

ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Engineering Design Team PCI 16D* at our website: ***Click HERE***

PCI 16D

User's Guide

Revision: A
February 2004

Control Information

Control Item	Details
Document Owner	Mark Mason
Information Label	EDT Public
Supersedes	None
File Location	frm:/pci16d/p16d.doc
Document Number	008-00969-00

Revision History

Revision	Date	Revision Description	Originator
Draft	16-Feb-04	Convert from FrameMaker to Word; update pciload section	S Vasil

The information in this document is subject to change without notice and does not represent a commitment on the part of Engineering Design Team, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

Engineering Design Team, Inc. (“EDT”), makes no warranties, express or implied, including without limitation the implied warranties of merchantibility and fitness for a particular purpose, regarding the software described in this document (“the software”). EDT does not warrant, guarantee, or make any representations regarding the use or the results of the use of the software in terms of its correctness, accuracy, reliability, currentness, or otherwise. The entire risk as to the results and performance of the software is assumed by you. The exclusion of implied warranties is not permitted by some jurisdictions. The above exclusion may not apply to you.

In no event will EDT, its directors, officers, employees, or agents be liable to you for any consequential, incidental, or indirect damages (including damages for loss of business profits, business interruption, loss of business information, and the like) arising out of the use or inability to use the software even if EDT has been advised of the possibility of such damages. Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitations may not apply to you. EDT’s liability to you for actual damages for any cause whatsoever, and regardless of the form of the action (whether in contract, tort [including negligence], product liability or otherwise), will be limited to $50.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, without the express written agreement of Engineering Design Team, Inc.

© Copyright Engineering Design Team, Inc. 1997–2004. All rights reserved.

Sun, SunOS, SBus, SPARC, and SPARCstation are trademarks of Sun Microsystems, Incorporated.

Windows NT/2000/XP is a registered trademark of Microsoft Corporation.

Intel and Pentium are registered trademarks of Intel Corporation.

UNIX is a registered trademark of X/Open Company, Ltd.

OPEN LOOK is a registered trademark of UNIX System Laboratories, Inc.

Red Hat is a trademark of Red Hat Software, Inc.

IRex is a trademark of Silicon Graphics, Inc.

AIX is a registered trademark of International Business Machines Corporation.

Xilinx is a registered trademark of Xilinx, Inc.

Kodak is a trademark of Eastman Kodak Company.

The software described in this manual is based in part on the work of the independent JPEG Group.

EDT and Engineering Design Team are trademarks of Engineering Design Team, Inc.

Contents

Overview **5**

After Installing **6**

Testing 6
Building the Sample Programs 7
Uninstalling 7
Upgrading the Firmware 8

Real-time Input and Output **9**

Elements of EDT Interface Applications 9
DMA Library Routines 11
EDT Message Handler Library 45
Message Definitions 45
Files 46

Hardware **53**

PCI Local Bus Interface 53
FIFO 53
Device Interface 53
Logic Levels 53

Signals **55**

Handshake Signals 57
Data Signals 57
Synchronous Control Signals 57
Asynchronous Control Signals 58
Timing 58
Using the Signal for Data I/O 59

Registers **60**

Configuration Space 60
PCI Local Bus Addresses 61
Scatter-gather DMA 62
Performing DMA 63
Flash ROM Access Registers 68
Device Control Registers 69

Specifications **74**

References **75**

Overview

The PCI 16D is a single-slot, 16-bit parallel input/output interface for PCI Bus-based computer systems. It is designed for continuous input or output between a user device and PCI Bus host memory. The PCI 16D features 128 bytes of FIFO storage in each direction and can support continuous data rates of up to 10 MB per second. The board also includes diagnostic capability.

The PCI 16D supports scatter-gather Direct Memory Access (DMA) in hardware, adapting to the memory management model of the host architecture. It includes a software driver and software library, enabling applications to access the PCI 16D and transfer data continuously or in bursts across the PCI 16D interface using standard library calls.

The PCI 16D provides 16 data inputs and 16 data outputs for direct memory access (DMA). The interface also provides three status inputs, three control outputs and a device interrupt. If you use the PCI 16D DMA input, the 16 data outputs are available for additional control signals. If you use the PCI 16D for DMA output, you can use the 16 data inputs for additional device status inputs.

The PCI 16D uses a simple clock/acknowledge protocol for the DMA data. Handshake polarity is programmable.

This manual describes the operation of the PCI 16D with UNIX-based and Windows operating systems.

After Installing

After installing the PCI 16D, test the board and build the sample programs, if you wish. Instructions for uninstalling the software and upgrading the firmware are also provided if necessary.

Testing

The PCI 16D is tested at the factory with a device that reads and writes the interface based on the state of the DIR (direction) line.

You can perform diagnostics on the PCI 16D by installing an optional loopback connector and executing the `looptest.c` diagnostic program included with the standard PCI 16D software. The diagnostic program requires that the PCI 16D driver be installed.

To test the PCI 16D, loop output data from the host back panel or the end of the device cable back to the input. The optional EDT PCI 16D Loopback Kit, part number 012-00067, provides test connectors for both loopback configurations. The loopback kit contains complete instructions for the PCI 16D loopback diagnostics.

The diagnostic program `looptest.c`, used in conjunction with the loopback test connectors, performs the tests described in the table below. To install the program, at the command prompt, enter:

```
make looptest
```

Use `looptest` with the following command-line arguments:

```
looptest [-u n] [-e]
```

`-u n` The device number of the board to test: use 0 for the first PCI 16D board, 1 for the second, etc. The default device is 0.

`-e` Prints first miscompared word only.

Test Name	Test Operation Performed
Programmed I/O Loopback Data Test	Write and read walking ones and zeroes (with each bit uniquely high and low), then read each word and compare the expected values with the actual values.
Block Loopback	Write and read walking ones and zeroes (with each bit uniquely high and low) using DMA, then read each word and compare the expected and actual values.
Function/Status Bits	Output all function bits, and compare the reported function bits in the Status Register with the looped-back status bits.
DINT signal	Program the driver to get a SIGIO signal on DINT, then pulse DIR (which loops back to ATTN), and check for occurrence of the SIGIO signal.
Error on DINT	Program the driver to give an EINTR error if DINT comes in between I/O requests; pulse DIR; then perform a read and check for an error return.
5-second read timeout	Program the driver to give a 5-second read timeout, then check for the duration and an ETIMEDOUT error return.

5-second write timeout	Program the driver to give a 5-second write timeout, then check for the duration and an ETIMEDOUT error return.

See the “Signals” section on page 55 and the “Registers” section on page 60 for more information.

Building the Sample Programs

UNIX-based Systems

To build any of the example programs on UNIX-based systems, cd to /opt/EDTp16d and enter the command:

```
make program name
```

where *file* is the name of the example program you wish to install.

To build and install all the example programs, enter the command:

```
make
```

Outcome: All example programs display a message that explains their usage when you enter their names without parameters.

Windows Systems

To build any of the example programs on Windows systems:

1. Run `pci11 Utilities`.
2. Enter the command:

   ```
   nmake program.exe
   ```

   where *file* is the name of the example program you wish to build.

To build and install all the example programs, simply enter the command:

```
nmake
```

Outcome: All example programs display a message that explains their usage when you enter their names without parameters.

Note: You can also build the sample programs by setting up a project in Windows Visual C++. Contact EDT for instructions.

Uninstalling

Solaris Systems

To remove the PCI 11W driver on Solaris systems:

1. Become root or superuser.
2. Enter:

   ```
   pkgrm EDTp16d
   ```

For further details, consult your operating system documentation, or call Engineering Design Team.

Linux Systems

To remove the PCI 11W driver on Linux systems, enter:

```
cd /opt/EDTp16d
```

```
make unload
cd /
rm –rf /opt/EDTp16d
```

Windows Systems

To remove the PCI 11W toolkit on Windows systems, use the Windows Add/Remove utility. For further details, consult your Windows documentation.

You can always get the most recent update of the software from our web site, **www.edt.com**. See the document titled *Contact Us*.

Upgrading the Firmware

Field upgrades to the PCI firmware may occasionally be necessary when upgrading to a new device driver.

The Xilinx file is downloaded to the board's PCI interface Xilinx PROM using the *pciload* program:

1. Navigate to the directory in which you installed the driver (for UNIX-based systems, usually */opt/EDTp16d*; for Windows, usually *C:\EDT\p16d*).
2. At the prompt, enter:

   ```
   pciload verify
   ```

   This will compare the current PCI Xilinx file in the package with what is currently on the board's PROM.

   Note: If more than one board is installed on a system, use the following, where N is the board unit number:

   ```
   pciload –u N verify
   ```

   Outcome: Dates and revision numbers of the PROM and File ID will be displayed. If these numbers match, there is no need for a field upgrade. If they differ, upgrade the flash PROM as follows:

   a. At the prompt, enter:

      ```
      pciload update
      ```

   b. Shut down the operating system and turn the host computer off and then back on again. The board reloads firmware from flash ROM only during power-up. Therefore, after running *pciload*, the new bit file is not in the Xilinx until the system has been power-cycled; simply rebooting is not adequate.

To just see what boards are in the system, run pciload without any arguments:

```
pciload
```

To see other pciload options, run:

```
pciload help
```

Real-time Input and Output

The PCI 16D device driver can perform two kinds of DMA transfers: continuous and noncontinuous.

To perform continuous transfers, use ring buffers. The ring buffers are a set of buffers that applications can access continuously, reading and writing as required. When the last buffer in the set has been accessed, the application then cycles back to the first buffer. See `edt_configure_ring_buffers` for a complete description of the ring buffer parameters that can be configured. See the sample programs `simple_getdata.c` and `simple_putdata.c` distributed with the driver for examples of using the ring buffers.

For noncontinuous transfers, the driver uses DMA system calls `read` and `write`. Each `read` and `write` system call performs a single, noncontinuous DMA transfer.

Note: For portability, use the library calls `edt_reg_read`, `edt_reg_write`, `edt_reg_or`, or `edt_reg_and` to read or write the hardware registers rather than ioctls.

Elements of EDT Interface Applications

Applications for performing continuous transfers typically include the following elements:

```
#include "edtinc.h"

main()
{
    EdtDev *edt_p = edt_open("pcd", 0) ;
    char *buf_ptr; int outfd = open("outfile", 1) ;

    /* Configure a ring buffer with four 1MB buffers */
    edt_configure_ring_buffers(edt_p, 1024*1024, 4, EDT_READ, NULL) ;

    /* start 4 buffers*/
    edt_start_buffers(edt_p, 4) ;

    /* This loop will capture data indefinitely, but the write()
     * (or whatever processing on the data) must be able to keep up.
    */
    while ((buf_ptr = edt_wait_for_buffers(edt_p, 1)) != NULL)
        write(outfd, buf_ptr, 1024*1024) ;
        edt_start_buffers(edt_p, 1) ;

    edt_close(edt_p) ;

}
```

Applications for performing noncontinuous transfers typically include the following elements. This example opens a specific DMA channel with `edt_open_channel`, assuming that a multi-channel Xilinx firmware file has been loaded:

```
#include "edtinc.h"

main()

{
    EdtDev *edt_p = edt_open_channel("pcd", 1, 2) ;
    char buf[1024] ;
    int numbytes, outfd = open("outfile", 1) ;
```

```
      /*
       * Because read()s are noncontinuous, unless is there hardware
       * handshaking there will be gaps in the data between each read().
       */
      while ((numbytes = edt_read(edt_p, buf, 1024)) > 0)
         write(outfd, buf, numbytes) ;
      edt_close(edt_p) ;

  }
```

You can use ring buffer mode for real-time data capture using a small number (typically 1 MB) of buffers configured in a round-robin data FIFO. During capture, the application must be able to transfer or process the data before data acquisition wraps around and overwrites the buffer currently being processed.

