
`DMI32_SEEK(h,w,s)`	This requests a seek for the referenced I/O direction.
`DMI32_SEEK_RX(h,s)`	This requests a seek for the I/O read direction.
`DMI32_SEEK_TX(h,s)`	This requests a seek for the I/O write direction.

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_rx_seek(void* handle, U32 offset, int verbose)
{
    U32 status;

    status  = dmi32_seek(handle, DMI32_WHICH_RX, offset);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_seek() failure: %ld\n", (long) status);
    }
    else
    {
        printf("I/O Read Offset set to 0x%lX:\n", (long) offset);
    }

    return(status);
}
```

5.15. dmi32_status_text()

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

Prototype

```
U32 dmi32_status_text(U32 status, char* text, size_t size);
```

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

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_status_text(U32 stat, int verbose)
{
    char    buf[128];
    U32     status;

    status  = dmi32_status_text(stat, buf, sizeof(buf));

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_status_text() failure: 0x%lX\n", (long) status);
    }
    else
    {
        printf("Status: 0x%lX: %s\n", (long) stat, buf);
    }

    return(status);
}
```

5.16. dmi32_tell()

This function is the entry point to retrieving the I/O Offset parameter used by the read and write services. The function should only be called after a successful open of the respective device via `dmi32_open()`. This service is similar to the ANSI `ftell()` call available for files as the offset is similar to a file pointer. The read and write services each maintain their own, independent logical file pointer. The following table lists macros associated with this service.

NOTE: Using this service is equivalent to accessing the I/O Offset parameters.

Macro (Services)	Description
`DMI32_TELL_RX(h,g)`	This retrieves the logical file pointer for read operations.
`DMI32_TELL_TX(h,g)`	This retrieves the logical file pointer for write operations.

Prototype

```
U32 dmi32_tell(void* handle, U32 which, unsigned long* offset);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`which`	This is any bitwise or'd combination of `DMI32_WHICH_IO_XXX` bits. Set the bits according to the read or write side to be affected. If none are set then no action is taken.
`offset`	The current offset is recorded here. If NULL the offset is not reported.

Return Value	Description
`GSC_SUCCESS`	The operation succeeded.

Otherwise …	A `GSC_XXX` error status reflecting the problem encountered.

This service is also accompanied by a small set of utility macros designed to enhance use of this API service. These are given in the table below. In the table, the argument "`h`" refers to the device handle used to access the device, the argument "`g`" is a `U32*` where the retrieved offset is recorded and the argument "`w`" refers to any bitwise or'd combination of the `DMI32_WHICH_IO_RX` and `DMI32_WHICH_IO_TX` macros. If the "which" bits reference both I/O directions, then any data returned will refer to the Rx side as it is processed last. If the pointer is NULL then nothing is retrieved.

Macro (Services)	**Description**
`DMI32_TELL(h,w,g)`	This requests the offset for the referenced I/O direction.
`DMI32_TELL_RX(h,g)`	This requests the offset for the I/O read direction.
`DMI32_TELL_TX(h,g)`	This requests the offset for the I/O write direction.

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_rx_tell(void* handle, U32* offset, int verbose)
{
    U32             status;
    unsigned long   value;

    status      = dmi32_tell(handle, DMI32_WHICH_RX, &value);
    offset[0]   = (U32) value;

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_tell() failure: %ld\n", (long) status);
    }
    else
    {
        printf("I/O Read Offset is 0x%lX:\n", (long) offset[0]);
    }

    return(status);
}
```

5.17. dmi32_version_get()

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

Macro (Services)	**Description**
`DMI32_VERSION_GET_DRIVER(h,b,s)`	This retrieves the driver version string.
`DMI32_VERSION_GET_LIBRARY(h,b,s)`	This retrieves the API Library version string.

Prototype

```
U32 dmi32_version_get(void* handle, U8 id, char* version, size_t size);
```

Argument	**Description**
`handle`	This is an API device handle obtained via `dmi32_open()`. This is ignored except when accessing the Device Driver's version data.
`id`	This indicates the version number desired. It should be either `GSC_VERSION_LIBRARY` or `GSC_VERSION_DRIVER`.
`version`	This is a buffer where the version string is recorded.
`size`	This is the size of the above buffer.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_version_get(void* handle, U8 id, int verbose)
{
    U32     status;
    char    ver[32];

    status  = dmi32_version_get(handle, id, ver, sizeof(ver));

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_version_get() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Version: %s\n", ver);
    }

    return(status);
}
```

5.18. dmi32_write()

This function is the entry point to writing DRAM data to a DMI32. The function should only be called after a successful open of the respective device via `dmi32_open()`. The operation will be carried out according to the current set of write side I/O Parameters. If the Overlap Enable option is disabled, then the function will block and return either when the requested amount of data has been written or when the timeout period has lapsed, which ever occurs first. If the Overlap Enable option is enabled, then function will return immediately and the operation will be carried out in the background. In this case the application must either use I/O Completion Notification or query the write side I/O Status parameters to determine when the operation completes and how much data was written. The service writes up to the requested number of bytes, which must be a multiple of four bytes (only full 32-bit data values are written). Only a single write operation can be active at a time. If a request is made while a write operation is in progress, then the new request will fail. If overlapped I/O is requested and the function returns an error status, the overlapped operation may not have been initiated. In most cases, no matter how an I/O operation ends though, even if it could not be started, an I/O completion event will be triggered if at all possible.

> **NOTE:** The API automatically advances the write side I/O Offset parameter according to the amount of data written. The advancement corresponds to the "`count`" amount reported by this service.
>
> **NOTE:** Only a single PIO based I/O request can be active at a time. If a PIO read request is active when a PIO write request is made, then the write request will fail.
>
> **NOTE:** Thoroughly examine the various I/O Parameters to determine the settings required for each application.

Prototype

```
U32 dmi32_write(
    void*       handle,
    const void* buffer,
    U32         bytes,
    U32*        count);
```

Argument	**Description**
`handle`	This is an API device handle obtained via `dmi32_open()`.
`buffer`	This is the source for the data to send. It must remain accessible by the API until the operation completes. The pointer must not be NULL. The buffer can be an application allocated buffer or either of the API Buffers.
`bytes`	This is the desired number of bytes to write. The API will limit this to the value specified by the macro `GSC_IO_STATUS_COUNT_MASK` and will round it down to an integral multiple of four bytes.
`count`	The API records the number of bytes actually transferred here. The value recorded may be less than the amount requested due to various factors; request limits, timeout, abort or other errors.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_write(
    void*       handle,
    const void* buffer,
    U32         bytes,
    U32*        count,
    int         verbose)
{
    U32 status;
    status  = dmi32_write(handle, buffer, bytes, count);

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

    return(status);
}
```

5.19. dmi32_xfer_buf_get()

This function is the entry point to retrieving the current Transfer Buffer definition. The function should only be called after a successful open of the respective device via `dmi32_open()`. This service is an alternate and simpler method of obtaining the Transfer Buffer definition as it accesses both the Transfer Offset and Transfer Size parameters via a single call.

Prototype

```
U32 dmi32_xfer_buf_get(void* handle, U32* offset, U32* size);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`offset`	The current Transfer Buffer offset is recorded here. If NULL the offset is not reported.
`size`	The current Transfer Buffer size is recorded here. If NULL the size is not reported.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_xfer_buf_report(void* handle, int verbose)
{
    U32 offset;
    U32 size;
    U32 status;

    status  = dmi32_xfer_buf_get(handle, &offset, &size);

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        printf("dmi32_xfer_buf_get() failure: %ld\n", (long) status);
    }
    else
    {
        printf("Transfer Buffer:\n");
        printf("  Offset: 0x%lX\n", (long) offset);
        printf("  Size:   ");

        if (size == DMI32_XFER_SIZE_4GB)
            printf("4GB\n");
        else
            printf("0x%lX\n", (long) size);
    }

    return(status);
}
```

5.20. dmi32_xfer_buf_set()

This function is the entry point to updating the Transfer Buffer definition. The function should only be called after a successful open of the respective device via `dmi32_open()`. This service is an alternate and simpler method of updating the Transfer Buffer definition as it accesses both the Transfer Offset and Transfer Size parameters via a single call.

Prototype

```
U32 dmi32_xfer_buf_set(void* handle, U32 offset, U32 size);
```

Argument	Description
`handle`	This is an API device handle obtained via `dmi32_open()`.
`offset`	This is the desired Transfer Buffer offset. The actual offset applied may differ depending on the request, the amount of installed DRAM and other parameter settings.
`size`	This is the desired Transfer Buffer size. The actual size applied may differ depending on the request, the amount of installed DRAM and other parameter settings.

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

Example

```
#include <stdio.h>

#include "dmi32_api.h"
#include "dmi32_dsl.h"

U32 dmi32_dsl_xfer_buf_update(
    void*   handle,
    U32     offset,
    U32     size,
    int     verbose)
{
    U32 status;
```

```
	status	= dmi32_xfer_buf_set(handle, offset, size);

	if (!verbose)
	{
	}
	else if (status != GSC_SUCCESS)
	{
		printf("dmi32_xfer_buf_set() failure: %ld\n", (long) status);
	}
	else
	{
		printf("Transfer Buffer Requested:\n");
		printf("  Offset: 0x%lX\n", (long) offset);
		printf("  Size:   ");

		if (size == DMI32_XFER_SIZE_4GB)
			printf("4GB\n");
		else
			printf("0x%lX\n", (long) size);
	}

	return(status);
}
```

6. Configuration Parameters

The DMI32 and the API Library include a number of configurable features. These features are refered to by the API as Configuration Parameters. This section describes all of the DMI32 Configuration Parameters.

6.1. Parameter Macros

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

Parameter Categories	**Description**
`DMI32_IO_XXX`	These refer to the Input/Output Parameters. These pertain to data transfer between the host and the DMI32 DRAM memory.
`DMI32_IRQ_XXX`	These refer to the Interrupt Parameters.
`DMI32_MISC_XXX`	These refer to the Miscellaneous Parameters.
`DMI32_USER_IO_XXX`	These refer to the User Input/Output Parameters.
`DMI32_XFER_XXX`	These refer to the Transfer Parameters. These refer to data transfer across the cable interface.

6.1.1. Parameter Definitions

The first group of macros includes the parameter definitions. These are used to identify the specific parameter to be accessed. These macros begin with “`DMI32_`” and are followed immediately by upper case letters identifying the parameter category. For example “`DMI32_MISC_`” prefaces all Miscellaneous Parameter identifiers. These macros end with upper case letters indicating the name of the specific parameter. For example “`DMI32_MISC_STRICT_ARGUMENTS`” identifies the Miscellaneous Strict Arguments parameter.