The example below shows real-time data capture using ring buffers, although it includes no error checking. In this example, `process_data(bufptr)` must execute in the same amount of time it takes DMA to fill a single buffer or faster.

```
#include "edtinc.h"

main()
{
   EdtDev *edt_p = edt_open("pcd", 0) ;

   /* Configure four 1 MB buffers:
    * one for DMA
    * one for the second DMA register on most EDT boards
    * one for "process_data(bufptr)" to work on
    * one to keep DMA away from "process_data()"
    */
   edt_configure_ring_buffers(edt_p, 1*1024*1024, 4, EDT_READ, NULL) ;

   edt_start_buffers(edt_p, 4) ; /* start 4 buffers */

   for (;;)
   {
      char *bufptr ;

      /* Wait for each buffer to complete, then process it.
       * The driver continues DMA concurrently with processing.
       */
      bufptr = edt_wait_for_buffers(edt_p, 1) ;
      process_data(bufptr) ;

      edt_start_buffers(edt_p, 1) ;

   }

}
```

Check compiler options in the EDT-provided `make` files.

DMA Library Routines

The DMA library provides a set of consistent routines across many of the EDT products, with simple yet powerful ring-buffered DMA capabilities. The following table lists the general DMA library routines, described in an order corresponding roughly to their general usefulness.

Routine	Description
Startup/Shutdown	
edt_open	Opens the EDT Product for application access.
edt_open_channel	Opens a specific channel on the EDT Product for application access.
edt_close	Terminates access to the EDT Product and releases resources.
edt_parse_unit	Parses an EDT device name string.
Input/Output	
edt_read	Single, application-level buffer read from the EDT Product.
edt_write	Single, application-level buffer write to the EDT Product.
edt_start_buffers	Begins DMA transfer from or to specified number of buffers.
edt_stop_buffers	Stops DMA transfer after the current buffer(s) complete(s).
edt_check_for_buffers	Checks whether the specified number of buffers have completed without blocking.
edt_done_count	Returns absolute (cumulative) number of completed buffers.
edt_get_todo	Gets the number of buffers that the driver has been told to acquire.
edt_wait_for_buffers	Blocks until the specified number of buffers have completed.
edt_wait_for_next_buffer	Waits for the next buffer that completes DMA.
edt_wait_buffers_timed	Blocks until the specified number of buffers have completed; returns a pointer to the time that the last buffer finished.
edt_next_writebuf	Returns a pointer to the next buffer scheduled for output DMA.
edt_set_buffer	Sets which buffer should be started next.
edt_set_buffer_size	Used to change the size or direction of one of the ring buffers.
edt_last_buffer	Waits for the last buffer that has been transferred.
edt_last_buffer_timed	Like edt_last_buffer but also returns the time at which the dma was complete on this buffer.
edt_configure_ring_buffers	Configures the ring buffers.
edt_buffer_addresses	Returns an array of addresses referencing the ring buffers.
edt_disable_ring_buffers	Stops DMA transfer, disables ring buffers and releases resources.
edt_ring_buffer_overrun	Detects ring buffer overrun which may have corrupted data.
edt_reset_ring_buffers	Stops DMA in progress and resets the ring buffers.
edt_configure_block_buffers	Configures ring buffers using a contiguous block of memory.
edt_startdma_action	Specifies when to perform the action at the start of a dma transfer as set by edt_startdma_reg().
edt_enddma_action	Specifies when to perform the action at the end of a dma transfer as set by edt_ednddma_reg().

Routine	Description
edt_startdma_reg	Specifies the register and value to use at the start of dma, as set by edt_startdma_action().
edt_abort_dma	Cancels the current DMA, resets pointers to the current buffer.
edt_ablort_current_dma	Cancels the current DMA, moves pointers to the next buffer.
edt_get_bytecount	Returns the number of bytes transferred.
edt_timeouts	Returns the cumulative number of timeouts since the device was opened.
edt_get_timeout_count	Returns the number of bytes transferred as of the last timeout.
edt_set_timeout_action	Sets the driver behavior on a timeout.
edt_get_timeout_goodbits	Returns the number of bits from the remote device since the last timeout.
edt_do_timeout	Causes the driver to perform the same actions as it would on a timeout (causing partially filled fifos to be flushed and dma to be aborted).
edt_get_rtimeout	Gets the DMA read timeout period.
edt_set_rtimeout	Sets how long to wait for a DMA read to complete, before returning.
edt_get_wtimeout	Gets the DMA write timeout period.
edt_set_wtimeout	Sets how long to wait for a DMA write to complete, before returning.
edt_get_timestamp	Gets the seconds and microseconds timestamp of dma completion on the buffer specified by bufnum.
edt_get_reftime	Gets the seconds and mircroseconds timestamp in the same format as the buffer_timed function.
edt_ref_tmstamp	Used for debugging. Able to see a history with `setdebug -g` with an application defined event in the same timeline as driver events.
edt_get_burst_enable	Returns a value indicating whether PCI Bus burst transfers are enabled during DMA.
edt_set_burst_enable	Turns on or off PCI Bus burst transfers during DMA.
edt_get_firstflush	Returns the value set by edt_set_firstflush(). This is an obsolete function.
edt_set_firstflush	Tells whether and when to flush FIFOs before DMA.
edt_flush_fifo	Flushes the EDT Product FIFOs.
edt_get_goodbits	Returns the number of bits from the remote device.
Control	
edt_set_event_func	Defines a function to call when an event occurs.
edt_remove_event_func	Removes a previously set event function.
edt_reg_read	Reads the contents of the specified EDT Product register.
edt_reg_write	Writes a value to the specified EDT Product register.
edt_reg_and	ANDs the value provided with the value of the specified EDT Product register.
edt_reg_or	ORs the value provided with the value of the specified EDT Product

Routine	Description
	register.
edt_get_foicount	Returns the number of RCI modules connected to the EDT FOI (fiber optic interface) board.
edt_set_foiunit	Sets which RCI unit to address with subsequent serial and register read/write functions.
edt_intfc_write	A convenience routine, partly for backward compatability, to access the XILINX interface registers.
edt_intfc_write_short	A convenience routine, partly for backward compatability, to access the XILINX interface registers.
edt_intfc_write_32	A convenience routine, partly for backward compatability, to access the XILINX interface registers.
Utility	
edt_msleep	Sleep for the specified number of microseconds.
edt_alloc	Allocate page-aligned memory in a system-independent way.
edt_free	Free the memory allocated with *edt_alloc*.
edt_perror	Prints a system error message in case of error.
edt_errno	Returns an operating system-dependent error number.
edt_access	Determines file access independent of operating system.
edt_get_bitpath	Obtains pathname to the currently loaded interface bitfile from the driver.

edt_open

Description

Opens the specified EDT Product and sets up the device handle.

Syntax

```
#include "edtinc.h"
EdtDev *edt_open(char *devname, int unit) ;
```

Arguments

devname	a string with the name of the EDT Product board. For example, `"edt"`.
unit	specifies the device unit number

Return

A handle of type (EdtDev *), or NULL if error. (The structure(EdtDev *) is defined in libedt.h.) If an error occurs, check the errno global variable for the error number. The device name for the EDT Product is “edt”. Once opened, the device handle may be used to perform I/O using edt_read(), edt_write(), edt_configure_ring_buffers(), and other input-output library calls.

edt_open_channel

Description

Opens a specific DMA channel on the specified EDT Product, when multiple channels are supported by the Xilinx firmware, and sets up the device handle. Use edt_close() to close the channel.

Syntax

```
#include "edtinc.h"
EdtDev *edt_open_channel(char *devname, int unit, int channel) ;
```

Arguments

devname	a string with the name of the EDT Product board. For example, `"edt"`.
unit	specifies the device unit number
channel	specifies the DMA channel number counting from zero

Return

A handle of type (EdtDev *), or NULL if error. (The structure(EdtDev *) is defined in libedt.h.) If an error occurs, check the errno global variable for the error number. The device name for the EDT Product is “edt”. Once opened, the device handle may be used to perform I/O using edt_read(), edt_write(), edt_configure_ring_buffers(), and other input-output library calls.

edt_close

Description

Shuts down all pending I/O operations, closes the device or channel and frees all driver resources associated with the device handle.

Syntax

```
#include "edtinc.h"
int edt_close(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

0 on success; –1 on error. If an error occurs, call `edt_perror()` to get the system error message.

edt_parse_unit

Description

Parses an EDT device name string. Fills in the name of the device, with the `default_device` if specified, or a default determined by the package, and returns a unit number. Designed to facilitate a flexible device/unit command line argument scheme for application programs. Most EDT example/utility programs use this subroutine to allow users to specify either a unit number alone or a device/unit number concatenation.

For example, if you are using a PCI CD, then either `xtest -u 0` or `xtest -u pcd0` could both be used, since `xtest` sends the argument to `edt_parse_unit`, and the subroutine parses the string to returns the device and unit number separately.

Syntax

```
int edt_parse_unit(char *str, char *dev, char *default_dev)
```

Arguments

str	device name string. Should be either a unit number (“0” - “8”) or device/unit concantenation (“pcd0,” “pcd1,” etc.)
dev	device string, filled in by the routine. For example, “pcd.”
default_dev	device name to use if none is given in the str argument. If NULL, will be filled in by the default device for the package in use. For example, if the code base is from a PCI CD package, the default_dev will be set to “pcd.”

Return

Unit number or -1 on error. The first device is unit 0.

See Also

example/utility programs xtest.c, initcam.c, take.c

edt_read

Description

Performs a read on the EDT Product. For those on UNIX systems, the UNIX 2 GB file offset bug is avoided during large amounts of input or output, that is, reading past 231 bytes does not fail. This call is not multibuffering, and no transfer is active when it completes.

Syntax

```
#include "edtinc.h"
int edt_read(EdtDev *edt_p, void *buf, int size);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
buf	address of buffer to read into
size	size of read in bytes

Return

The return value from read, normally the number of bytes read; –1 is returned in case of error. Call edt_perror() to get the system error message.

Note If using timeouts, call edt_timeouts after edt_read returns to see if the number of timeouts has incremented. If it has incremented, call edt_get_timeout_count to get the number of bytes transferred into the buffer. DMA does not automatically continue on to the next buffer, so you need to call edt_start_buffers to move on to the next buffer in the ring.

edt_write

Description

Perform a write on the EDT Product. For those on UNIX systems, the UNIX 2 GB file offset bug is avoided during large amounts of input or output; that is, writing past 231 does not fail. This call is not multibuffering, and no transfer is active when it completes.

Syntax

```
#include "edtinc.h"
int edt_write(EdtDev *edt_p, void *buf, int size);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
buf	address of buffer to write from
size	size of write in bytes

Return

The return value from write; –1 is returned in case of error. Call edt_perror() to get the system error message.

Note

If using timeouts, call edt_timeouts after edt_write returns to see if the number of timeouts has incremented. If it has incremented, call edt_get_timeout_count to get the number of bytes transferred into the buffer. DMA does not automatically continue on to the next buffer, so you need to call edt_start_buffers to move on to the next buffer in the ring.

edt_start_buffers

Description

Starts DMA to the specified number of buffers. If you supply a number greater than the number of buffers set up, DMA continues looping through the buffers until the total count has been satisfied.

Syntax

```
#include "edtinc.h"
int edt_start_buffers(EdtDev *edt_p, int bufnum);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
bufnum	Number of buffers to release to the driver for transfer. An argument of 0 puts the driver in free running mode, and transfers run continuously until **edt_stop_buffers()** is called.

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_stop_buffers

Description

Stops DMA transfer after the current buffer has completed. Ring buffer mode remains active, and transfers will be continued by calling edt_start_buffers().

Syntax

```
#include "edtinc.h"
int edt_stop_buffers(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_check_for_buffers

Description

Checks whether the specified number of buffers have completed without blocking.

Syntax

```
#include "edtinc.h"
void *edt_check_for_buffers(EdtDev *edt_p, int count);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.
count	number of buffers. Must be 1 or greater. Four is recommended.

Return

Returns the address of the ring buffer corresponding to count if it has completed DMA, or NULL if count buffers are not yet complete.

Note If the ring buffer is in free-running mode and the application cannot process data as fast as it is acquired, DMA will wrap around and overwrite the referenced buffer. The application must ensure that the data in the buffer is processed or copied out in time to prevent overrun.

edt_done_count

Description

Returns the cumulative count of completed buffer transfers in ring buffer mode.

Syntax

```
#include "edtinc.h"
int edt_done_count(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

The number of completed buffer transfers. Completed buffers are numbered consecutively starting with 0 when edt_configure_ring_buffers() is invoked. The index of the ring buffer most recently completed by the driver equals the number returned modulo the number of ring buffers. –1 is returned if ring buffer mode is not configured. If an error occurs, call edt_perror() to get the system error message.

edt_get_todo

Description

Gets the number of buffers that the driver has been told to acquire. This allows an application to know the state of the ring buffers within an interrupt, timeout, or when cleaning up on close. It also allows the application to know how close it is getting behind the acquisition. It is not normally needed.

Syntax

```
uint_t edt_get_todo(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Example

```
int curdone;
int curtodo;
curdone=edt_done_count(pdv_p);
curtodo=edt_get_todo(pdv_p);
/* curtodo--curdone how close the dma is to catching with our
processing */
```

Return

Number of buffers started via edt_start_buffers.

See Also

edt_done_count(), edt_start_buffers(), edt_wait_for_buffers()

edt_wait_for_buffers

Description

Blocks until the specified number of buffers have completed.

Syntax

```
#include "edtinc.h"
void *edt_wait_buffers(EdtDev *edt_p, int count);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
count	How many buffers to block for. Completed buffers are numbered relatively; start each call with 1.

Return

Address of last completed buffer on success; NULL on error. If an error occurs, call edt_perror() to get the system error message.

Note If using timeouts, call edt_timeouts after edt_wait_for_buffers returns to see if the number of timeouts has incremented. If it has incremented, call edt_get_timeout_count to get the number of bytes transferred into the buffer. DMA does not automatically continue on to the next buffer, so you need to call edt_start_buffers to move on to the next buffer in the ring.

Note If the ring buffer is in free-running mode and the application cannot process data as fast as it is acquired, DMA will wrap around and overwrite the referenced buffer. The application must ensure that the data in the buffer is processed or copied out in time to prevent overrun.

edt_wait_for_next_buffer

Description

Waits for the next buffer that finishes DMA. Depending on how often this routine is called, buffers that have already completed DMA might be skipped.

Syntax

```
#include "edtinc.h"
void *edt_wait_for_next_buffer(EdtDev *edt_p) ;
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

Returns a pointer to the buffer, or NULL on failure. If an error occurs, call edt_perror() to get the system error message.

edt_wait_buffers_timed

Description

Blocks until the specified number of buffers have completed with a pointer to the time the last buffer finished.

Syntax

```
#include "edtinc.h"
void *edt_wait_buffers_timed (EdtDev *edt_p, int count, uint *timep);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
count	buffer number for which to block. Completed buffers are numbered cumulatively starting with 0 when the EDT Product is opened.
timep	pointer to an array of two unsigned integers. The first integer is seconds, the next integer is microseconds representing the system time at which the buffer completed.

Return

Address of last completed buffer on success; NULL on error. If an error occurs, call edt_perror() to get the system error message.

Note If the ring buffer is in free-running mode and the application cannot process data as fast as it

is acquired, DMA will wrap around and overwrite the referenced buffer . The application must ensure that the data in the buffer is processed or copied out in time to prevent overrun.

edt_next_writebuf

Description

Returns a pointer to the next buffer scheduled for output DMA, in order to fill the buffer with data.

Syntax

```
#include "edtinc.h"
void *edt_next_writebuf(EdtDev *edt_p) ;
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

Returns a pointer to the buffer, or NULL on failure. If an error occurs, call edt_perror() to get the system error message.

edt_set_buffer

Description

Sets which buffer should be started next. Usually done to recover after a timeout, interrupt, or error.

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Syntax

```
#include "edtinc.h"
void *edt_next_writebuf(EdtDev *edt_p) ;
```

Example

```
u_int curdone;
edt_stop_buffers(edt_p);
curdone=edt_done_count(edt_p);
edt_set_buffer(edt_p, 0);
```

Return

0 on success, -1 on failure.

See Also

edt_stop_buffers(), edt_done_count(), edt_get_todo()

edt_set_buffer_size

Description

Used to change the size or direction of one of the ring buffers. Almost never used. Mixing directions requires detailed knowledge of the interface since pending preloaded DMA transfers need to be coordinated with the interface fifo direction. For example, a dma write will complete when the data is in the output fifo, but the dma read should not be started until the data is out to the external device. Most applications requiring fast mixed reads/writes have worked out more cleanly using seperate, simultaneous, read and write dma transfers using different dma channels.

Arguments

edt_p	device handle returned from edt_open or edt_open_channel
which_buf	index of ring buffer to change
size	size to change it to
write_flag	direction

Syntax

```
int edt_set_buffer_size(EdtDev *edt_p, unsigned int which_buf,
unsigned int size, unsigned int write_flag)
```

Example

```
u_int bufnum=3;
u_int bsize=1024;
u_int dirflag=EDT_WRITE;
int ret;
ret=edt_set_buffer_size(edt_p, bufnum, bsize, dirflag);
```

Return

0 on success, -1 on failure.

See Also

edt_open_channel(), redpcd8.c, rd16.c, rdssdio.c, wrssdio.c

edt_last_buffer

Description

Waits for the last buffer that has been transferred. This is useful if the application cannot keep up with buffer transfer. If this routine is called for a second time before another buffer has been transferred, it will block waiting for the next transfer to complete.

Arguments

edt_p	device struct returned from edt_open
nSkipped	pointer to an integer which will be filled in with number of buffers skipped, if any.

Syntax

```
unsigned char *edt_last_buffer(EdtDev *edt_p, int *nSkipped)
```

Example

```
int skipped_bufs;
u_char *buf;
buf=edt_last_buffer(edt_p, &skipped_bufs);
```

Return

Address of the image.

See Also

edt_wait_for_buffers, edt_last_buffer_timed

edt_last_buffer_timed

Description

Like edt_last_buffer but also returns the time at which the dma was complete on this buffer. “timep” should point to an array of unsigned integers which will be filled in with the seconds and microseconds of the time the buffer was finished being transferred.

Arguments

edt_p	device struct returned from edt_open
timep	pointer to an unsigned integer array

Syntax

```
unsigned char *edt_last_buffer_timed(EdtDev *edt_p, u_int *timep)
```

Example

```
u_int timestamp [2];
u_char *buf;
buf=edt_last_buffer_timed(edt_p, timestamp);
```

Return

Address of the image.

See Also

edt_wait_for_buffers(), edt_last_buffer(), edt_wait_buffers_timed

edt_configure_ring_buffers

Description

Configures the EDT device ring buffers. Any previous configuration is replaced, and previously allocated buffers are released. Buffers can be allocated and maintained within the EDT device library or within the user application itself.

Syntax

```
#include "edtinc.h"
int edt_configure_ring_buffers(EdtDev *edt_p, int bufsize, int nbufs,
                int data_output, void *bufarray[]);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
bufsize	size of each buffer. For optimal efficiency, allocate a value approximating throughput divided by 20: that is, if transfer occurs at 20 MB per second, allocate 1 MB per buffer. Buffers significantly larger or smaller can overuse memory or lock the system up in processing interrupts at this speed.
nbufs	number of buffers. Must be 1 or greater. Four is recommended for most applications.
data_direction	Indicates whether this connection is to be used for input or output. Only one direction is possible per device or subdevice at any given time: EDT_READ = 0 EDT_WRITE = 1
bufarray	If NULL, the library will allocate a set of page-aligned ring buffers. If not NULL, this argument is an array of pointers to application-allocated buffers; these buffers must match the size and number of buffers specified in this call and will be used as the ring buffers.

Return

0 on success; –1 on error. If all buffers cannot be allocated, none are allocated and an error is returned. Call edt_perror() to get the system error message.

edt_buffer_addresses

Description

Returns an array containing the addresses of the ring buffers.

Syntax

```
#include "edtinc.h"
void **edt_buffer_addresses(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

An array of pointers to the ring buffers allocated by the driver or the library. The array is indexed from zero to n-1 where n is the number of ring buffers set in edt_configure_ring_buffers().

edt_disable_ring_buffers

Description

Disables the EDT device ring buffers. Pending DMA is cancelled and all buffers are released.

Syntax

```
#include "edtinc.h"
int edt_disable_ring_buffers(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_ring_buffer_overrun

Description

Returns true (1) when DMA has wrapped around the ring buffer and overwritten the buffer which the application is about to access. Returns false (0) otherwise.

Syntax

```
#include "edtinc.h"
int edt_ring_buffer_overrun(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

1 (true) when overrun has occurred, corrupting the current buffer, 0 (false) otherwise.

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_reset_ring_buffers

Description

Stops any DMA currently in progress, then resets the ring buffer to start the next DMA at bufnum.

Syntax

```
#include "edtinc.h"
int edt_reset_ring_buffers(EdtDev *edt_p, int bufnum) ;
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.
bufnum	The index of the ring buffer at which to start the next DMA. A number larger than the number of buffers set up sets the current done count to the number suppliedmodulo the number of buffers.

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_configure_block_buffers

Description

Similar to edt_configure_ring_buffers, except that it allocates the ring buffers as a single large block, setting the ring buffer addresses from within that block. This allows reading or writing buffers from/to a file in single chunks larger than the buffer size, which is sometimes considerable more efficient. Buffer sizes are rounded up by PAGE_SIZE so that DMA occurs on a page boundary.

Syntax

```
int edt_configure_block_buffers(EdtDev 8edt_p, int bufsize, int
numbufs, int write_flag, int header_size, int header_before)
```

Arguments

edt_p	device struct returned from edt_open
bufsize	size of the individual buffers
numbufs	number of buffers to create
write_flag	1, if these buffers are set up to go out; 0 otherwise
header_size	if non-zero, additional memory (header_size bytes) will be allocated for each buffer for Header data. The loocation of this header space is determined by the argument `header_before`.
header_before	if non-zero, the header space defined by header_size is placed before the DMA buffer; otherwise, it comes after the DMA buffer. The value returned by edt_wait_for_buffers is always the DMA buffer.

Return

0 on success, -1 on failure.

See Also

edt_configure_ring_buffers

edt_startdma_action

Description

Specifies when to perform the action at the start of a dma transfer as specified by edt_startdma_reg(). A common use of this is to write to a register which signals an external device that dma has started, to trigger the device to start sending. The default is no dma action. The PDV library uses this function to send a trigger to a camera a the start of dma. This function allows the register write to occur in a critical section with the start of dma and at the same time.

Syntax

```
void edt_startdma_action(EdtDev *edt_p, uint_t val);
```

Arguments

edt_p	device struct returned from edt_open
val	One of EDT_ACT_NEVER, EDT_ACT_ONCE, or EDT_ACT_ALWAYS

Example