6.1.2. Value Definitions

The second group of macros identifies predefined values associated with the respective parameters. These macros begin with the Parameter Definition and are followed by a single underscore (“`_`”) then upper case letters that reflect the meaning of the respective values. For example the macro “`DMI32_MISC_STRICT_ARGUMENTS_DISABLE`” is the value that represents the parameter’s *disabled* setting.

6.1.3. Service Definitions

The third group of macros performs operations on parameters. These are utility macros that retrieve parameter settings and states or assign parameter values. These macros include the Parameter Definition followed by a double underscore (“`__`”) then upper case letters that reflect the action to perform. For example “`DMI32_MISC_STRICT_ARGUMENTS__GET()`” retrieves the current setting of the Miscellaneous Strict Arguments parameter. These macros include arguments, which are described as follows.

6.1.3.1. Device Handle: h

In the service macros, the argument `h` refers to the device handle used to access the respective device. This handle is obtained by calling `dmi32_open()`. This argument must not be NULL.

6.1.3.2. Which Bits: w

In the service macros, the argument `w` refers to any combination of the `DMI32_WHICH_XXX` bits. Refer to paragraph 3.6 on page 16. With some parameters this argument is unused or is specified inside the macro's replacement text. In those cases the `w` is not included as a macro argument.

6.1.3.3. Set Value: s

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

6.1.3.4. Get Value: g

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

6.2. Input/Output Parameters

The purpose of the I/O Parameters is to permit access to and control of read (Rx) and write (Tx) I/O operations. All I/O Parameters are put in a default state when the device is opened and are returned to that state via the `dmi32_init()` service. The configuration of the I/O Parameters is retained entirely within the API and cannot be altered by any DMI32 registers. When accessing the I/O Parameters any number or combination of `DMI32_WHICH_TX` or `DMI32_WHICH_RX` may be used, even none. The write side I/O Parameters will be accessed only if the transmit bit is set and the read side I/O Parameters will be accessed only if the receive bit is set. If neither is set, then no action will be taken. The following table summarizes the I/O Parameters.

Parameter Macros	Description
`DMI32_IO_ABORT`	This refers to aborting an active I/O operation.
`DMI32_IO_ABORTED`	This refers to a previous I/O abort request.
`DMI32_IO_BUFFER_POINTER`	This refers to a pointer to an API Buffers.
`DMI32_IO_BUFFER_SIZE`	This refers to the size of an API Buffer.
`DMI32_IO_CALLBACK_ARG`	This refers to an arbitrary, application supplied callback argument.
`DMI32_IO_CALLBACK_FUNC`	This refers to an application supplied callback function.
`DMI32_IO_MODE`	This refers to the I/O data transfer mode (PIO or DMA).
`DMI32_IO_OFFSET`	This refers to the starting offset of a data transfer operation.
`DMI32_IO_OVERLAP_ENABLE`	This refers to the use of overlapping or blocking I/O operations.
`DMI32_IO_STATUS`	This refers to the status of the most recent I/O operation.
`DMI32_IO_TIMEOUT`	This refers to the I/O Timeout period in seconds.

6.2.1. I/O Parameter: Abort

The purpose of this parameter is to abort an ongoing I/O operation. This parameter is applicable to active I/O requests only. No action occurs if none are active at the time an abort request is made. There is also no affect on future I/O requests. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_IO_ABORT`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IO_ABORT_DEFAULT`	This is the default action to take, which is to do nothing.
`DMI32_IO_ABORT_NO`	As a "set" option this means do not perform an abort. As a "get" option it

	means that an abort did not occur.
`DMI32_IO_ABORT_YES`	As a “set” option this requests an abort. This value is never returned as a “get” option as the parameter auto clears after an abort request.

Macro (Services)	**Description**
`DMI32_IO_ABORT__GET(h,w,g)`	This requests if an abort has been requested.
`DMI32_IO_ABORT__RX_GET(h,g)`	This requests if an Rx abort has been requested.
`DMI32_IO_ABORT__RX_SET(h,s)`	This applies an option to an Rx I/O operation.
`DMI32_IO_ABORT__RX_YES(h)`	This aborts an Rx operation.
`DMI32_IO_ABORT__SET(h,w,s)`	This applies an option to an I/O operation.
`DMI32_IO_ABORT__TX_GET(h,g)`	This requests if a Tx abort has been requested.
`DMI32_IO_ABORT__TX_SET(h,s)`	This applies an option to a Tx I/O operation.
`DMI32_IO_ABORT__TX_YES(h)`	This aborts a Tx operation.
`DMI32_IO_ABORT__YES(h,w)`	This aborts an Rx and/or Tx operation.

6.2.2. I/O Parameter: Aborted

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

Macro (Parameter)	**Description**
`DMI32_IO_ABORTED`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_IO_ABORTED_NO`	This means that an I/O operation was not aborted.
`DMI32_IO_ABORTED_YES`	This means that an I/O operation was aborted.

Macro (Services)	**Description**
`DMI32_IO_ABORTED__GET(h,w,g)`	This retrieves the status of an operation.
`DMI32_IO_ABORTED__RX_GET(h,g)`	This retrieves the status of an Rx operation.
`DMI32_IO_ABORTED__TX_GET(h,g)`	This retrieves the status of a Tx operation.

6.2.3. I/O Parameter: Buffer Pointer

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

> **NOTE:** Applications must obtain a fresh pointer each time a change is made to the API Buffer size. Refer to the I/O Buffer Size parameter in the next subsection.

Macro (Parameter)	**Description**
`DMI32_IO_BUFFER_POINTER`	This is the identifier for this parameter.

Macro (Services)	**Description**
`DMI32_IO_BUFFER_POINTER__GET(h,w,g)`	This retrieves an API Buffer pointer.
`DMI32_IO_BUFFER_POINTER__RX_GET(h,g)`	This retrieves the Rx API Buffer pointer.
`DMI32_IO_BUFFER_POINTER__TX_GET(h,g)`	This retrieves the Tx API Buffer pointer.

6.2.4. I/O Parameter: Buffer Size

The purpose of this parameter is to adjust and retrieve the size of the respective API Buffer. Requesting a size of zero essentially frees any existing buffer. The following tables describe the macros associated with this parameter.

> **NOTE:** The API has no control over the amount of memory the OS will grant in response to an API Buffer allocation request. Each of the API Buffers is a contiguous block of memory requested of the OS by the Device Driver. The OS manages these types of resources differently than application memory resources so the size of the API Buffer obtained may be significantly less then requested by the application. Applications must therefore examine this parameter after it is adjusted to guard against memory protection faults.
>
> **NOTE:** Each time the application requests an API Buffer size change, the pointer used to access the buffer is likely to also change. Applications must therefore obtain a fresh pointer following a size change request. Refer to the I/O Buffer Pointer parameter description in the previous section.

Macro (Parameter)	**Description**
`DMI32_IO_BUFFER_SIZE`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_IO_BUFFER_SIZE_DEFAULT`	This is the default size, which is zero.
`DMI32_IO_BUFFER_SIZE_MASK`	This is a mask of the valid bits that may be used.

Macro (Services)	**Description**
`DMI32_IO_BUFFER_SIZE__GET(h,w,g)`	This retrieves a current size setting.
`DMI32_IO_BUFFER_SIZE__RESET(h,w)`	This requests a size change to zero.
`DMI32_IO_BUFFER_SIZE__RX_GET(h,g)`	This retrieves the current Rx size setting.
`DMI32_IO_BUFFER_SIZE__RX_RESET(h)`	This requests an Rx size change to zero.
`DMI32_IO_BUFFER_SIZE__RX_SET(h,s,g)`	This requests an Rx size change and retrieves the results.
`DMI32_IO_BUFFER_SIZE__SET(h,w,s,g)`	This requests a size change and retrieves the results.
`DMI32_IO_BUFFER_SIZE__TX_GET(h,g)`	This retrieves the current Tx size setting.
`DMI32_IO_BUFFER_SIZE__TX_RESET(h)`	This requests a Tx size change to zero.
`DMI32_IO_BUFFER_SIZE__TX_SET(h,s,g)`	This requests a Tx size change and retrieves the results.

6.2.5. I/O Parameter: Callback Argument

The purpose of this parameter is to modify and report the application provided argument that it receives as “`arg3`” for an I/O completion callback event. Applications are free to assign virtually any arbitrary value desired. The following tables describe the macros associated with this parameter.

> **NOTE:** Applications must remember that the macros `GSC_NO_CHANGE` and `GSC_DEFAULT` have special meaning when applying parameter modifications. If the application specific value being supplied for this parameter happens to equal either of these values, then the results will be according to the API’s use of these special values rather than the application’s intent.
>
> **NOTE:** This parameter can be accessed and altered during the callback, but the callback must return before subsequent callbacks can be made on the same I/O transfer direction.

Macro (Parameter)	**Description**
`DMI32_IO_CALLBACK_ARG`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_IO_CALLBACK_ARG_DEFAULT`	This is the default, which is zero.

Macro (Services)	Description
`DMI32_IO_CALLBACK_ARG__GET(h,w,g)`	This retrieves a current setting.
`DMI32_IO_CALLBACK_ARG__RESET(h,w)`	This requests a setting change to the default.
`DMI32_IO_CALLBACK_ARG__RX_GET(h,g)`	This retrieves the current Rx setting.
`DMI32_IO_CALLBACK_ARG__RX_RESET(h)`	This requests an Rx setting change to the default.
`DMI32_IO_CALLBACK_ARG__RX_SET(h,s)`	This requests an Rx setting change.
`DMI32_IO_CALLBACK_ARG__SET(h,w,s)`	This requests a setting change.
`DMI32_IO_CALLBACK_ARG__TX_GET(h,g)`	This retrieves the current Tx setting.
`DMI32_IO_CALLBACK_ARG__TX_RESET(h)`	This requests a Tx setting change to the default.
`DMI32_IO_CALLBACK_ARG__TX_SET(h,s)`	This requests a Tx setting change.

6.2.6. I/O Parameter: Callback Function

The purpose of this parameter is to modify and report the application provided function pointer for I/O completion events. The specified callback function must be of the type `dmi32_callback_func_t` (page 21). The given function is called for each completed I/O operation, regardless of how the operation completes. The following tables describe the macros associated with this parameter.

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

Macro (Parameter)	Description
`DMI32_IO_CALLBACK_FUNC`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IO_CALLBACK_FUNC_DEFAULT`	This is the default, which is NULL. This disables the callback feature.