```
edt_startdma_action(edt_p, EDT_ACT_ALWAYS);
edt_startdma_reg(edt_p, PDV_CMD, PDV_ENABLE_GRAB);
```

Return

void

See Also

edt_startdma_reg(), edt_reg_write(), edt_reg_read()

edt_enddma_action

Description

Specifies when to perform the action at the end of a dma transfer as specified by edt_enddma_reg(). A common use of this is to write to a register which signals an external device that dma is complete, or to change the state of a signal which will be changed at the start of dma, so the external device can look for an edge. The default is no end of dma action. Most applications can set the output signal, if needed, from the application with edt_reg_write(). This routine is only needed if the action must happen within microseconds of the end of dma.

Syntax

```
void edt_enddma_action(EdtDev *edt_p, uint_t val);
```

Arguments

edt_p	device struct returned from edt_open
val	One of EDT_ACT_NEVER, EDT_ACT_ONCE, or EDT_ACT_ALWAYS

Example

```
u_int fnct_value=0x1;
edt_enddma_action(edt_p, EDT_ACT_ALWAYS);
edt_enddma_reg(edt_p, PCD_FUNCT, fnct_value);
```

Return

void

See Also

edt_startdma_action(), edt_startdma_reg(), edt_reg_write(), edt_reg_read()

edt_startdma_reg

Description

Sets the register and value to use at the start of dma, as set by edt_startdma_action().

Syntax

```
void edt_startdma_reg(EdtDev *edt_p, uint_t desc, uint_t val);
```

Arguments

edt_p	device struct returned from edt_open
desc	register description of which register to use as in edtreg.h
val	value to write

Example

```
edt_startdma_action(edt_p, EDT_ACT_ALWAYS);
edt_startdma_reg(edt_p, PDV_CMD, PDV_ENABLE_GRAB);
```

Return

void

See Also

edt_startdma_action()

edt_abort_dma

Description

Stops any transfers currently in progress, resets the ring buffer pointers to restart on the current buffer.

Syntax

```
#include "edtinc.h"
int edt_abort_dma(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_abort_current_dma

Description

Stops the current transfers, resets the ring buffer pointers to the next buffer.

Syntax

```
#include "edtinc.h"
int edt_abort_current_dma(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.

Return

0 on success, -1 on failure

edt_get_bytecount

Description

Returns the number of bytes transferred since the last call of edt_open, accurate to the burst size, if burst is enabled.

Syntax

```
#include "edtinc.h"
int edt_get_bytecount(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

The number of bytes transferred, as described above.

edt_timeouts

Description

Returns the number of read and write timeouts that have occurred since the last call of edt_open.

Syntax

```
#include "edtinc.h"
int edt_timeouts(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

The number of read and write timeouts that have occurred since the last call of edt_open.

edt_get_timeout_count

Description

Returns the number of bytes transferred at last timeout.

Syntax

```
#include "edtinc.h"
int edt_get_timeout_count(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

The number of bytes transferred at last timeout.

edt_set_timeout_action

Description

Sets the driver behavior on a timeout.

Syntax

```
#include "edtinc.h"
void edt_set_timeout_action(EdtDev *edt_p, int action);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*	
action	integer configures the any action taken on a timeout. Definitions:	
	EDT_TIMEOUT_NULL	no extra action taken
	EDT_TIMEOUT_BIT_STROBE	flush any valid bits left in input circuits of SSDIO.

Return

No return value.

edt_get_timeout_goodbits

Description

Returns the number of good bits in the last long word of a read buffer after the last timeout. This routine is called after a timeout, if the timeout action is set to EDT_TIMEOUT_BIT_STROBE. (See edt_set_timeout_action on page 28.)

Syntax

```
#include "edtinc.h"
int edt_get_timeout_goodbits(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

Number 0–31 represents the number of good bits in the last 32-bit word of the read buffer associated with the last timeout.

edt_do_timeout

Description

Causes the driver to perform the same actions as it would on a timeout (causing partially filled fifos to be flushed and dma to be aborted). Used when the application has knowledge that no more data will be sent/accepted. Used when a common timeout cannot be known, such as when acquiring data from a telescope ccd array where the amount of data sent depends on unknown future celestial events. Also used by the library when the operating system can not otherwise wait for an interrupt and timeout at the same time.

Syntax

```
int edt_do_timeout(EdtDev *edt_p)
```

Arguments

edt_p	device struct returned from edt_open

Example

```
edt_do_timeout(edt_p);
```

Return

0 on success, -1 on failure

See Also

ring buffer discussion

edt_get_rtimeout

Description

Gets the current read timeout value: the number of milliseconds to wait for DMA reads to complete before returning.

Syntax

```
#include "edtinc.h"
int edt_get_rtimeout(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from edt_open or edt_open_channel

Return

The number of milliseconds in the current read timeout period.

edt_set_rtimeout

Description

Sets the number of milliseconds for data read calls, such as edt_read(), to wait for DMA to complete before returning. A value of 0 causes the I/O operation to wait forever—that is, to block on a read. Edt_set_rtimeout affects edt_wait_for_buffers (see page XX) and edt_read (see page XX).

Syntax

```
#include "edtinc.h"
int edt_set_rtimeout(EdtDev *edt_p, int value);
```

Arguments

edt_p	device handle returned from edt_open or edt_open_channel
value	The number of milliseconds in the timeout period.

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_get_wtimeout

Description

Gets the current write timeout value: the number of milliseconds to wait for DMA writes to complete before returning.

Syntax

```
#include "edtinc.h"
int edt_get_wtimeout(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from edt_open or edt_open_channel

Return

The number of milliseconds in the current write timeout period.

edt_set_wtimeout

Description

Sets the number of milliseconds for data write calls, such as edt_write(), to wait for DMA to complete before returning. A value of 0 causes the I/O operation to wait forever—that is, to block on a write. Edt_set_wtimeout affects edt_wait_for_buffers (see page XX) and edt_write (see page XX).

Syntax

```
#include "edtinc.h"
int edt_set_wtimeout(EdtDev *edt_p, int value);
```

Arguments

edt_p	device handle returned from edt_open or edt_open_channel
value	The number of milliseconds in the timeout period.

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_get_timestamp

Description

Gets the seconds and microseconds timestamp of when dma was completed on the buffer specified by bufnum. “bufnum” is moded by the number of buffers in the ring buffer, so it can either be an index, or the number of buffers completed.

Syntax

```
int edt_get_timestamp(EdtDev *edt_p, u_int *timep, u_int bufnum)
```

Arguments

edt_p	device struct returned from edt_open
timep	pointer to an unsigned integer array
bufnum	buffer index, or number of buffers completed

Example

```
int timestamp[2];
u_int bufnum=edt_done_count(edt_p);
edt_get_timestamp(edt_p, timestamp, bufnum);
```

Return

0 on success, -1 on failure. Fills in timestamp pointed to by timep.

See Also

edt_timestamp(), edt_done_count(), edt_wait_buffers_timed

edt_get_reftime

Description

Gets the seconds and microseconds timestamp in the same format as the buffer_timed functions. Used for debugging and coordinating dma completion time with other events.

Syntax

```
int edt_get_reftime(EdtDev *edt_p, u_int *timep)
```

Arguments

edt_p	device struct returned from edt_open
timep	pointer to an unsigned integer array
bufnum	buffer index, or number of buffers completed

Example

```
int timestamp[2];
edt_get_regtime(edt_p, timestamp);
```

Return

0 on success, -1 on failure. Fills in timestamp pointed to by timep.

See Also

edt_timestamp(), edt_done_count(), edt_wait_buffers_timed

edt_ref_tmstamp

Description

Used for debugging and viewing a history with setdebug -g with an application-defined event in the same timeline as driver events.

Syntax

```
int edt_ref_tmstamp(EdtDev *edt_p, u_int val)
```

Arguments

edt_p	device struct returned from edt_open
val	an arbitrary value meaningful to the application

Example

```
#define BEFORE_WAIT 0x11212aaaa
#define AFTER_WAIT 0x3344bbbb
u_char *buf;
edt_ref_tmstamp(edt_p, BEFORE_WAIT);
buf=edt_wait_for_buffer(edt_p);
edt_reg_tmstamp(edt_p, AFTER_WAIT);
/* now look at output of setdebug -g */
```

Return

0 on success, -1 on failure.

See Also

documentation on setdebug

edt_get_burst_enable

Description

Returns the value of the burst enable flag, determining whether the DMA master transfers as many words as possible at once, or transfers them one at a time as soon as the data is acquired. Burst transfers are enabled by default to optimize use of the bus. For more information, see edt_set_burst_enable on page 33.

Syntax

```
#include "edtinc.h"
int edt_get_burst_enable(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from edt_open or edt_open_channel

Return

A value of 1 if burst transfers are enabled; 0 otherwise.

edt_set_burst_enable

Description

Sets the burst enable flag, determining whether the DMA master transfers as many words as possible at once, or transfers them one at a time as soon as the data is acquired. Burst transfers are enabled by default to optimize use of the bus; however, you may wish to disable them if data latency is an issue, or for diagnosing DMA problems.

Syntax

```
#include "edtinc.h"
void edt_set_burst_enable(EdtDev *edt_p, int onoff);
```

Arguments

edt_p	device handle returned from edt_open or edt_open_channel
onoff	A value of 1 turns the flag on (the default); 0 turns it off.

Return

No return value.

edt_get_firstflush

Description

Returns the value set by edt_set_firstflush(). This is an obsolete function that was only used as a kludge to detect EDT_ACT_KBS (also obsolete).

Syntax

```
int edt_get_firstflush(EdtDev *edt_p)
```

Arguments

edt_p	*device struct returned from edt_open.*

Example

```
int application_should_already_know_this;
application_should_already_know_this=edt_get_firstflush(edt_p);
```

Return

Yes

See Also

edt_set_firstflush

edt_set_firstflush

Description

Tells whether and when to flush the FIFOs before DMA transfer. By default, the FIFOs are not flushed. However, certain applications may require flushing before a given DMA transfer, or before each transfer.

Syntax

```
#include "edtinc.h"
int *edt_set_firstflush(EdtDev *edt_p, int flag) ;
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.	
flag	Tells whether and when to flush the FIFOs. Valid values are:	
	EDT_ACT_NEVER	don't flush before DMA transfer (default)
	EDT_ACT_ONCE	flush before the start of the next DMA transfer
	EDT_ACT_ALWAYS	flush before the start of every DMA transfer

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_flush_fifo

Description

Flushes the board's input and output FIFOs, to allow new data transfers to start from a known state.

Syntax

```
#include "edtinc.h"
void edt_flush_fifo(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

No return value.

edt_get_goodbits

Description

Returns the current number of good bits in the last long word of a read buffer (0 through 31).

Syntax

```
#include "edtinc.h"
int edt_get_goodbits(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*

Return

Number 0–31 represents the number of good bits in the 32-bit word of the current read buffer.

edt_set_event_func

Description

Defines a function to call when an event occurs. Use this routine to send an application-specific function when required; for example, when DMA completes, allowing the application to continue executing until the event of interest occurs.

If you wish to receive notification of one event only, and then disable further event notification, send a final argument of 0 (see the continue parameter described below). This disables event notification at the time of the callback to your function.

Syntax

```
#include "edtinc.h"
int edt_set_event_func(EdtDev *edt_p, int event, void (*func)(void
*),
                                    void *data, int continue);
```

Arguments

edt_p device handle returned from *edt_open* or *edt_open_channel*.

event The event that causes the function to be called. Valid events are:

Event	Description	Board
EDT_PDV_EVENT_ACQUIRE	Image has been acquired; shutter has closed; subject can be moved if necessary; DMA will now restart	PCI DV, PCI DVK, PCI FOI
EDT_PDV_EVENT_FVAL	Frame Valid line is set	PCI DV, PCI DVK
EDT_EVENT_P16D_DINT	Device interrupt occurred	PCI 16D
EDT_EVENT_P11W_ATTN	Attention interrupt occurred	PCI 16D

	EDT_EVENT_P11W_CNT	Count interrupt occurred	PCI 16D
	EDT_EVENT_PCD_STAT1	Interrupt occurred on Status 1 line	PCI CD
	EDT_EVENT_PCD_STAT2	Interrupt occurred on Status 2 line	PCI CD
	EDT_EVENT_PCD_STAT3	Interrupt occurred on Status 3 line	PCI CD
	EDT_EVENT_PCD_STAT4	Interrupt occurred on Status 4 line	PCI CD
	EDT_EVENT_ENDDMA	DMA has completed	ALL
func	The function you've defined to call when the event occurs.		
data	Pointer to data block (if any) to send to the function as an argument; usually edt_p.		
continue	Flag to enable or disable continued event notification. A value of 0 causes an implied edt_remove_event_func as the event is triggered.		