Macro (Services)	Description
`DMI32_IO_CALLBACK_FUNC__GET(h,w,g)`	This retrieves a current function pointer.
`DMI32_IO_CALLBACK_FUNC__RESET(h,w)`	This requests a setting change to the default, which disables the callback feature.
`DMI32_IO_CALLBACK_FUNC__RX_GET(h,g)`	This retrieves the current Rx function pointer.
`DMI32_IO_CALLBACK_FUNC__RX_RESET(h)`	This requests an Rx setting change to the default, which disables the callback feature.
`DMI32_IO_CALLBACK_FUNC__RX_SET(h,s)`	This requests an Rx function pointer change.
`DMI32_IO_CALLBACK_FUNC__SET(h,w,s)`	This requests a function pointer change.
`DMI32_IO_CALLBACK_FUNC__TX_GET(h,g)`	This retrieves the current Tx function pointer.
`DMI32_IO_CALLBACK_FUNC__TX_RESET(h)`	This requests a Tx setting change to the default, which disables the callback feature.
`DMI32_IO_CALLBACK_FUNC__TX_SET(h,s)`	This requests a Tx function pointer change.

6.2.7. I/O Parameter: Mode

The purpose of this parameter is to modify and report the data transfer mode used by the API during I/O requests. In virtually all cases, the default mode of DMA offers the best performance. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_IO_MODE`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IO_MODE_DEFAULT`	This selects the default, which is DMA.
`DMI32_IO_MODE_DMA`	This selects the non-Demand Mode DMA.

`DMI32_IO_MODE_PIO`	This selects Programmed I/O, which used repetitive register reads and writes.

Macro (Services)	**Description**
`DMI32_IO_MODE__DMA(h,w)`	This requests a setting of DMA.
`DMI32_IO_MODE__GET(h,w,g)`	This retrieves a current setting.
`DMI32_IO_MODE__PIO(h,w)`	This requests a setting of PIO.
`DMI32_IO_MODE__RESET(h,w)`	This requests a setting change to the default.
`DMI32_IO_MODE__RX_DMA(h)`	This requests an Rx setting of DMA.
`DMI32_IO_MODE__RX_GET(h,g)`	This retrieves the current Rx setting.
`DMI32_IO_MODE__RX_PIO(h)`	This requests an Rx setting of PIO.
`DMI32_IO_MODE__RX_RESET(h)`	This requests an Rx setting change to the default.
`DMI32_IO_MODE__RX_SET(h,s)`	This requests an Rx setting change.
`DMI32_IO_MODE__SET(h,w,s)`	This requests a setting change.
`DMI32_IO_MODE__TX_DMA(h)`	This requests a Tx setting of DMA.
`DMI32_IO_MODE__TX_GET(h,g)`	This retrieves the current Tx setting.
`DMI32_IO_MODE__TX_PIO(h)`	This requests a Tx setting of PIO.
`DMI32_IO_MODE__TX_RESET(h)`	This requests a Tx setting change to the default.
`DMI32_IO_MODE__TX_SET(h,s)`	This requests a Tx setting change.

6.2.8. I/O Parameter: Offset

The purpose of this parameter is to modify and report the starting offset within DRAM used by subsequent read and write operations. The I/O Offset is equivalent to a file pointer maintained during file I/O operations. Setting the offset is analogous to a `seek()` request and retrieving the offset is analogous to a `tell()` request. The parameter is maintained independently for reads and writes and each is adjusted automatically by the API during I/O requests. Offsets must be made to four byte boundaries. The following tables describe the macros associated with this parameter.

Macro (Parameter)	**Description**
`DMI32_IO_OFFSET`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_IO_OFFSET_DEFAULT`	This selects the default.
`DMI32_IO_OFFSET_MASK`	This is the set of bits that may be set within an offset value.
`DMI32_IO_OFFSET_MAX`	This selects the maximum offset.
`DMI32_IO_OFFSET_MIN`	This selects the minimum offset.

Macro (Services)	**Description**
`DMI32_IO_OFFSET__GET(h,w,g)`	This retrieves a current setting.
`DMI32_IO_OFFSET__RESET(h,w)`	This requests a setting change to the default.
`DMI32_IO_OFFSET__RX_GET(h,g)`	This retrieves the current Rx setting.
`DMI32_IO_OFFSET__RX_RESET(h)`	This requests an Rx setting change to the default.
`DMI32_IO_OFFSET__RX_SET(h,s)`	This requests an Rx setting change.
`DMI32_IO_OFFSET__SET(h,w,s)`	This requests a setting change.
`DMI32_IO_OFFSET__TX_GET(h,g)`	This retrieves the current Tx setting.
`DMI32_IO_OFFSET__TX_RESET(h)`	This requests a Tx setting change to the default.
`DMI32_IO_OFFSET__TX_SET(h,s)`	This requests a Tx setting change.

6.2.9. I/O Parameter: Overlap Enable

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

Macro (Parameter)	Description
`DMI32_IO_OVERLAP_ENABLE`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IO_OVERLAP_ENABLE_DEFAULT`	This selects the default, which is *no*.
`DMI32_IO_OVERLAP_ENABLE_NO`	This selects the *no* option. I/O requests block until the request completes or times out.
`DMI32_IO_OVERLAP_ENABLE_YES`	This selects the *yes* option. I/O requests return immediately while the data transfer occurs in the background.

Macro (Services)	Description
`DMI32_IO_OVERLAP_ENABLE__GET(h,w,g)`	This retrieves a current setting.
`DMI32_IO_OVERLAP_ENABLE__NO(h,w)`	This requests a setting of *no*.
`DMI32_IO_OVERLAP_ENABLE__RESET(h,w)`	This requests a setting change to the default.
`DMI32_IO_OVERLAP_ENABLE__RX_GET(h,g)`	This retrieves the current Rx setting.
`DMI32_IO_OVERLAP_ENABLE__RX_NO(h)`	This requests an Rx setting of *no*.
`DMI32_IO_OVERLAP_ENABLE__RX_RESET(h)`	This requests an Rx setting change to the default.
`DMI32_IO_OVERLAP_ENABLE__RX_SET(h,s)`	This requests an Rx setting change.
`DMI32_IO_OVERLAP_ENABLE__RX_YES(h)`	This requests an Rx setting of *yes*.
`DMI32_IO_OVERLAP_ENABLE__SET(h,w,s)`	This requests a setting change.
`DMI32_IO_OVERLAP_ENABLE__TX_GET(h,g)`	This retrieves the current Tx setting.
`DMI32_IO_OVERLAP_ENABLE__TX_NO(h)`	This requests a Tx setting of *no*.
`DMI32_IO_OVERLAP_ENABLE__TX_RESET(h)`	This requests a Tx setting change to the default.
`DMI32_IO_OVERLAP_ENABLE__TX_SET(h,s)`	This requests a Tx setting change.
`DMI32_IO_OVERLAP_ENABLE__TX_YES(h)`	This requests a Tx setting of *yes*.
`DMI32_IO_OVERLAP_ENABLE__YES(h,w)`	This requests a setting of *yes*.

6.2.10. I/O Parameter: Status

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

Macro (Parameter)	Description
`DMI32_IO_STATUS`	This is the identifier for this parameter.

Macro (Services)	Description
`DMI32_IO_STATUS__GET(h,w,g)`	This retrieves a current status.
`DMI32_IO_STATUS__RX_GET(h,g)`	This retrieves the current Rx status.
`DMI32_IO_STATUS__TX_GET(h,g)`	This retrieves the current Tx status.

6.2.11. I/O Parameter: Timeout

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

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

Macro (Parameter)	Description
`DMI32_IO_TIMEOUT`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IO_TIMEOUT_DEFAULT`	This selects the default, which is 10 seconds
`DMI32_IO_TIMEOUT_MAX`	This selects the maximum timeout limit, which is one hour.
`DMI32_IO_TIMEOUT_NO_WAIT`	This sets the timeout to zero (0) seconds. This means the I/O request will terminate rather than wait for additional data transfer to occur.

Macro (Services)	Description
`DMI32_IO_TIMEOUT__GET(h,w,g)`	This retrieves a current setting.
`DMI32_IO_TIMEOUT__NO_WAIT(h,w)`	This requests a setting to *do not wait*.
`DMI32_IO_TIMEOUT__RESET(h,w)`	This requests a setting change to the default.
`DMI32_IO_TIMEOUT__RX_GET(h,g)`	This retrieves the current Rx setting.
`DMI32_IO_TIMEOUT__RX_NO_WAIT(h)`	This requests an Rx setting to *do not wait*.
`DMI32_IO_TIMEOUT__RX_RESET(h)`	This requests an Rx setting change to the default.
`DMI32_IO_TIMEOUT__RX_SET(h,s)`	This requests an Rx setting change.
`DMI32_IO_TIMEOUT__SET(h,w,s)`	This requests a setting change.
`DMI32_IO_TIMEOUT__TX_GET(h,g)`	This retrieves the current Tx setting.
`DMI32_IO_TIMEOUT__TX_NO_WAIT(h)`	This requests a Tx setting to *do not wait*.
`DMI32_IO_TIMEOUT__TX_RESET(h)`	This requests a Tx setting change to the default.
`DMI32_IO_TIMEOUT__TX_SET(h,s)`	This requests a Tx setting change.

6.3. Interrupt Parameters

The purpose of the Interrupt Parameters is to permit access to and control of the DMI32 hardware based interrupts. All Interrupt Parameters are put in a default state when the device is opened and are returned to that state via the `dmi32_init()` service. The hardware based interrupt configuration is returned to its default state via the `dmi32_reset()` service. The configuration of the Interrupt Parameters is retained mostly within the DMI32 firmware registers. Applications have access to the DMI32 interrupt registers but it is advised that they be accessed only through the Interrupt Parameter services. When using the service `dmi32_config()` any number or combination of `DMI32_WHICH_IRQ_XXX` bits may be used, even none. An interrupt is accessed only if it's respective "which" bit is set. If none is set, then no action will be taken. The following table summarizes the Interrupt Parameters.

Parameter Macros	Description
`DMI32_IRQ_CALLBACK_ARG`	This refers to an arbitrary, application supplied callback argument.
`DMI32_IRQ_CALLBACK_FUNC`	This refers to an application supplied callback function.
`DMI32_IRQ_ENABLE`	This refers to the enabled/disabled state of an interrupt.
`DMI32_IRQ_STATE`	This refers to the active/inactive state of an interrupt source.

6.3.1. Interrupt Parameter: Callback Argument

The purpose of this parameter is to modify and report the application provided argument that is receives as "`arg3`" during an interrupt callback event. Applications are free to assign virtually any arbitrary value desired. The following tables describe the macros associated with this parameter.