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_remove_event_func

Description

Removes an event function previously set with edt_set_event_func.

Note This routine is implemented on PCI Bus platforms only.

Syntax

```
#include "edtinc.h"
int edt_remove_event_func(EdtDev *edt_p, int event);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*.
event	The event that causes the function to be called. Valid events are as listed in edt_set_event_func on page 35.

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_reg_read

Description

Reads the specified register and returns its value. Use this routine instead of using ioctls.

Syntax

```
#include "edtinc.h"
uint edt_reg_read(EdtDev *edt_p, uint address);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
address	The name of the register to read. Use the names provided in the register descriptions in the section entitled “Hardware.”

Return

The value of the register.

edt_reg_write

Note Use this routine with care; it writes directly to the hardware. An incorrect value can crash your system, possibly causing loss of data.

Description

Write the specified value to the specified register. Use this routine instead of using ioctls.

Syntax

```
#include "edtinc.h"
void edt_reg_write(EdtDev *edt_p, uint address, uint value);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
address	The name of the register to write. Use the names provided in the register descriptions in the section entitled “Hardware.”
value	The desired value to write in the register.

Return

No return value.

edt_reg_and

Note Use this routine with care; it writes directly to the hardware. An incorrect value can crash your system, possibly causing loss of data.

Description

Performs a bitwise logical AND of the value of the specified register and the value provided in the argument; the result becomes the new value of the register. Use this routine instead of using ioctls.

Syntax

```
#include "edtinc.h"
uint edt_reg_and(EdtDev *edt_p, uint address, uint mask);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
address	The name of the register to modify. Use the names provided in the register descriptions in the section entitled “Hardware.”
mask	The value to AND with the register.

Return

The new value of the register.

edt_reg_or

Note Use this routine with care; it writes directly to the hardware. An incorrect value can crash your system, possibly causing loss of data.

Description

Performs a bitwise logical OR of the value of the specified register and the value provided in the argument; the result becomes the new value of the register. Use this routine instead of using ioctls.

Syntax

```
#include "edtinc.h"
uint edt_reg_or(EdtDev *edt_p, uint address, uint mask);
```

Arguments

edt_p	device handle returned from *edt_open* or *edt_open_channel*
address	The name of the register to modify. Use the names provided in the register descriptions in the section entitled “Hardware.”
mask	The value to OR with the register.

Return

The new value of the register.

edt_get_foicount

Description

Returns the number of RCI modules connected to the EDT FOI (fiber optic interface) board.

Syntax

```
int edt_get_foicount(EdtDev *edt_p)
```

Arguments

edt_p	device struct returned from edt_open

Example

```
int num-rcis;
num_rcia=edt_get_foicount(edt_p);
```

Return

Integer

See Also

edt_set_foiunit(), edt_get_foiunit(), edt_set_foicount()

edt_set_foicount

Description

Sets which RCI unit to address with subsequent serial and register read/write functions. Used with the PDV FOI.

Syntax

```
int edt_set_foicount(EdtDev *edt_p, int unit)
```

Arguments

edt_p	device struct returned from edt_open
unit	unit number of RCI unit

Example

```
int nextunit;
nextunit=3;
edt_set_foiunit(edt_p, nextunit);
```

Return

0 on success, -1 on failure

See Also

pdv_serial_write(), edt_reg_write(), edt_reg_read(), pdv_serial_read()

edt_intfc_write

Description

A convenience routine, partly for backward compatability, to access the XILINX interface registers. The register descriptors used be edt_reg_write() can also be used, since edt_intfc_write masks off the offset.

Syntax

```
void edt_intfc_write(EdtDev *edt_p, uint_t offset, uchar_t val)
```

Arguments

edt_p	device struct returned from edt_open
offset	integer offset into XILINX interface, or register descriptor
val	unsigned character value to set

Example

```
u_char fnct1=1;
edt_intfc_write(edt_p, PCD_FUNCT, fnct1);
```

Return

void

See Also

edt_intfc_read(), edt_reg_write(), edt_intfc_write_short()

edt_intfc_read

Description

A convenience routine, partly for backward compatability, to access the XILINX interface registers. The register descriptors used be edt_reg_write() can also be used, since edt_intfc_read masks off the offset.

Syntax

```
u_char
edt_intfc_read(EdtDev *edt_p, uint_t offset)
```

Arguments

edt_p	device struct returned from edt_open
offset	integer offset into XILINX interface, or register descriptor
val	unsigned character value to set

Example

```
u_char rfnct=edt_intfc_read(edt_p, PCD_FUNCT);
```

Return

void

See Also

edt_intfc_write(), edt_reg_read(), edt_intfc_read_short()

edt_intfc_write_short

Description

A convenience routine, partly for backward compatability, to access the XILINX interface registers. The register descriptors used be edt_reg_write() can also be used, since edt_intfc_write_short masks off the offset.

Syntax

```
void edt_intfc_write_short(EdtDev *edt_p, uint_t offset, u_short val)
```

Arguments

edt_p	device struct returned from edt_open
offset	integer offset into XILINX interface, or register descriptor
val	unsigned character value to set

Example

```
u_short width=1024;
edt_intfc_write_short(edt_p, CAM_WIDTH, width);
```

Return

void

See Also

edt_intfc_write(), edt_reg_write()

edt_intfc_read_short

Description

A convenience routine, partly for backward compatability, to access the XILINX interface registers. The register descriptors used be edt_reg_write() can also be used, since edt_intfc_read_short masks off the offset.

Syntax

```
u_short
edt_intfc_read_short(EdtDev *edt_p, unit_t offset)
```

Arguments

edt_p	device struct returned from edt_open
offset	integer offset into XILINX interface, or register descriptor
val	unsigned character value to set

Example

```
u_short r_camw=edt_intfc_read_short(edt_p, CAM_WIDTH);
```

Return

void

See Also

edt_intfc_read(), edt_reg_read()

edt_intfc_write_32

Description

A convenience routine, partly for backward compatability, to access the XILINX interface registers. The register descriptors used be edt_reg_write() can also be used, since edt_intfc_write_32 masks off the offset.

Syntax

```
void edt_intfc_write_32(EdtDev *edt_p, uint_t offset, unit_t val)
```

Arguments

edt_p	device struct returned from edt_open
offset	integer offset into XILINX interface, or register descriptor
val	unsigned character value to set

Example

```
u_int value=0x12345678;
edt_intfc_write_32(edt_p, MAGIC_OFF1, value);
```

Return

void

See Also

edt_intfc_read_32(), edt_reg_write()

edt_intfc_read_32

Description

A convenience routine, partly for backward compatability, to access the XILINX interface registers. The register descriptors used be edt_reg_write() can also be used, since edt_intfc_read_32 masks off the offset.

Syntax

```
uint_t
edt_intfc_read_32(EdtDev *edt_p, uint_t offset)
```

Arguments

edt_p	device struct returned from edt_open
offset	integer offset into XILINX interface, or register descriptor
val	unsigned character value to set

Example

```
u_int r_actkbs=edt_intfc_read_32(edt_p, EDT_ACT_KBS);
```

Return

void

edt_msleep

Description

Causes the process to sleep for the specified number of microseconds.

Syntax

```
#include "edtinc.h"
int edt_microsleep(u_int usecs) ;
```

Arguments

usecs	The number of microseconds for the process to sleep.

Return

0 on success; –1 on error. If an error occurs, call edt_perror() to get the system error message.

edt_alloc

Description

Convenience routine to allocate memory in a system-independent way. The buffer returned is page aligned. Uses VirtualAlloc on Windows systems, *valloc* on UNIX-based systems.

Syntax

```
#include "edtinc.h"
int
edt_alloc(int nbytes)
```

Arguments

nbytes	number of bytes of memory to allocate.

Example

```
unsigned char *buf = edt_alloc(1024);
```

Returns

The address of the allocated memory, or NULL on error. If NULL, use edt_perror on page 43 to print the error.

edt_free

Description

Convenience routine to free the memory allocated with pdv_alloc (above).

Syntax

```
#include "edtinc.h"

int
edt_free(unsigned char *buf)
```

Arguments

buf	Address of memory buffer to free.

Example

```
edt_free(buf);
```

Returns

0 if successful, –1 if unsuccessful.

edt_perror

Description

Formats and prints a system error.

Syntax

```
#include "edtinc.h"

void
edt_perror(char *errstr)
```

Arguments

errstr	Error string to include in the printed error output.

Return

No return value. See edt_errno below for an example.

edt_errno

Description

Returns an operating system-dependent error number.

Syntax

```
#include "edtinc.h"

int
edt_errno(void)
```

Arguments

None.

Return

32-bit integer representing the operating system-dependent error number generated by an error.

Example

```
if ((edt_p = edt_open("p11w",0))==NULL
{
      int error_num;

      edt_perror("edt_open");
      error_num = edt_errno(edt_p);
}
```

edt_access

Description

Determines file access, independent of operating system. This a convenience routine that maps to acccess() on Unix/Linus systems and _access() on Windows systems.

Syntax

```
int edt_access(char *fname, int perm)
```

Arguments

edt_p	device struct returned from edt_open
fname	path name of the file to check access permissions
perm	permission flag(s) to test for. See access() (Unix/Linux) or _access() (Windows) for valid values.

Example

```
if(edt_access("file.ras", F_OK))
printf("Warning: overwriting file %s\n");
```

Return

0 on success, -1 on failure

edt_get_bitpath

Description

Obtains pathname to the currently loaded interface bitfile from the driver. The program "bitload" sets this string in the driver when an interface bitfile is successfully loaded.

Syntax

```
#include "edtinc.h"
```

```
int edt_get_bitpath(EdtDev *edt_p, char *bitpath, int size);
```

Arguments

edt_p	device handle returned from edt_open or edt_open_channel
bitpath	address of a character buffer of at least 128 bytes
size	number of bytes in the above character buffer

Return

0 on success, -1 on failure

EDT Message Handler Library

The edt error library provides generalized error and message handling for the edt and pdv libraries. The primary purpose of the routines is to provide a method for application programs to intercept and handle edtlib and pdvlib error, warning debug messages, but can also be used for application messages.

By default, output goes to the console (stdout), but user defined functions can be substituted. For example, a function that pops up a window and displays the text in that window. Different message levels can be set for different output, and multiple message handles can even exist within an application, with different message handlers associated with them.