> **NOTE:** Applications must remember that the macros `GSC_NO_CHANGE` and `GSC_DEFAULT` have special meaning when applying parameter modifications. If the application specific value being supplied for this parameter happens to equal either of these values, then the results will be according to the API's use of these special values rather than the application's intent.
>
> **NOTE:** This parameter can be accessed and altered during the callback, but the callback must return before subsequent callbacks can be made on the same interrupt.

Macro (Parameter)	Description
`DMI32_IRQ_CALLBACK_ARG`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IRQ_CALLBACK_ARG_DEFAULT`	This is the default, which is zero.

Macro (Services)	Description
`DMI32_IRQ_CALLBACK_ARG__GET(h,w,g)`	This retrieves a current setting.
`DMI32_IRQ_CALLBACK_ARG__RESET(h,w)`	This requests a setting change to the default.
`DMI32_IRQ_CALLBACK_ARG__SET(h,w,s)`	This requests a setting change.
`DMI32_IRQ_CALLBACK_ARG__XXX_GET(h,g)`	This retrieves the current interrupt “`XXX`” setting. *
`DMI32_IRQ_CALLBACK_ARG__XXX_RESET(h)`	This requests a setting change for interrupt “`XXX`” to the default. *
`DMI32_IRQ_CALLBACK_ARG__XXX_SET(h,s)`	This requests an interrupt “`XXX`” setting change. *

* The `XXX` refers individually to `BS`, `BE`, `UIOA`, `UIOB` and `MBUR`.

6.3.2. Interrupt Parameter: Callback Function

The purpose of this parameter is to modify and report the application provided callback function pointer for an interrupt callback event. The specified callback function must be of the type `dmi32_callback_func_t` (page 21). The given function is called for each respective interrupt. The following tables describe the macros associated with this parameter.

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

Macro (Parameter)	Description
`DMI32_IRQ_CALLBACK_FUNC`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IRQ_CALLBACK_FUNC_DEFAULT`	This is the default, which is NULL.

Macro (Services)	Description
`DMI32_IRQ_CALLBACK_FUNC__GET(h,w,g)`	This retrieves a current function pointer.
`DMI32_IRQ_CALLBACK_FUNC__RESET(h,w)`	This requests a function pointer change to the default, which disables callback notification.
`DMI32_IRQ_CALLBACK_FUNC__SET(h,w,s)`	This requests a function pointer change.
`DMI32_IRQ_CALLBACK_FUNC__XXX_GET(h,g)`	This retrieves the current interrupt “`XXX`” function pointer. *
`DMI32_IRQ_CALLBACK_FUNC__XXX_RESET(h)`	This requests a function pointer change for interrupt “`XXX`” to the default, which disables callback notification.
`DMI32_IRQ_CALLBACK_FUNC__XXX_SET(h,s)`	This requests an interrupt “`XXX`” function pointer change. *

* The `XXX` refers individually to `BS`, `BE`, `UIOA`, `UIOB` and `MBUR`.

6.3.3. Interrupt Parameter: Enable

The purpose of this parameter is to modify and report the enabled state of the respective interrupt. This parameter is used to enable and disable generation of interrupts for the respective interrupt source. An interrupt source will not generate an interrupt unless the interrupt is enabled. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_IRQ_ENABLE`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IRQ_ENABLE_DEFAULT`	This is the default, which is disabled.
`DMI32_IRQ_ENABLE_NO`	This option disables the interrupt.
`DMI32_IRQ_ENABLE_YES`	This option enables the interrupt.

Macro (Services)	Description
`DMI32_IRQ_ENABLE__GET(h,w,g)`	This retrieves a current setting.
`DMI32_IRQ_ENABLE__NO(h,w)`	This requests that a number of interrupts be disabled.
`DMI32_IRQ_ENABLE__RESET(h,w)`	This requests a setting change to the default.
`DMI32_IRQ_ENABLE__SET(h,w,s)`	This requests a setting change.
`DMI32_IRQ_ENABLE__XXX_GET(h,g)`	This retrieves the current interrupt “`XXX`” setting.
`DMI32_IRQ_ENABLE__XXX_NO(h)`	This requests that interrupt “`XXX`” be disabled.
`DMI32_IRQ_ENABLE__XXX_RESET(h)`	This requests a setting change for interrupt “`XXX`” to the default.
`DMI32_IRQ_ENABLE__XXX_SET(h,s)`	This requests an interrupt “`XXX`” setting change.
`DMI32_IRQ_ENABLE__XXX_YES(h)`	This requests that interrupt “`XXX`” be enabled.
`DMI32_IRQ_ENABLE__YES(h,w)`	This requests that a number of interrupts be enabled.

* The `XXX` refers individually to `BS`, `BE`, `UIOA`, `UIOB` and `MBUR`.

6.3.4. Interrupt Parameter: State

The purpose of this read-only parameter is to report the state of the respective interrupt source. This parameter reports whether or not an interrupt source is active. If an interrupt source is active, then it will generate an interrupt when enabled. The following tables describe the macros associated with this parameter.

> **NOTE:** If an interrupt is enabled and it becomes active, the interrupt will be serviced. The result of this is that the state for an enabled interrupt will always appear as inactive. The state for interrupts which are disabled may be reported as either active or inactive, depending on the interrupt sources’ state at the time it is accessed.

Macro (Parameter)	Description
`DMI32_IRQ_STATE`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_IRQ_STATE_ACTIVE`	This reflects that the source was active.
`DMI32_IRQ_STATE_INACTIVE`	This reflects that the source was inactive.

Macro (Services)	Description
`DMI32_IRQ_STATE__GET(h,w,g)`	This retrieves a current state.
`DMI32_IRQ_STATE__XXX_GET(h,g)`	This retrieves the current interrupt “`XXX`” state. *

* The `XXX` refers individually to `BS`, `BE`, `UIOA`, `UIOB` and `MBUR`.

6.4. Miscellaneous Parameters

The purpose of the Miscellaneous Parameters is to permit access to and control of DMI32 parameters which do not readily fit into the other parameter categories. All Miscellaneous Parameters are put in a default state when the device is opened and are returned to that state via the `dmi32_init()` service. The hardware based Miscellaneous Parameters are returned to their default states via the `dmi32_reset()` service. The configuration of one or more Miscellaneous Parameters is retained within the DMI32 firmware registers. Applications have access to these

DMI32 registers but it is advised that these features be accessed only through the Miscellaneous Parameter services. When using the service `dmi32_config()` the Miscellaneous Parameters ignores the "which" bit parameter. The following table summarizes the Miscellaneous Parameters.

Parameter Macros	**Description**
`DMI32_MISC_BOARD_JUMPERS`	This refers to the board's User Jumpers.
`DMI32_MISC_DRAM_CAPACITY_MB`	This refers to the board's DRAM capacity.
`DMI32_MISC_DRAM_SIZE_MB`	This refers to the board's current DRAM size.
`DMI32_MISC_FEATURES`	This refers to support for various board features.
`DMI32_MISC_MAP_GSC_REGS`	This refers to mapping of the GSC firmware registers into API memory space.
`DMI32_MISC_MAP_PLX_REGS`	This refers to mapping of the PLX feature set registers into API memory space.
`DMI32_MISC_STRICT_ARGUMENTS`	This refers to how the API responds when it receives application argument values that are invalid or unrecognized.
`DMI32_MISC_STRICT_CONFIG`	This refers to how the API responds when it receives application requests that are invalid given the board's current configuration.
`DMI32_MISC_STRICT_DRAM_LIMITS`	This refers to how the API responds when it receives application argument values that are outside the board's current DRAM memory range.

6.4.1. Miscellaneous Parameter: Board Jumpers

The purpose of this read-only parameter is to report the state of the User Jumper pins on the board. These jumpers can be used to distinguish between multiple boards placed within the same system. The jumper state is reported in the lower two bits of the value retrieved. The following tables describe the macros associated with this parameter.

Macro (Parameter)	**Description**
`DMI32_MISC_BOARD_JUMPERS`	This is the identifier for this parameter.

Macro (Services)	**Description**
`DMI32_MISC_BOARD_JUMPERS__GET(h,g)`	This retrieves the current state.

6.4.2. Miscellaneous Parameter: DRAM Capacity MB

The purpose of this read-only parameter is to report the DMI32's DRAM memory capacity, which is the maximum amount of memory supported. This refers to the board's capability, which may exceed the amount of memory actually installed. The capacity is reported in megabyte units. The following tables describe the macros associated with this parameter.

Macro (Parameter)	**Description**
`DMI32_MISC_DRAM_CAPACITY_MB`	This is the identifier for this parameter.

Macro (Services)	**Description**
`DMI32_MISC_DRAM_CAPACITY_MB__GET(h,g)`	This retrieves the DRAM capacity.

6.4.3. Miscellaneous Parameter: DRAM Size MB

The purpose of this read-only parameter is to report the size of the DMI32's DRAM memory. This refers to the amount of memory actually installed, and may be less than the board's capacity. The size is reported in megabyte units. In the `dmi32_misc_param_t` structure the parameter is listed separately. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_MISC_DRAM_SIZE_MB`	This is the identifier for this parameter.

Macro (Services)	Description
`DMI32_MISC_DRAM_SIZE_MB__GET(h,g)`	This retrieves the DRAM size.

6.4.4. Miscellaneous Parameter: Features

The purpose of this read-only parameter is to determine the firmware's support for various features. The feature in question is passed as the "`set`" value. Support for the specified feature is reported in the "`get`" value. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_MISC_FEATURES`	This is the identifier for this parameter.

Macro (Set Values)	Description
`DMI32_MISC_FEATURES_FR`	This refers to the Feature Register.
`DMI32_MISC_FEATURES_IRQ_MBUR`	This refers to the Multi-Buffer Under Run interrupt.
`DMI32_MISC_FEATURES_LMBR`	This refers to the Local Memory Bank Register.
`DMI32_MISC_FEATURES_SW_DIR`	This refers to the software selectable Start Word Direction feature.

Macro (Get Values)	Description
`DMI32_MISC_FEATURES_ABSENT`	This indicates that the feature is unsupported in firmware.
`DMI32_MISC_FEATURES_PRESENT`	This indicates that the feature is supported in firmware.

Macro (Services)	Description
`DMI32_MISC_FEATURES__GET(h,s,g)`	This requests support status for a specified feature.
`DMI32_MISC_FEATURES__XXX(h,g)`	This requests support status for the respective feature. *

* The `XXX` refers individually to `FR`, `LMBR`, `IRQ_MBUR` and `SW_DIR`.

6.4.5. Miscellaneous Parameter: GSC Register Mapping

The purpose of this parameter is to control and report the mapping of GSC registers into API memory space. This parameter should always be enabled, even though the registers are not directly accessible by applications. If it is disabled, the API's access to DMI32 firmware registers operates with reduced efficiency. The following tables describe the macros associated with this parameter.

> **NOTE:** There are circumstances where this feature cannot be enabled and utilized. This is usually limited to embedded hosts here the BIOS doesn't place all PCI memory access regions on CPU Page Size boundaries. If the BIOS cannot be configured to utilize such boundaries, then API performance is degraded.
>
> **NOTE:** Parameter access utility macros are limited for this parameter as it should always be enabled. The parameter should only be disabled for testing purposes.

Macro (Parameter)	Description
`DMI32_MISC_MAP_GSC_REGS`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_MISC_MAP_GSC_REGS_DEFAULT`	This is the default, which is *enabled*.
`DMI32_MISC_MAP_GSC_REGS_DISABLE`	This refers to the *disabled* state. When disabled, access to firmware registers must go through the Device Driver, which reduces efficiency.

DMI32_MISC_MAP_GSC_REGS_ENABLE	This refers to the *enabled* state. When enabled, access to firmware registers is done entirely within the API, which increases efficiency.

Macro (Services)	**Description**
DMI32_MISC_MAP_GSC_REGS__ENABLE(h)	This requests that the option be enabled.
DMI32_MISC_MAP_GSC_REGS__GET(h,g)	This requests the current setting.
DMI32_MISC_MAP_GSC_REGS__RESET(h)	This requests a setting change to the default.
DMI32_MISC_MAP_GSC_REGS__SET(h,s)	This requests a change to the current setting.

6.4.6. Miscellaneous Parameter: PLX Register Mapping

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

> **NOTE:** There are circumstances where this feature cannot be enabled and utilized. This is usually limited to embedded hosts here the BIOS doesn't place all PCI memory access regions on CPU Page Size boundaries. If the BIOS cannot be configured to utilize such boundaries, then API performance is degraded.
>
> **NOTE:** Parameter access utility macros are limited for this parameter as it should always be enabled. The parameter should only be disabled for testing purposes.

Macro (Parameter)	**Description**
DMI32_MISC_MAP_PLX_REGS	This is the identifier for this parameter.

Macro (Values)	**Description**
DMI32_MISC_MAP_PLX_REGS_DEFAULT	This is the default, which is *enabled*.
DMI32_MISC_MAP_PLX_REGS_DISABLE	This refers to the *disabled* state. When disabled, access to PLX registers must go through the Device Driver, which reduces efficiency.
DMI32_MISC_MAP_PLX_REGS_ENABLE	This refers to the *enabled* state. When enabled, access to PLX registers is done entirely within the API, which increases efficiency.

Macro (Services)	**Description**
DMI32_MISC_MAP_PLX_REGS__ENABLE(h)	This request that the option be enabled.
DMI32_MISC_MAP_PLX_REGS__GET(h,g)	This requests the current setting.
DMI32_MISC_MAP_PLX_REGS__RESET(h)	This requests a setting change to the default.
DMI32_MISC_MAP_PLX_REGS__SET(h,s)	This request a change to the current setting.

6.4.7. Miscellaneous Parameter: Strict Arguments

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

Macro (Parameter)	Description
`DMI32_MISC_STRICT_ARGUMENTS`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_MISC_STRICT_ARGUMENTS_DEFAULT`	This is the default, which is lenient processing.
`DMI32_MISC_STRICT_ARGUMENTS_DISABLE`	This refers to lenient processing.
`DMI32_MISC_STRICT_ARGUMENTS_ENABLE`	This refers to strict processing.

Macro (Services)	Description
`DMI32_MISC_STRICT_ARGUMENTS__GET(h,g)`	This requests the current setting.
`DMI32_MISC_STRICT_ARGUMENTS__NO(h)`	This requests lenient processing.
`DMI32_MISC_STRICT_ARGUMENTS__RESET(h)`	This requests a setting change to the default.
`DMI32_MISC_STRICT_ARGUMENTS__SET(h,s)`	This requests a setting change.
`DMI32_MISC_STRICT_ARGUMENTS__YES(h)`	This requests strict processing.

6.4.8. Miscellaneous Parameter: Strict Configuration

The purpose of this parameter is to control and retrieve the setting that governs the API's handling of certain invalid configurations. For example, if the setting supplied when adjusting a parameter is inconsistent with the board's current configuration, then the API can either respond with an error condition or try to proceed gracefully. If Strict Configuration processing is enabled, then processing is terminated with an error status. Otherwise the API is lenient and will try to proceed gracefully. This policy is applicable to parameter processing only, but is NOT applicable to the DMI32 at this time. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_MISC_STRICT_CONFIG`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_MISC_STRICT_CONFIG_DEFAULT`	This is the default, which is lenient processing.
`DMI32_MISC_STRICT_CONFIG_DISABLE`	This refers to lenient processing.
`DMI32_MISC_STRICT_CONFIG_ENABLE`	This refers to strict processing.

Macro	Description
`DMI32_MISC_STRICT_CONFIG__GET(h,g)`	This requests the current setting.
`DMI32_MISC_STRICT_CONFIG__NO(h)`	This requests lenient processing.
`DMI32_MISC_STRICT_CONFIG__RESET(h)`	This requests that the default be selected.
`DMI32_MISC_STRICT_CONFIG__SET(h,s)`	This requests a setting change.
`DMI32_MISC_STRICT_CONFIG__YES(h)`	This requests strict processing.

6.4.9. Miscellaneous Parameter: Strict DRAM Limits

The purpose of this parameter is to control and retrieve the setting restricting an application's DRAM addressing to what is installed on the DMI32. When enable the API restricts I/O accesses and relevant I/O Parameter and Transfer Parameter change requests so that they fall entirely within the installed memory. While enabled and the API is configured to be lenient, the default, the API will quietly adjust requests so that they fall within installed memory. If disabled, then applications are free to access any addressable region without limits. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_MISC_STRICT_DRAM_LIMITS`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_MISC_STRICT_DRAM_LIMITS_DEFAULT`	This is the default, which is enabled.
`DMI32_MISC_STRICT_DRAM_LIMITS_DISABLE`	This refers to the limits being ignored.

`DMI32_MISC_STRICT_DRAM_LIMITS_ENABLE`	This refers to the limits being observed.

Macro (Services)	**Description**
`DMI32_MISC_STRICT_DRAM_LIMITS__GET(h,g)`	This requests the current setting.
`DMI32_MISC_STRICT_DRAM_LIMITS__NO(h)`	This requests that the limits be ignored.
`DMI32_MISC_STRICT_DRAM_LIMITS__RESET(h)`	This requests a setting change to the default.
`DMI32_MISC_STRICT_DRAM_LIMITS__SET(h,s)`	This requests a setting change.
`DMI32_MISC_STRICT_DRAM_LIMITS__YES(h)`	This requests that the limits be observed.

6.5. Transfer Parameters

The purpose of the Transfer Parameters is to permit access to and control of those parameters that pertain exclusively to the DMI32's cable based data transfer feature. All Transfer Parameters are put in a default state when the device is opened and are returned to that state via the `dmi32_init()` service. The configuration of some of these parameters is retained within the DMI32 firmware registers. Those maintained in DMI32 firmware registers are put in a default state via the `dmi32_reset()` service. Applications have access to the DMI32 registers but it is advised that these features be accessed only through the Transfer Parameter services. When using the services `dmi32_config()` the "which" bits argument is ignored. The following table summarizes the Transfer Parameters.

Parameter Macros	**Description**
`DMI32_XFER_CLKIN_TEST`	This refers to the board's Clock In Test feature.
`DMI32_XFER_DIRECTION`	This refers to the data transfer direction across the cable interface.
`DMI32_XFER_ENABLE`	This refers to data transfer enable/disable setting.
`DMI32_XFER_INTERNAL_CLOCK_SPEED`	This refers to the board's Internal Clock Speed feature.
`DMI32_XFER_MODE`	This refers to the data transfer mode (Continuous or Single Shot).
`DMI32_XFER_OFFSET`	This refers to off offset of the next data transfer operation.
`DMI32_XFER_SIZE`	This refers to the size of the next data transfer operation.
`DMI32_XFER_START_WORD`	This refers to Start Word value for the next data transfer operation.
`DMI32_XFER_START_WORD_DIR`	This refers to the Start Word transmit direction for the next data transfer operation.
`DMI32_XFER_STATE`	This refers to current data transfer state (Active or Inactive).
`DMI32_XFER_STATE_RX`	This refers to current data transfer receive state (Active or Inactive).
`DMI32_XFER_STATE_TX`	This refers to current data transfer transmit state (Active or Inactive).
`DMI32_XFER_TEST_MODE`	This refers to the board's data transfer Test Mode feature.

6.5.1. Transfer Parameter: CLKIN Test

The purpose of this parameter is to check the clock on the cable's CLKIN signal. When requested, the API will wait for the check to complete and then return. If completed successfully, the results reported is the clock rate in megahertz. If unsuccessful the results indicate either that a clock was not detected or that it did not complete. The following tables describe the macros associated with this parameter.

Macro (Parameter)	**Description**
`DMI32_XFER_CLKIN_TEST`	This is the identifier for this parameter.

Macro (Set Values)	**Description**
`DMI32_XFER_CLKIN_TEST_DEFAULT`	This is the default, which is to do nothing.
`DMI32_XFER_CLKIN_TEST_NO`	This reflects a request to do nothing.

`DMI32_XFER_CLKIN_TEST_YES`	This reflects a request to perform a check.

Macro (Get Values)	Description
`DMI32_XFER_CLKIN_TEST_BUSY`	The test did not complete. This should never be reported.
`DMI32_XFER_CLKIN_TEST_NONE`	No clock was detected.
Otherwise …	The detected clock rate in megahertz.

Macro (Services)	Description
`DMI32_XFER_CLKIN_TEST__GET(h,g)`	This requests the current recorded results.
`DMI32_XFER_CLKIN_TEST__SET(h,s)`	This makes a request.
`DMI32_XFER_CLKIN_TEST__YES(h,g)`	This requests a test and the results.

6.5.2. Transfer Parameter: Direction

The purpose of this parameter is to control and retrieve the setting for the cable's data transfer direction. The following tables describe the macros associated with this parameter.

NOTE: This parameter cannot be set while a cable transfer is in progress.

Macro (Parameter)	Description
`DMI32_XFER_DIRECTION`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_XFER_DIRECTION_DEFAULT`	This is the default, which is to receive.
`DMI32_XFER_DIRECTION_RX`	This reflects the receive data direction.
`DMI32_XFER_DIRECTION_TX`	This reflects the transmit data direction.