Message Definitions

User application messages

EDTAPP_MSG_FATAL

EDTAPP_MSG_WARNING

EDTAPP_MSG_INFO_1

EDTAPP_MSG_INFO_2

Edtlib messages

EDTLIB_MSG_FATAL

EDTLIB_MSG_WARNING

EDTLIB_MSG_INFO_1

EDTLIB_MSG_INFO_2

Pdvlib messages

PDVLIB_MSG_FATAL

PDVLIB_MSG_WARNING

PDVLIB_MSG_INFO_1

PDVLIB_MSG_INFO_2

Library and application messages

EDT_MSG_FATAL (defined as EDTAPP_MSG_FATAL | EDTLIB_MSG_FATAL | PDVLIB_MSG_FATAL)

EDT_MSG_WARNING (defined as EDTAPP_MSG_WARNING | EDTLIB_MSG_WARNING | PDVLIB_MSG_WARNING)

EDT_MSG_INFO_1 (defined as EDTAPP_MSG_INFO_1 | EDTLIB_MSG_INFO_2 | PDVLIB_MSG_INFO_2)

EDT_MSG_INFO_2 (defined as EDTAPP_MSG_INFO_2 | EDTLIB_MSG_INFO_2 | PDVLIB_MSG_INFO_2)

Message levels are defined by flag bits, and each bit can be set or cleared individually. So for example if you want a message handler to be called for fatal and warning application messages only, you would specify EDTAPP_MSG_FATAL | EDTAPP_MSG_WARNING.

As you can see, the edt and pci dv libraries have their own message flags. These can be turned on and off from within an application, and also by setting the environment variables EDTDEBUG and PDVDEBUG, respectively, to values greater than zero.

Application programs would normally specify combinations of either the EDTAPP_MSG_ or EDT_MSG flags for their messages.

Files

edt_error.h: header file (automatically included if edtinc.h is included)

edt_error.c: message subroutines

The EdtMsgHandler structure is defined in edt_error.h. Application programmers should not access structure elements directly; instead always go through the error subroutines.

edt_msg_init

Description

Initializes a message handle to defaults. The message file is initialized to stderr. The output subroutine pointer is set to fprintf (console output). The message level is set to EDT_MSG_WARNING | EDT_MSG_FATAL.

Syntax

```
void edt_msg_init(EdtMsgHandler *msg_p)
```

Arguments

msg_p	pointer to message handler structure to initialize

Return

Void

Example

```
EdtMsgHandler msg_p;
edt_msg_init(&msg_p);
```

See Also

edt_msg_output

edt_msg

Description

Submits a message to the default message handler, which will conditionally (based on the flag bits) send the message as an argument to the default message handler function. Uses the default message handle, and is equivalent to calling edt_msg_output(edt_msg_default_handle(), ...). To submit a message for handling from other than the default message handle, use edt_msg_output.

Syntax

```
int edt_msg(int level, char *format, ...)
```

Arguments

level	an integer variable that contains flag bits indicating what 'level' message it is. Flag bits are described in the overview.
format	a string and arguments describing the format. Uses vsprintf to print formatted text to a string, and sends the result to the handler subroutine. Refer to the printf manual page for formatting flags and options.

Return

Void

Example

```
edt_msg(EDTAPP_MSG_WARNING, "file '%s' not found", fname);
```

edt_msg_output

Description

Submits a message using the msg_p message handle, which will conditionally (based on the flag bits) send the message as an argument to the handle's message handler function. To submit a message for handling by the default message handle, edt_msg.

Syntax

```
int edt_msg_output(EdtMsgHandler *msg_p, int level, char *format,
...)
```

Arguments

msg_p	pointer to message handler, initiailzed by edt_msg_init
level	an integer variable that contains flag bits indicating what 'level' message it is. Flag bits are described in the overview.
format	a string and arguments describing the format. Uses vsprintf to print formatted text to a string, and sends the result to the handler subroutine. Refer to the printf manual page for formatting flags and options.

Return

Void

Example

```
EdtMsgHandler msg_p;
edt_msg_init(&msg_p);
edt_msg_set_function(msg_p, (EdtMsgFunction *)my_error_popup);
```

```
edt_msg_set_level(msg_p, EDT_MSG_FATAL | EDT_MSG_WARNING);
if (edt_access(fname, 0) != 0)
     edt_msg_output(msg_p, EDTAPP_MSG_WARNING, "file '%s' not
found", fname);
```

edt_msg_close

Description

Closes and frees up memory associated with a message handle. Use only on message handles that have been explicitly initialized by edt_msg_init. Do not try to close the default message handle.

Syntax

```
int edt_msg_close(EdtMsgHandler *msg_p)
```

Arguments

msg_p	the message handle to close

Return

0 on success, -1 on failure

edt_msg_set_level

Description

Sets the "message level" flag bits that determine whether to call the message handler for a given message. The flags set by this function are ANDed with the flags set in each edt_msg call, to determine whether the call goes to the message function and actually results in any output.

Syntax

```
void edt_msg_set_level(EdtMsgHandler *msg_p, int newlevel)
```

Arguments

msg_p	the message handle

Example

```
edt_msg_set_level(edt_msg_default_level(),
EDT_MSG_FATAL|EDT_MSG_WARNING);
```

Return

Void

edt_msg_set_function

Description

Sets the function to call when a message event occurs. The default message function is printf (outputs to the console); edt_msg_set_function allows programmers to substitute any type of message handler (pop-up callback, file write, etc).

Syntax

```
void edt_msg_set_function(EdtErrorFunction f)
```

Arguments

msg_p	the message handle

Example

```
See edt_msg
```

Return

Void

See Also

edt_msg, edt_msg_set_level

edt_msg_set_msg_file

Description

Sets the output file pointer for the message handler. Expects a file handle for a file that is already open.

Syntax

```
void edt_msg_set_msg_file(EdtMsgHandler *msg_p, FILE *fp)
```

Arguments

msg_p	the message handle
p	pointer to a file handle that is already open, to which the messages should be output

Example

```
EdtMsgHandler msg_p;
      FILE *fp = fopen("messages.out", "w");
      edt_msg_init(&msg_p);
      edt_msg_set_file(&msg_p, fp);
```

Return

Void

edt_msg_perror

Description

Conditionally outputs a system perror using the default message pointer.

Syntax

```
int edt_msg_perror(int level, char *msg)
```

Arguments

level	message level, described in the overview
msg	message to concatenate to the system error

Example

```
if ((fp = fopen ("file.txt", "r")) == NULL)
```

```
edt_sysperror(EDT_FATAL, "file.txt");
```

Return

0 on success, -1 on failure

See Also

edt_perror

The following routines are specific to the PCI 16D:

Routine	Description
p16d_set_command	Sets the value of the Command register.
p16d_get_command	Gets the value of the Command register.
p16d_set_config	Sets the value of the Configuration register.
p16d_get_config	Gets the value of the Configuration register.
p16d_get_stat	Gets the value of the Status register.

p16d_set_command

Description

Sets the value of the Command register.

Syntax

```
#include "edtinc.h"
void p16d_set_command(EdtDev *edt_p, u_short val);
```

Arguments

edt_p	device handle returned from `edt_open`
val	Command register value you want to set

Example

```
u_short val;

val = p16d_get_command(edt_p);
val |= P16D_INIT;
p16d_set_command(edt_p, val);
```

Return

None

p16d_get_command

Description

Gets the value in the Command register.

Syntax

```
#include "edtinc.h"
u_short p16d_get_command(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from `edt_open`

Return

Unsigned short containing the value currently in the Command register.

p16d_set_config

Description

Sets the value of the Configuration register.

Syntax

```
include "edtinc.h"
void p16d_set_config(EdtDev *edt_p, u_short val);
```

Arguments

edt_p	device handle returned from `edt_open`
val	Configuration register value you want to set

Example

```
u_short val;

val = p16d_get_config(edt_p);
val |= P16D_SWAP;
p16d_set_config(edt_p, val);
```

Return

None

p16d_get_config

Description

Gets the value in the Configuration register.

Syntax

```
#include "edtinc.h"
u_short p16d_get_config(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from `edt_open`

Return

Unsigned short containing the value currently in the Configuration register.

p16d_get_stat

Description

Gets the value in the Status register.

Syntax

```
#include "edtinc.h"
u_short p16d_get_stat(EdtDev *edt_p);
```

Arguments

edt_p	device handle returned from `edt_open`

Return

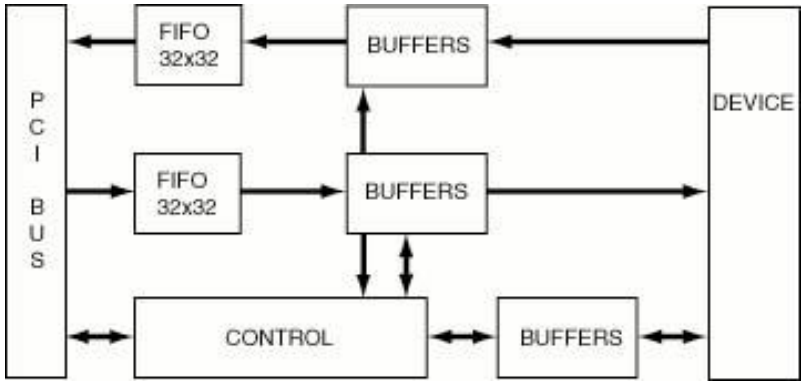
Unsigned short containing the value currently in the Status register.

Example

```
u_short stat;
stat = p16d_get_stat(edt_p);
```

Hardware

This section describes the PCI 16D interface, registers, connectors, and timing. The following figure is a diagram of the PCI 16D interface:



PCI Local Bus Interface

The interface to the PCI Local Bus supports data transfer at 2, 4 and 16 bytes per request. The interface is implemented using programmable logic and high-speed register files. The PCI Local Bus DMA address is maintained in an application-specific integrated circuit (ASIC).

FIFO

The PCI 16D uses First-In-First-Out (FIFO) memory to buffer the data flow to and from the PCI Local Bus. These FIFOs can store up to 128 bytes.

Device Interface

The device interface is implemented with Unibus Driver/Receivers and 180/390 terminators. The receivers have a 1 V hysteresis and 2V noise immunity. The drivers are open-collector type. The device handshake and control are implemented in the ASIC.

See the section entitled “Signals” on page 55 for further details on device signal usage.

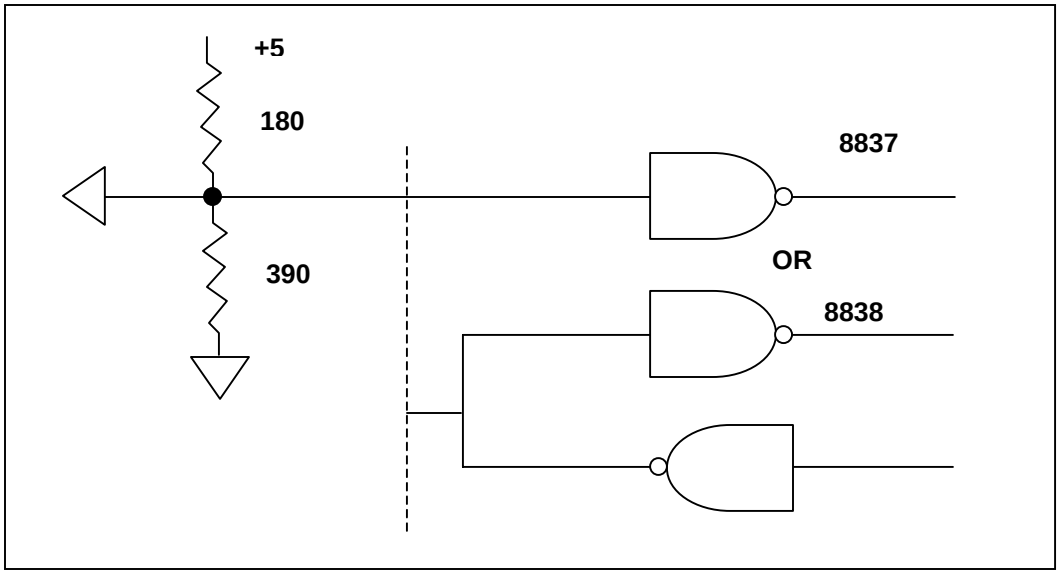
Logic Levels

The PCI 16D uses the DEC Unibus drivers and receivers. These parts have Schmitt trigger inputs and a switching threshold set to equalize high and low noise margins. You can use TTL devices for short distances, but we do not recommend it. The drivers are inverting open-collector drivers, capable of driving the required 123-Ω terminating networks at each cable end.