Macro (Services)	Description
`DMI32_XFER_DIRECTION__GET(h,g)`	This requests the current setting.
`DMI32_XFER_DIRECTION__RESET(h)`	This changes the current setting to the default.
`DMI32_XFER_DIRECTION__RX(h)`	This requests the receive direction.
`DMI32_XFER_DIRECTION__SET(h,s)`	This changes the current setting.
`DMI32_XFER_DIRECTION__TX(h)`	This requests the transmit direction.

6.5.3. Transfer Parameter: Enable

The purpose of this parameter is to control and retrieve the board's enabling or inhibiting of the data transfer process. If enabled, the data transfer process is enabled. If the Start Word Direction is set to output, then the Start Word is sent and board starts transmitting or recording cable data, depending on the Transfer Direction. If the Start Word Direction is set to input, then the board waits for the Start Word to be received before it starts transmitting or recording cable data, depending on the Transfer Direction. If the parameter is disabled, data transfer ceases. Data transmission stops almost immediately. Data reception stops after a brief delay. The following tables describe the macros associated with this parameter.

NOTE: Applications must be aware that data transfer between the host and the DMI32's DRAM is not possible while cable based data transfer is in progress. I/O requests will fail while cable data transfer is active.

Macro (Parameter)	Description
`DMI32_XFER_ENABLE`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_XFER_ENABLE_DEFAULT`	This is the default, which is to inhibit data flow.

`DMI32_XFER_ENABLE_NO`	This disables data flow.
`DMI32_XFER_ENABLE_YES`	This enables data flow.

Macro (Services)	**Description**
`DMI32_XFER_ENABLE__GET(h,g)`	This requests the current setting.
`DMI32_XFER_ENABLE__NO(h)`	This request that the data stop flowing.
`DMI32_XFER_ENABLE__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_ENABLE__SET(h,s)`	This requests a setting change.
`DMI32_XFER_ENABLE__YES(h)`	This request that the data flow.

6.5.4. Transfer Parameter: Internal Clock Speed

The purpose of this parameter is to control and retrieve the board feature that divides the internal clock while in test mode. The following tables describe the macros associated with this parameter.

> **NOTE:** This parameter cannot be set while a cable transfer is in progress.

> **NOTE:** This parameter is applicable only when the Transfer Test Mode parameter is enabled.

Macro (Parameter)	**Description**
`DMI32_XFER_INTERNAL_CLOCK_SPEED`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_XFER_INTERNAL_CLOCK_SPEED_DEFAULT`	This is the default, which is the unaltered clock speed.
`DMI32_XFER_INTERNAL_CLOCK_SPEED_HIGH`	This reflects the unaltered clock speed.
`DMI32_XFER_INTERNAL_CLOCK_SPEED_LOW`	This reflects the divided clock speed.

Macro (Services)	**Description**
`DMI32_XFER_INTERNAL_CLOCK_SPEED__GET(h,g)`	This requests the current setting.
`DMI32_XFER_INTERNAL_CLOCK_SPEED__HIGH(h)`	This requests the unaltered clock speed.
`DMI32_XFER_INTERNAL_CLOCK_SPEED__LOW(h)`	This requests the divided clock speed.
`DMI32_XFER_INTERNAL_CLOCK_SPEED__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_INTERNAL_CLOCK_SPEED__SET(h,s)`	This requests a setting change.

6.5.5. Transfer Parameter: Mode

The purpose of this parameter is to control and retrieve the setting for the board's cable data transfer of a single DRAM buffer or of multiple DRAM buffers. The following tables describe the macros associated with this parameter.

> **NOTE:** This parameter cannot be set while a cable transfer is in progress.

> **NOTE:** When using the DMI32 in Multi-Buffer mode, applications are responsible for updating the Transfer Offset parameter if the DMI32 is to switch from one Transfer Buffer to another. Otherwise the DMI32 will continue to transfer the same block of data. Applications can use the Buffer Start interrupt as a queue to update the Transfer Offset parameter to the next location.

Macro (Parameter)	**Description**
`DMI32_XFER_MODE`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_XFER_MODE_DEFAULT`	This is the default, which is the single buffer mode.
`DMI32_XFER_MODE_SINGLE`	This reflects the single buffer mode.

`DMI32_XFER_MODE_MULTI`	This reflects the multi-buffer mode.

Macro (Services)	**Description**
`DMI32_XFER_MODE__GET(h,g)`	This requests the current setting.
`DMI32_XFER_MODE__MULTI(h)`	This requests the multi-buffer mode.
`DMI32_XFER_MODE__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_MODE__SET(h,s)`	This requests a setting change.
`DMI32_XFER_MODE__SINGLE(h)`	This requests the single buffer mode.

6.5.6. Transfer Parameter: Offset

The purpose of this parameter is to modify and report the starting offset within DRAM used by subsequent cable based data transfers. Offsets must be made to four byte boundaries and must not extend past the end of the DMI32's DRAM. The following tables describe the macros associated with this parameter.

> **NOTE:** When using the DMI32 in Multi-Buffer mode, applications are responsible for updating the Transfer Offset parameter if the DMI32 is to switch from one Transfer Buffer to another. Otherwise the DMI32 will continue to transfer the same block of data. Applications can use the Buffer Start interrupt as a queue to update the Transfer Offset parameter to the next location.

Macro (Parameter)	**Description**
`DMI32_XFER_OFFSET`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_XFER_OFFSET_DEFAULT`	This is the default, which is zero.
`DMI32_XFER_OFFSET_MASK`	This is the set of bits that may be set within an offset value.
`DMI32_XFER_OFFSET_MAX`	This is the maximum offset.
`DMI32_XFER_OFFSET_MIN`	This is the minimum offset.

Macro (Services)	**Description**
`DMI32_XFER_OFFSET__GET(h,g)`	This retrieves the current setting.
`DMI32_XFER_OFFSET__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_OFFSET__SET(h,s)`	This requests a setting change.

6.5.7. Transfer Parameter: Size

The purpose of this parameter is to control and retrieve the number of bytes transferred across the cable in the next data transfer operation. This parameter operates by accessing the board's Transfer Buffer Size Register. The size must be in four byte increments and must not cause the transfer to extend beyond the end of memory. The following tables describe the macros associated with this parameter.

> **NOTE:** The 4GB macro has a value of zero, which is a special case for the DMI32 firmware. This macro (and the value zero) are given special attention in the API to insure that applications have full access to the board's capabilities.

Macro (Parameter)	**Description**
`DMI32_XFER_SIZE`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_XFER_SIZE_4GB`	This refers to 4GB.
`DMI32_XFER_SIZE_DEFAULT`	This is the default, which is the minimum.
`DMI32_XFER_SIZE_MASK`	This is the mask of bits that can be set in the value specified.
`DMI32_XFER_SIZE_MAX`	This is the maximum, which is 4GB - 4.
`DMI32_XFER_SIZE_MIN`	This is the minimum, which is four bytes.

Macro (Services)	Description
`DMI32_XFER_SIZE__GET(h,g)`	This requests the current setting.
`DMI32_XFER_SIZE__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_SIZE__SET(h,s)`	This requests a setting change.

6.5.8. Transfer Parameter: Start Word

The purpose of this parameter is to modify and report the Start Word command value sent across the User I/O cable signals as part of initiating data transfer. The following tables describe the macros associated with this parameter.

> **NOTE:** Applications are free to assign any 7-bit Start Word value desired. However, for the DMI32 to capture a Start Word sent across the cable, the D6 bit must be set. If the D6 bit is not set, then the DMI32 will not recognize the User I/O value as a Start Word.

Macro (Parameter)	Description
`DMI32_XFER_START_WORD`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_XFER_START_WORD_DEFAULT`	This is the default, which is the hardware's default after a reset.
`DMI32_XFER_START_WORD_MASK`	This is the set of bits that may be set within a Start Word value.

Macro (Services)	Description
`DMI32_XFER_START_WORD__GET(h,g)`	This retrieves the current setting.
`DMI32_XFER_START_WORD__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_START_WORD__SET(h,s)`	This requests a setting change.

6.5.9. Transfer Parameter: Start Word Direction

The purpose of this parameter is to modify and report the direction from which the Start Word will be posted on the User I/O cable signals as part of initiating data transfer. The following tables describe the macros associated with this parameter.

> **NOTE:** Some firmware versions do not support this feature.

Macro (Parameter)	Description
`DMI32_XFER_START_WORD_DIR`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_XFER_START_WORD_DIR_DEFAULT`	This is the default, which is the receive direction.
`DMI32_XFER_START_WORD_DIR_RX`	This refers to the receive direction.
`DMI32_XFER_START_WORD_DIR_TX`	This refers to the transmit direction.

Macro (Services)	Description
`DMI32_XFER_START_WORD_DIR__GET(h,g)`	This retrieves the current setting.
`DMI32_XFER_START_WORD_DIR__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_START_WORD_DIR__RX(h)`	This requests the receive direction.
`DMI32_XFER_START_WORD_DIR__SET(h,s)`	This requests a setting change.
`DMI32_XFER_START_WORD_DIR__TX(h)`	This requests the transmit direction.

6.5.10. Transfer Parameter: State

The purpose of this read-only parameter is to retrieve the data transfer state. This refers to the transfer of data across the cable and is independent of the data transfer direction. In essence, this reflects the enabled/disabled state of the transfer process. The following tables describe the macros associated with this parameter.

Macro (Parameter)	**Description**
`DMI32_XFER_STATE`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_XFER_STATE_ACTIVE`	This means that data is being transferred across the cable.
`DMI32_XFER_STATE_INACTIVE`	This means that data is not being transferred across the cable.

Macro (Services)	**Description**
`DMI32_XFER_STATE__GET(h,g)`	This requests the current state.

6.5.11. Transfer Parameter: State Rx

The purpose of this read-only parameter is to retrieve the data transfer state for data being received. This reflects the recording of receive data and indicates that the Start Word has been send or received. The following tables describe the macros associated with this parameter.

Macro (Parameter)	**Description**
`DMI32_XFER_STATE_RX`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_XFER_STATE_RX_ACTIVE`	This means that data is being recorded from the cable.
`DMI32_XFER_STATE_RX_INACTIVE`	This means that data is not being recorded from the cable.

Macro (Services)	**Description**
`DMI32_XFER_STATE_RX__GET(h,g)`	This requests the current state.

6.5.12. Transfer Parameter: State Tx

The purpose of this read-only parameter is to retrieve the data transfer state for data being transmitted. This reflects the transmission of DRAM data to the cable and indicates that the Start Word has been send or received. The following tables describe the macros associated with this parameter.

Macro (Parameter)	**Description**
`DMI32_XFER_STATE_TX`	This is the identifier for this parameter.

Macro (Values)	**Description**
`DMI32_XFER_STATE_TX_ACTIVE`	This means that data is being transmitted to the cable.
`DMI32_XFER_STATE_TX_INACTIVE`	This means that data is not being transmitted to the cable.

Macro (Services)	**Description**
`DMI32_XFER_STATE_TX__GET(h,g)`	This requests the current state.

6.5.13. Transfer Parameter: Test Mode

The purpose of this parameter is to control and retrieve the setting controlling the board's clock generation and routing configuration. When enabled, the cable's CLKIN and CLKOUT signals are reversed and the CLKOUT signal is derived from the DMI32's onboard oscillator. This is done to facilitate connecting two DMI32 boards back-to-back using a strait pass-through cable. The following tables describe the macros associated with this parameter.

NOTE: This parameter cannot be set while a cable transfer is in progress.

Macro (Parameter)	Description
`DMI32_XFER_TEST_MODE`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_XFER_TEST_MODE_DEFAULT`	This is the default, which is normal operation.
`DMI32_XFER_TEST_MODE_NO`	This reflects the normal operating mode.
`DMI32_XFER_TEST_MODE_YES`	This reflects the reversed clock signal mode.

Macro (Services)	Description
`DMI32_XFER_TEST_MODE__GET(h,g)`	This requests the current setting.
`DMI32_XFER_TEST_MODE__NO(h)`	This requests the normal mode.
`DMI32_XFER_TEST_MODE__RESET(h)`	This requests a setting change to the default.
`DMI32_XFER_TEST_MODE__SET(h,s)`	This requests a setting change.
`DMI32_XFER_TEST_MODE__YES(h)`	This requests the reversed clock signal mode.

6.6. User I/O Parameters

The purpose of the User I/O Parameters is to permit access to and control of the board's two User I/O data ports. All User I/O Parameters are put in a default state when the device is opened and are returned to that state via the `dmi32_init()` and `dmi32_reset()` services. The configuration of these parameters is retained within the DMI32 firmware registers. Applications have access to the DMI32 registers but it is advised that these features be accessed only through the User I/O Parameter services. When using the service `dmi32_config()` the "which" bits are used to refer to the port(s) of interest. In additions, some of the parameters may be referenced by which is currently configured as an input or output. The following table summarizes the User I/O Parameters.

Parameter Macros	Description
`DMI32_USER_IO_DIRECTION`	This refers to the User I/O direction.
`DMI32_USER_IO_INPUT`	This refers to the User I/O input value.
`DMI32_USER_IO_OUTPUT`	This refers to the User I/O output value.

6.6.1. User I/O Parameter: Direction

The purpose of this parameter is to control and retrieve the port's data transfer direction. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_USER_IO_DIRECTION`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_USER_IO_DIRECTION_DEFAULT`	This is the default, which is input.
`DMI32_USER_IO_DIRECTION_IN`	This refers to the input direction.
`DMI32_USER_IO_DIRECTION_OUT`	This refers to the output direction.

Macro (Services)	Description
`DMI32_USER_IO_DIRECTION__GET(h,w,g)`	This requests a current setting.
`DMI32_USER_IO_DIRECTION__SET(h,w,s)`	This requests a setting change.
`DMI32_USER_IO_DIRECTION__IN(h,w)`	This requests a setting change to input.
`DMI32_USER_IO_DIRECTION__OUT(h,w)`	This requests a setting change to output.
`DMI32_USER_IO_DIRECTION__RESET(h,w)`	This requests a setting change to the default.
`DMI32_USER_IO_DIRECTION__`*`X`*`_GET(h,g)`	This requests the setting for port *X*. *
`DMI32_USER_IO_DIRECTION__`*`X`*`_IN(h)`	This requests that port *X* be configured as an input *

`DMI32_USER_IO_DIRECTION__X_OUT(h)`	This requests that port *X* be configured as an output *
`DMI32_USER_IO_DIRECTION__X_RESET(h)`	This requests a setting change for port *X* to the default. *
`DMI32_USER_IO_DIRECTION__X_SET(h,s)`	This requests a setting change for port *X*. *

* The "*X*" refers individually to A and B.

6.6.2. User I/O Parameter: Input

The purpose of this read-only parameter is to retrieve a port's current input value, which is the value currently on the cable. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_USER_IO_INPUT`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_USER_IO_INPUT_MASK`	This is the set of bits that may be set when reading an input port.

Macro (Services)	Description
`DMI32_USER_IO_INPUT__A_GET(h,g)`	This request the current input from port A.
`DMI32_USER_IO_INPUT__B_GET(h,g)`	This request the current input from port B.
`DMI32_USER_IO_INPUT__CUR_GET(h,g)`	This request the input from that port, A or B, currently functioning as an input. If neither is functioning as an input then neither is read.
`DMI32_USER_IO_INPUT__GET(h,w,g)`	This requests a current input.

6.6.3. User I/O Parameter: Output

The purpose of this parameter is to update and retrieve a port's current output value, which is the value the port will put on the cable when functioning as an output. The following tables describe the macros associated with this parameter.

Macro (Parameter)	Description
`DMI32_USER_IO_OUTPUT`	This is the identifier for this parameter.

Macro (Values)	Description
`DMI32_USER_IO_OUTPUT_DEFAULT`	This is the default output value, which is zero.
`DMI32_USER_IO_OUTPUT_MASK`	This is the set of bits that may be set when reading an input port.

Macro (Services)	Description
`DMI32_USER_IO_OUTPUT__CUR_GET(h,g)`	This request the value for that port, A or B, currently functioning as an output. If neither is functioning as an output then neither is read.
`DMI32_USER_IO_OUTPUT__CUR_RESET(h)`	This request a value update for that port, A or B, currently functioning as an output to the default output value. If neither is functioning as an output then neither is written.
`DMI32_USER_IO_OUTPUT__CUR_SET(h,s)`	This request a value update for that port, A or B, currently functioning as an output. If neither is functioning as an output then neither is written.
`DMI32_USER_IO_OUTPUT__GET(h,w,g)`	This requests a port's current value.
`DMI32_USER_IO_OUTPUT__RESET(h,w)`	This requests that a port's current output be set to the default.
`DMI32_USER_IO_OUTPUT__SET(h,w,s)`	This requests a change to a port's current value.
`DMI32_USER_IO_OUTPUT__X_GET(h,g)`	This request the current value for port *X*.
`DMI32_USER_IO_OUTPUT__X_RESET(h)`	This request a value change for port *X* to the default.

`DMI32_USER_IO_OUTPUT__X_SET(h,s)`	This request a value change for port *X*.

* The "*X*" refers individually to A and B.

7. Operation

This section explains some operational procedures using the DMI32 with the SDK. This is in no way intended to be a comprehensive guide on using the DMI32. This is simply to address a very few issues relating to the board's use.

7.1. Overview

Before accessing any DMI32, applications first verify the state of the API (by calling `dmi32_api_status()`). Once done, access is permissible to any and all installed boards (by calling `dmi32_open()`). An open request returns a handle to the device which is specific to and usable only by the API. (The API prevents access to the device other than by this handle.) Using this handle, applications have full access to and control over the DMI32 and its features. Applications can configure all available device and API settings, perform I/O and data transfer operations and when done release access to the device (by calling `dmi32_close()`).

7.2. Data Reception

7.2.1. Overview

Data reception is a relatively simple process. The following steps provide an overview of the kind of actions needed.

1. Return the API and the device to a known state by calling `dmi32_init()`. This places the API and the DMI32 in the same state it was in when first opened.
2. Configure the I/O Parameters, which can be done using the many `DMI32_IO_XXX()` macros.
3. Configure the Interrupt Parameters, which can be done using the many `DMI32_IRQ_XXX()` macros.
4. Configure the Transfer Parameters, which can be done using the many `DMI32_XFER_XXX()` macros.
5. Configure the Miscellaneous Parameters, which can be done using the many `DMI32_MISC_XXX()` macros.
6. Initiate data transfer in from the cable interface and read the data from the board's DRAM.

7.2.2. Details

The following is a specific set of actions an application may make in configuring a board for data reception. These steps are focused primarily on the task of getting data into the DMI32 and may vary significantly depending on the application's needs.

1. Return the API and the device to a known state by calling `dmi32_init()`. This places the API and the DMI32 in the same state it was in when first opened.
2. Configure the board's Test Mode, which should normally be disabled. Refer to the `DMI32_XFER_TEST_MODE__XXX()` macros. If Test Mode is not needed, then this step can be ignored.
3. If Test Mode is enabled, then configure the Transfer Internal Clock Speed parameter. Refer to the `DMI32_XFER_INTERNAL_CLOCK_SPEED__XXX()` macros. If Test Mode is not needed, then this step can be ignored.
4. Configure the Transfer Mode parameter. Refer to the `DMI32_XFER_MODE__XXX()` macros. If using the Multi-Buffer mode, then additional work will be required as data transfers begins in order to define additional Transfer Buffers.

5. Configure the Transfer Direction parameter. Refer to the `DMI32_XFER_DIRECTION__RX()` macro.
6. Configure the Start Word parameter. Refer to the `DMI32_XFER_START_WORD__XXX()` macros.
7. Configure the Start Word Direction parameter. Refer to the `DMI32_XFER_START_WORD_DIR__XXX()` macros.
8. Define the Transfer Buffer using the `dmi32_xfer_buf_set()` service. If using the Multi-Buffer mode, then this step will have to be repeated as data transfers begin in order to define additional Transfer Buffers. This service serves the same purpose as defining the Transfer Buffer using the `DMI32_XFER_OFFSET__XXX()` and `DMI32_XFER_SIZE__XXX()` macros.
9. Configure the User I/O Direction parameter for both User I/O ports. Refer to the `DMI32_USER_IO_DIRECTION__XXX()` macros.
10. Enable the data reception process by exercising the `DMI32_XFER_ENABLE__YES()` macro.
11. If in Single Buffer mode simply wait for the reception process to complete before proceeding. If in Multi-Buffer mode, update the Transfer Offset and Transfer Size parameters at the appropriate points in time in order to record data in additional Transfer Buffers. When finished, either exercise the `DMI32_XFER_ENABLE__NO()` macro to abruptly stop data reception (after a short delay) or switch to Single Buffer mode to gracefully halt reception at the end of the current buffer.
12. When data reception has ended, proceed as follows.
13. Configure the I/O Offset parameter. Refer to the `dmi32_seek()` service, and its related macros, or the `DMI32_IO_OFFSET__XXX()` macros.
14. Exercise the `dmi32_read()` service to retrieve data recorded by the DMI32.

7.3. Data Transmission

7.3.1. Overview

Data transmission is a relatively simple process. The following steps provide an overview of the kind of actions needed.