The recommended parts for the drivers are:

- the National Semiconductor DS8838 Quad Driver/Receiver, and
- the National Semiconductor DS8837 Hex Receiver.

The following figure shows the configuration of the PCI 16D drivers.



Signals

This section describes the kinds of signals the PCI 16D uses, how they are connected, and a suggestion for using the signals in your application for data input and output.

Connector Pinout

The PCI 16D uses a high-density 80-pin I/O connector. The pinout and construction of this connector adapts easily to standard 40-pin, .100 x .100 connectors.

The high-density mating connector is an AMP connector, AMP part number 749111-7, with a straight-shielded backshell (AMP P/N 749196-1) or right angle backshell (AMP P/N 749205-1). The pinout described in the table below ensures that the high density connector and the P1 and P2 connectors of the EDT cable mate directly with standard 40-pin connectors.

Interpret the connector pinout table below in one of the following ways, depending upon the type of cable you are using.

AMP	STD P1	Signal
1	1	DO15
2	2	DO00
3	3	DO14
4	4	DO01
5	5	DO13
6	6	DO02
7	7	DO12
8	8	DO03
9	9	DO11
10	10	DO04
11	11	DO10
12	12	DO05
13	13	DO09
14	14	DO06
15	15	DO08
16	16	DO07
17	17	GROUND
18	18	GROUND
19	19	RSVD IN
20	20	GROUND
21	21	RDSTRBL
22	22	GROUND
23	23	STAUS C
24	24	GROUND
25	25	STATUS C
26	26	GROUND
27	27	STATUS B
28	28	GROUND
29	29	DEVINIT L
30	30	GROUND
31	31	STATUS A
32	32	NC
33	33	RSVD IN
34	34	GROUND
35	35	SDMAEN L
36	36	GROUND
37	37	DMAINPUT L
38	38	GROUND
39	39	DCLKL P
40	40	GROUND

AMP	STD P2	Signal
41	1	DI15
42	2	DI00
43	3	DI14
44	4	DI01
45	5	DI13
46	6	DI02
47	7	DI12
48	8	DI03
49	9	DI11
50	10	DI04
51	11	DI10
52	12	DI05
53	13	DI09
54	14	DI06
55	15	DI08
56	16	DI07
57	17	NC
58	18	GROUND
59	19	GROUND
60	20	GROUND
61	21	OUTVALIDL
62	22	GROUND
63	23	FNCT0
64	24	GROUND
65	25	RSVD IN
66	26	GROUND
67	27	FNCT1
68	28	GROUND
69	29	RSVD IN
70	30	GROUND
71	31	FNCT2
72	32	FNCT2
73	33	RSVD IN
74	34	GROUND
75	35	RSVD IN
76	36	GROUND
77	37	DEVINT L
78	38	GROUND
79	39	DACK P
80	40	GROUND

Table 1. PCI 16D (p16d_3v.bit, p16d_5v.bit)

The following tables describe each signal by name, I/O type, and polarity. An I in the table indicates the signal is an input to the PCI 16D, and an O indicates a PCI 16D output. An H indicates the signal performs the function described in the table at a logic high (or +3 V). An L indicates the signal performs the named function at a logic low. A P indicates the signal is programmable.

Handshake Signals

These five signals perform the PCI 16D transfer cycle.

Name	I/O	Assert	Description
DCLK	I	P	The device asserts DCLK to initiate a transfer. The edge of DCLK is used to transfer data, and the DCLK pulse must be at lieast 80 ns wide. The polarity of the DCLK edge is determined by the CLKP bit in the Configuration register.
DACK	O	P	DACK is the data transfer acknowledge. DACK is asserted by the PCI 16D in response to a DCLK and is negated when the transfer is complete. The polarity of DACK assertion is controlled by the DACKP bit in the Configuration register.
DMAINPUT	O	L	DMAINPUT is asserted low when the PCI 16D is programmed to receive DMA input dat on the DI[0-15] inputs.
OUTVALID	O	L	OUTVALID is asserted low when data from the DO[0-15] outputs is valid. In DMA input mode you can use OUTVALID to qualify outputs for programmed I/O. In DMA output mode, OUTVALID indicates you can clock out valid data with DCLK.
SDMAEN	O	L	SDMAEN is asserted low during the actual DMA transfers on the PCI Local Bus. You can use SDMAEN to qualify DMAINPUT in applications in which the transfer direction changes.
RSTRB	O	L	RSTRB is an 80 ns (minimum) pulse that indicates the PCI Local Bus host has read the DIN signals under program control.

Data Signals

The following table describes the 32 signals that transfer the data. Inputs and outputs are separate for maximum flexibility. The drivers are open-collector, so you can tie the input to the output for a bi-directional bus. For bi-directional operation, write a value of 1 to any output you want to use as a bus input. To use the entire output register as an input bus, write the value FFFF (hexadecimal) to the output register to turn off all output drivers.

Name	I/O	Assert	Description
DIN[0-15]	I	H	Input data
DOUT[0-15]	O	H	Output data

Synchronous Control Signals

The following table describes the six signals controlling the synchronous DMA transfer cycle. Each signal controls a device or indicates device status. Devices can implement these signals as required for device applications.

Name	I/O	Assert	Description
FNCT0	O	H	Output for device control, set with FNCT0 in Command register.
FNCT1	O	H	Output for device control, set with FNCT1 in Command register.
FNCT2	O	H	Output for device control, set with FNCT2 in Command register.
STATA	I	H	Input for device status, read with STATA in Status register.
STATB	I	H	Input for device status, read with STATB in Status register.
STATC	I	H	Input for device status, read with STATC in Status register.

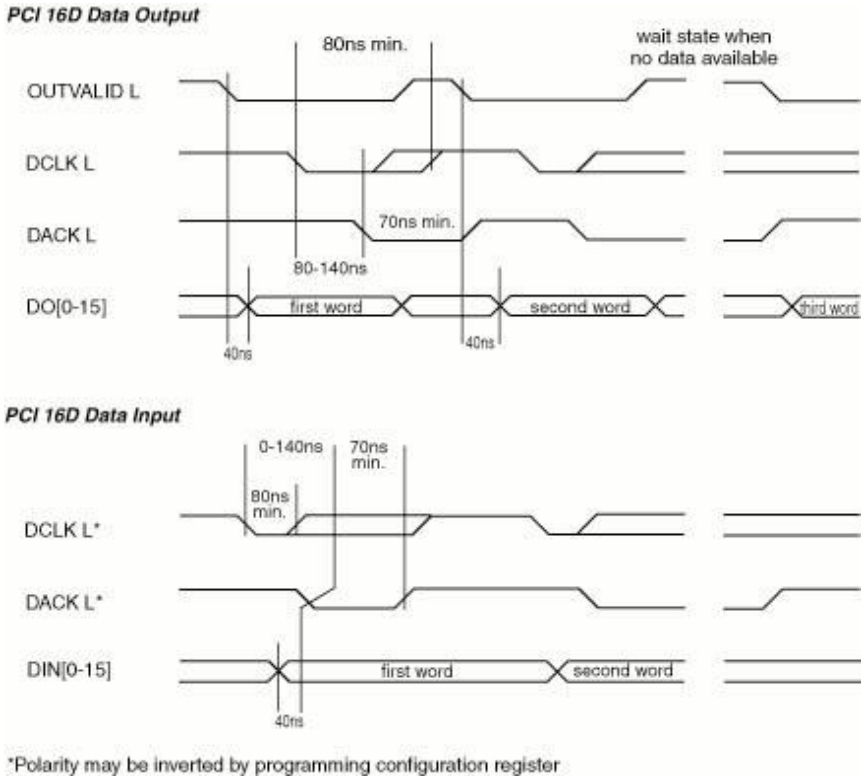
Asynchronous Control Signals

The following table describes the two signals controlling the device and host operating state, asynchronous with the control signals.

Name	I/O	Assert	Description
DEVINT	I	L	Interrupt the PCI Local Bus host if this signal is enabled in the Configuration and Command registers. Sensitive to negative edge.
DEVINIT	O	L	If used, DEVINIT is typically connected in the user device and used to reset the device from the PCI 16D Command register.

Timing

The following diagram shows the timing for the PCI 16D interface. The timing parameters shown in the diagram refer to the PCI 16D during DMA transfers, in response to `read` or `write` system calls.



The following table shows the PCI 16D timing specifications.

Timing Parameter	Minimum	Maximum
DCLK Pulse Width	80 ns	DC
DACK Pulse Width	70 ns	
DCLK Assert to DACK Assert	80 ns	140 ns
OUTVALID Pulse Width	30 ns	
DOUT Valid after DACK false	0	40 ns
DIN Setup to DCLK Assertion		–40 ns

Using the Signal for Data I/O

You can implement an interface hadshake in a programmed logic device. Use the following steps to set up the interface and the properties of the resulting handshake:

1. Pulse DEVINIT low to initialize the interface.
2. Determine the DMA direction from the state of DMAINPUT at the first high-to-low SDMAEN transition.
3. To input data to the PCI 16D, assert the first DCLK as soon as the data is available from the device. The data has a negative setup time with DCLK, so you can output data with DCLK while the device generates the next data.
4. The first output data from the PCI 16D, if any, is valid on the DOUT signals 40 ns after OUTVALID goes low. You can assert DCLK as soon as the device accepts this data.
5. If the input FIFO becomes full, or the output FIFO becomes empty, the DACK signal stays low until the FIFO or data are again available. The OUTVALID signal pulses high approximately 30 ns between data words as long as new data is available. Over long cables, the OUTVALID pulse can be too short to use as a handshake signal; use DACK to qualify data over long cables.
6. Use DINT from the device to direct the PCI 16D to terminate DMA, or use the DINIT or FUNC signals from the PCI 16D to direct the device to terminate transfers.

Registers

The PCI 16D has two memory spaces: the memory-mapped registers and the configuration space. Expansion ROM and I/O space are not implemented.

Applications can access the PCI 16D registers through the DMA library routines `edt_reg_read` or `edt_reg_write` using the name specified under "Access," or if necessary by means of *ioctl( )* calls with PCI 16D-specific parameters, as defined in the file `p16d.h`.

Configuration Space

The configuration space is a 64-byte portion of memory required to configure the PCI Local Bus and to handle errors. Its structure is specified by the PCI Local Bus specification. The structure as implemented for the PCI 16D is as shown in Figure 6 and described below.

Address Bits	31 16		15 0	
0x00	Device ID: 0x01		Vendor ID = 0x123D	
0x04	Status (see below)		Command (see below)	
0x08	Class Code = 0x088000			Revision ID = 0 (will be updated)
0x0C	BIST = 0x00	Header Type = 0x00	Latency Timer (set by OS)	Cache Line Size (set by OS)
0x10	DMA Base Address Register* (set by OS)			
	not implemented			
0x3C	Max_Lat = 0x04	Min_Gnt = 0x04	Interrupt Pin = 0x01	Interrupt Line (set by OS)

Figure 2. Configuration Space Addresses

Values for the status and command fields are shown in the following two tables. For complete descriptions of the bits in the status and command fields, see the *PCI Local Bus Specification*, Revision 2.1, available from:

PCI Special Interest Group
5440 SW Westgate Drive Suite 217
Portland, OR 97221
Phone: 800/433-5177 (United States) or 425/803-1191 (international)
Fax: 503/222-6190

www.pcisig.com

Bit	Name	Value	Bit	Name	Value
0–4	reserved	0	10	DEVSEL Timing	0
5	66 MHz Capable	0	11	Signaled Target Abort	implemented
6	UDF Supported	0	12	Received Target Abort	implemented
7	Fast Back-to-back Capable	0	13	Received Master Abort	implemented
8	Data Parity Error Detected	implemented	14	Signaled System Error	implemented
9	DEVSEL Timing	1	15	Detected Parity Error	implemented

Table 3. Configuration Space Status Field Values

Bit	Name	Value	Bit	Name	Value
0	IO Space	0	6	Parity Error Response	implemented
1	Memory Space	implemented	7	Wait Cycle Control	0
2	Bus Master	implemented	8	SERR# Enable	implemented
3	Special Cycles	0	9	Fast Back-to-back Enable	implemented
4	Memory Write and Invalidate Enable	0	10–15	reserved	0
5	VGA Palette Snoop	0			

Table 4. Configuration Space Command Field Values

PCI Local Bus Addresses

The following figure describes the PCI 16D interface registers in detail. The addresses listed are offsets from the gate array boot ROM base addresses. This base address is initialized by the host operating system at boot time.