1. Return the API and the device to a known state by calling `dmi32_init()`. This places the API and the DMI32 in the same state it was in when first opened.
2. Configure the I/O Parameters, which can be done using the many `DMI32_IO_XXX()` macros.
3. Configure the Interrupt Parameters, which can be done using the many `DMI32_IRQ_XXX()` macros.
4. Configure the Transfer Parameters, which can be done using the many `DMI32_XFER_XXX()` macros.
5. Configure the Miscellaneous Parameters, which can be done using the many `DMI32_MISC_XXX()` macros.
6. Write data into the board’s DRAM and initiate data transfer out the cable interface.

7.3.2. Details

The following is a specific set of actions an application may make in configuring a board for data transmission. These steps are focused primarily on the task of getting data out of the DMI32 and may vary significantly depending on the application's needs.

1. Return the API and the device to a known state by calling `dmi32_init()`. This places the API and the DMI32 in the same state it was in when first opened.
2. Configure the I/O Offset parameter. Refer to the `dmi32_seek()` service, and its related macros, or the `DMI32_IO_OFFSET__XXX()` macros.
3. Exercise the `dmi32_write()` service to initialize the DMI32's DRAM with the data image to be transmitted.
4. Configure the board's Test Mode, which should normally be disabled. Refer to the `DMI32_XFER_TEST_MODE__XXX()` macros. If Test Mode is not needed, then this step can be ignored.
5. If Test Mode is enabled, then configure the Transfer Internal Clock Speed parameter. Refer to the `DMI32_XFER_INTERNAL_CLOCK_SPEED__XXX()` macros. If Test Mode is not needed, then this step can be ignored.
6. Configure the Transfer Mode parameter. Refer to the `DMI32_XFER_MODE__XXX()` macros. If using the Multi-Buffer mode, then additional work will be required as data transfers begins in order to define additional Transfer Buffers.
7. Configure the Transfer Direction parameter. Refer to the `DMI32_XFER_DIRECTION__TX()` macro.
8. Configure the Start Word parameter. Refer to the `DMI32_XFER_START_WORD__XXX()` macros.
9. Configure the Start Word Direction parameter. Refer to the `DMI32_XFER_START_WORD_DIR__XXX()` macros.
10. Define the Transfer Buffer using the `dmi32_xfer_buf_set()` service. If using the Multi-Buffer mode, then this step will have to be repeated as data transfer begins in order to define additional Transfer Buffers. This service performs the same purpose as defining the Transfer Buffer using the `DMI32_XFER_OFFSET__XXX()` and `DMI32_XFER_SIZE__XXX()` macros.
11. Configure the User I/O Direction parameter for both User I/O ports. Refer to the `DMI32_USER_IO_DIRECTION__XXX()` macros.
12. Enable the data transmit process by exercising the `DMI32_XFER_ENABLE__YES()` macro.
13. If in Single Buffer mode simply wait for the transmission process to complete before proceeding. If in Multi-Buffer mode, update the Transfer Offset and Transfer Size parameters at the appropriate points in time in order to transmit data from additional Transfer Buffers. When finished, either exercise the `DMI32_XFER_ENABLE__NO()` macro to abruptly stop data transmission or switch to Single Buffer mode to gracefully halt transmission at the end of the current buffer.

7.4. Data Transfer Issues

7.4.1. I/O Abort Requests

The API includes the feature of aborting I/O operations. One issue with requesting an abort is that overlapped I/O operations occur in the background with threads having a priority of `THREAD_PRIORITY_HIGHEST`. This means that the I/O operation may have a higher priority than the requesting thread, resulting in the I/O requests completing before the API is able to register the abort request.

7.4.2. I/O Data Buffers

The API Library supports the use of Application Buffers (application allocated buffers) and API Buffers (the API's internally allocated buffers). Each has plusses and minuses and both can be used by `dmi32_read()` and `dmi32_write()` interchangeably. Application Buffers are entirely under application control and are obtained by `malloc()` and similar services. This permits an application to have any number of buffers of most any desired size, and can even exceed the size of physical memory. The drawback they have though is that allocations only appear to be contiguous, when, in fact, they are actually scattered throughout physical memory. In addition, they can be paged out to the hard disk as needed by the OS. A result of this is additional overhead when performing I/O. API Buffers, on the other hand, avoid this inefficiency because they occupy physically contiguous and immovable memory regions, and therefore require less overhead during I/O requests. The disadvantage though, is that these may be smaller than desired and the API supports only two. (While each of the two API Buffers is associated with a particular I/O data direction, both can be used interchangeably at will.)

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

API Buffers are accessed via the I/O Buffer Size and I/O Buffer Pointer parameters (`DMI32_IO_BUFFER_SIZE` and `DMI32_IO_BUFFER_POINTER`, respectively). Each buffer size starts out at zero (0) and the pointer as NULL. Application must first use the I/O Buffer Size parameter in order to make an allocation request. Since the resources for these memory regions are much more limited than for `malloc()` type requests, the size of the allocation obtained may be smaller than asked for. After a size request use the I/O Buffer Pointer parameter to get a pointer to the memory obtained. Each attempt to alter the size demands that the application update its pointer. Failure to do so is likely to produce a protection fault. When finished, setting the size to zero (0) frees the buffer.

7.4.3. I/O Timeout

In general the timeout settings should be made so that they expire only when something has gone wrong (see exception below). This is not critical with PIO, but it is with DMA. With PIO transfers, a timeout has no consequence except to cause the API to transfer no additional data. In this case no data is lost and an exact accounting of the amount of data transferred is accurately maintained. With DMA transfers, a timeout results in the DMA engine aborting the transfer midstream. For the DMI32 this means that the amount of data that was successfully transferred in that request is unknown. With DMA transfers, since they tend to complete very quickly, I/O timeouts are less likely. If a DMA operation times out, the amount of data transferred in that request will be unknown.

The exception to the above guidelines is with a timeout of zero (0). With the DMI32 the effect of a zero timeout means to perform the transfer and return when it is done, effectively ignoring the minimal time limit.

7.4.4. Blocking/Overlapped I/O

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

7.4.5. I/O Modes

The API Library offers two data transfer modes. Each has its pros and cons, which are described briefly below.

Mode	Description	
PIO	This mode uses repetitive register accesses to perform transfers.	
	Pros:	It is the most reliable mode offered. It is well suited for any size I/O request. This mode should never return a failure status for valid requests.
	Cons:	It is inefficient.
DMA	This mode uses non-Demand Mode DMA, which transfers data without processor intervention.	
	Pros:	This is the most efficient means provided. See note below.
	Cons:	This method becomes inefficient with smaller sized transfers. If an I/O timeout is encountered, the amount of data may be more than the amount reported. See note below. This mode could return a failure status, depending on system resources.

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

7.4.6. DMA Based I/O Requests

The two DMA engines on the DMI32 are each limited to transfers of 8,388,607 bytes. That is one byte shy of 8-megabytes. For the DMI32 that translates to a transfer limit of 8,388,604 bytes, or 2,097,151 32-bit values, which is one shy of 2-megasamples. The API breaks all DMA requests into smaller requests of at most (8M - 4) bytes each. So, if an application made a request for 8MB, the API would break that into one request for (8M - 4) bytes and another request for four bytes. Application should therefore consider making DMA requests smaller than, or a multiple of 8,388,604 bytes.

7.5. Event Notification

The API Library supports event notification for two sources or types of events. They include Interrupt Notification and I/O Completion Notification and operate independently. Notification for both sources includes both a callback mechanism and a wait mechanism. All are described below. Interrupt Notification is driven by interrupts generated by the DMI32 from any of the interrupt sources identified in the Interrupt Control Register (`DMI32_ICR`). I/O Completion Notification is associated with completed I/O requests. This applies to both blocking and overlapped I/O, and generally occurs no matter the outcome of the I/O request (i.e. successful transfer or not).

7.5.1. Event Callback

Using the callback mechanism each notification source can be assigned a callback function. Each source can have a single callback with an application specific value passed as an argument. Callbacks can be assigned to any source and in any combination desired. If a given callback is associated with multiple sources, then multiple callbacks will be made as the different events occur. So, for example, if a single callback is assigned to two different interrupts, then the callback function will be called separately for each interrupt, as often as each occurs. Since each source is associated with its own callback context, a thread context, such callbacks must support multithreaded operation. Applications are free to reconfigure callbacks during a callback context, but the callback for a given event must return before subsequent callback notification can occur for that same event. The prototype required for all callbacks is the data type `dmi32_callback_func_t`. The three arguments to the callback are each `U32` data types. Application must cast the values given to their respective types, which are described below.

7.5.1.1. Interrupt Notification Callback

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

Argument	Cast	Description
`arg1`	`void*`	This is the device handle received from `dmi32_open()`.
`arg2`	`U32`	This is the specific “which” bit for the interrupt that produced the callback. Refer to the `DMI32_WHICH_IRQ_XXX` macros.
`arg3`	`U32`	This is an application specific argument. This is the Interrupt Callback Argument parameter.

7.5.1.2. I/O Completion Notification Callback

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

Argument	Cast	Description
`arg1`	`void*`	This is the device handle received from `dmi32_open()`.
`arg2`	`U32`	This is the applicable I/O status data. Refer to the `GSC_IO_STATUS_XXX` macros.
`arg3`	`U32`	This is an application specific argument. This is the I/O Callback Argument parameter.

7.5.2. Event Waiting

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

7.6. Clock Checks

The API includes the Transfer CLKIN Test parameter to request that a check be performed on the board’s input clock. This parameter waits for the check to complete before returning and returns the detected clock rate. The below example illustrates a possible use of this parameter.

Example

```
#include <stdio.h>

#include "dmi32_api.h"
```

```
#include "dmi32_dsl.h"

U32 dmi32_dsl_clock_check(void* handle, U32* mhz, int verbose)
{
    U32             status;
    unsigned long   value;

    status  = DMI32_XFER_CLKIN_TEST__YES(handle, &value);
    mhz[0]  = (U32) value;

    if (!verbose)
    {
    }
    else if (status != GSC_SUCCESS)
    {
        mhz[0]  = 0;
        printf("Transfer CLKIN Test failure: %ld\n", (long) status);
    }
    else if (mhz[0] == DMI32_XFER_CLKIN_TEST_NONE)
    {
        mhz[0]  = 0;
        printf("No clock detected.\n");
    }
    else if (mhz[0] == DMI32_XFER_CLKIN_TEST_BUSY)
    {
        mhz[0]  = 0;
        printf("Clock test did not complete.\n");
    }
    else
    {
        printf("Clock Rate: %ldMHz\n", (long) mhz[0]);
    }

    return(status);
}
```

Document History

Revision	Description
October 29, 2007	Updated for SDK port to Linux, which is SDK version 6.0.0.
August 18, 2005	Initial release.