Note: The addresses 0x80 and 0x84 are used by the `pciload` utility to update the gate array. User applications must not modify use these registers. Results of running `pciload` do not take effect until after the board has been turned off and then on again.

Address Bits	31 16	15 0
0xC8	data	status
0xC4	configuration	command
0x84	not used	flash ROM data

0x80	flash ROM address
0x20	not used
0x1C	scatter-gather DMA next count and control
0x18	scatter-gather DMA current count and control
0x14	scatter-gather DMA next address
0x10	scatter-gather DMA current address
0x0C	main DMA next count and control
0x08	main DMA current count and control
0x04	main DMA next address

Byte Word	3	2	1	0
	1		0	

Table 5. PCI Local Bus Addresses

Scatter-gather DMA

PCI Direct Memory Access (DMA) devices in Intel-based computers access memory using physical addresses. Because the operating system uses a memory manager to connect the user program to memory, memory pages that appear contiguous to the user program are actually scattered throughout physical memory. Because DMA accesses physical addresses, a DMA read operation must *gather* data from noncontiguous pages, and a write must *scatter* the data back to the appropriate pages. The EDT Product driver uses information from the operating system to accomplish this. The operating system passes the driver a list of the physical addresses for the user program memory pages. With this information, the driver builds a scatter-gather (SG) table, which the DMA device uses sequentially.

Most other PCI computers offer memory management for the PCI bus as well, so the operating system needs to pass only the address and count for DMA. The addresses appear contiguous to the PCI bus.

The scatter-gather DMA list is stored in memory. The scatter-gather DMA channel copies it as required into the main DMA registers. The format of the DMA list in memory is as follows (illustrated in the following table):

- Each page entry takes eight bytes. Therefore, the scatter-gather DMA count is always evenly divisible by eight.
- The first word consists of the 32-bit start address of a memory page.
- The most significant 16 bits of the second word contain control data.
- The least significant 16 bits of the second word contain the count.

As of the current release, only bit 16 contains control information. When set to one, and when enabled by setting bit 28 of the Scatter-gather DMA Next Count and Control register, this bit causes the main DMA interrupt to be set when the marked page is complete.

Bits	63 32	31	16	0
Each entry	address	control (unused)	DMA int	count

Performing DMA

All main DMA registers are read-only. Only the corresponding scatter-gather DMA registers must write to them. To initiate a DMA transfer:

1. Set up one or more scatter-gather DMA lists in host memory, using the format described above and illustrated in the table above.
2. Write the address of the first entry in the list to the Scatter-gather Next DMA Address register.
3. Write the length of the scatter-gather DMA list to the Scatter-gather Next DMA Count and Control register, setting the interrupts as you require. Ensure that bit 29 of this register is set to 1—this starts the DMA.
4. If the DMA list is greater than one page, load the address of the first entry of the next page and its length, as described in steps 2 and 3, when bit 29 of the Scatter-gather Next DMA Count and Control register is asserted.

Main DMA Current Address Register

Size	32-bit
I/O	read-only
Address	0x00
Access	EDT_DMA_CUR_ADDR
Comments	Automatically copied from the main DMA next address register after main DMA completes.

Bit	Description
A31–0	The address of the current DMA or the last used address if no DMA is currently active.

Main DMA Next Address Register

Size	32-bit
I/O	read-only
Address	0x04 + (channel number x 20 hex)
Access	EDT_DMA_NXT_ADDR

Comments	The scatter-gather DMA fills this register when required from the scatter-gather DMA list.

Bit	Description
A31–0	Read the starting address of the next DMA.

Main DMA Current Count and Control Register

Size	32-bit
I/O	read-only
Address	0x08
Access	EDT_DMA_CUR_CNT
Comments	This register automatically copied from the main DMA next count and control register after main DMA completes.

Bit	Description
A31–16	Read-only versions of bits 31–16 of the scatter-gather DMA current count and control register.
D15–0	The number of words still to be transferred in the current DMA.

Main DMA Next Count and Control Register

Size	32-bit
I/O	read-only
Address	0x0C
Access	EDT_DMA_NXT_CNT
Comments	The scatter-gather DMA fills this register when required from the scatter-gather DMA list.

Bit	Description
A31–16	Read-only versions of bits 31–16 of the scatter-gather DMA next count and control register.
D15–0	The number of words still to be transferred in the current DMA.

Scatter-gather DMA Current Address Register

Size	32-bit
I/O	read-only
Address	0x10
Access	EDT_SG_CUR_ADDR
Comments	Automatically copied from the scatter-gather DMA next address register when that register is valid and the current scatter-gather DMA completes.

Bit	Description
A31–0	The address of the current DMA or the last used address if no DMA is currently active.

Scatter-gather DMA Next Address Register

Size	32-bit
I/O	read-write
Address	0x14
Access	EDT_SG_NXT_ADDR
Comments	The driver software writes this register as described in step 2 of the list in the Performing DMA section on page **Error! Bookmark not defined.**.

Bit	Description
A31–0	The starting address of the next DMA.

Scatter-gather DMA Current Count and Control Register

Size	32-bit
I/O	read-only
Address	0x18
Access	EDT_SG_CUR_CNT
Comments	The driver software can read this register for debugging or to monitor DMA progress.

Bit	Description
A31–16	Read-only versions of bits 31–16 of the scatter-gather DMA next count and control register.
D15–0	The number of words still to be transferred in the current DMA.

Scatter-gather DMA Next Count and Control Register

Size	32-bit
I/O	read-write
Address	0x1C
Access	EDT_SG_NXT_CNT
Comments	The driver software writes this register as described in step 2 of the list in the Performing DMA section on page 63.

Bit	EDT_	Description
D31	EN_RDY	Enable scatter-gather next empty interrupt. A value of 1 enables DMA_START (bit 29 of this register) to set DMA_INT (bit 12 of the Status register), thus causing an interrupt if the PCI_EN_INTR bit is set (bit 15 of the Main DMA Command and Configuration register). A value of 0 disables the DMA_START from causing an interrupt.
D30	DMA_DONE	Read-only: a value of 0 indicates that a scatter-gather DMA transfer is currently in progress. A value of 1 indicates that the current scatter-gather DMA is complete.
D29	DMA_START	Write a 1 to this bit to indicate that the values of this register and the SG DMA Next Address register are valid; this sets this bit to 0, indicating either that the copy is in progress, or that the device is waiting for the current DMA to complete. In either case, this register and the SG DMA Next Address register are not available for writing. Reading a value of 1 indicates that the SG DMA Next

		Count and SG DMA Next Address registers have been copied into the SG DMA Current Count and SG DMA Current Address registers and that the Next Count and Next Address registers are once more available for writing.
D28	EN_MN_DONE	A value of 1 enables the main DMA page done interrupt (bit 18).
D27	EN_SG_DONE	Enable scatter-gather DMA done interrupt. A value of 1 enables DMA_DONE (bit 30 of this register) to set DMA_INT (bit 12 of the Status register), thus causing an interrupt if the PCI_EN_INTR bit is set (bit 15 of the Main DMA Command and Configuration register). A value of 0 disables the DMA_DONE from causing an interrupt.
D26	DMA_ABORT	A value of 1 stops the DMA transfer in progress and cancels the next one, clearing bits 29 and 30. Always 0 when read.
D25	DMA_MEM_RD	A value of 1 specifies a read operation; 0 specifies write.
D24	BURST_EN	A value of 0 means bytes are written to memory as soon as they are received. A value of 1 means bytes are saved to write the most efficient number at once.
D23	MN_DMA_DONE	Read only: a value of 1 indicates that the main DMA is not active.
D22	MN_NXT_EMP	Read only: a value of 1 indicates that the main DMA next address and next count registers are empty.
D21–19		Reserved for EDT internal use.
D18	PG_INT	Read-only: a value of 1 indicates that the page interrupt is set (enabled by bit 28 of this register), and that the main DMA has completed transferring a page for which bit 16 (the page interrupt bit) was set in the scatter-gather DMA list (see Figure 6). If the PCI interrupt is enabled (bit 15 of the PCI interrupt and remote Xilinx configuration register), this bit causes a PCI interrupt. Clear this bit by disabling the page done interrupt (bit 28 of this register).
D17	CURPG_INT	Read-only: a value of 1 indicates that bit 16, the page interrupt bit, was set in the scatter-gather DMA list entry for the current main DMA page.
D16	NXTPG_INT	Read-only: a value of 1 indicates that bit 16, the page interrupt bit, was set in the scatter-gather DMA list entry for the next main DMA page.
D15–0		The number of bytes in the next scatter-gather DMA list.

Flash ROM Access Registers

Flash ROM Address Register

Size	32-bit
I/O	read-write
Address	0x80
Access	EDT_FLASHROM_ADDR
Comment	Use this register and the flash ROM data register (below) to update the program in the field-programmable gate array that implements the PCI interface.

Bit	Description
D31–25	Reserved for EDT internal use.
D24	A value of 1 causes the data in the flash ROM data register to be written to the address specified by bits 0 through 23. A value of 0 reads the data.
D23-0	Address of location in flash ROM that the next read or write will access.

Flash ROM Data Register

Size	32-bit
I/O	read-write
Address	0x84
Access	EDT_FLASHROM_DATA
Comment	Use this register and the flash ROM address register (above) to update the program in the field-programmable gate array that implements the PCI interface.

Bit	Description
D31–9	Not used
D8	A read-only bit indicating the position of the jumper that enables access to the protected area of the ROM that contains the executable program. A value of 1 indicates that the board can load a new program.
D7-0	The new program to load into flash ROM with a write operation (specified by setting bit A24 in the flash ROM address register), or the data that was read (specified by clearing bit A24 in the flash ROM address register).

Device Control Registers

Command Register

Size	16-bit
I/O	read-write
Address	0xC4
Access	P16_COMMAND

Bit	P16D_	Description
D15	EN_INT	Enable PCI interrupt, to set the interrupt on a board-wide basis. Write 1 to set the interrupt; write 0 to clear.
D14		Not used.
D13	EN_DINT	Enable device interrupt. Write 0 to clear the interrupt.
D12	FCLK	Force Clock. Set FCLK to produce the same effect as an external cycle request (DCLK); it is not necessary to clear FCLK afterward.
D11-10		Not used.
D9	ODDSTART	Set to 1 when DMA starts on an odd-word boundary. In order for ODDSTART to behave correctly, first clear the FIFOs with BCLR.
D8	BCLR	Board Clear. Set BCLR to abort the DMA in process and clear the PCI 16D FIFOs, provided DFRST in the Configuration register has not been set todesable a FIFO reset. This operation does not send INIT to the device. This value is not stored and need not be reset.
D7	INIT	Initialize device. The PCI 16D asserts the DEVINIT signal on the interface as long as INIT is set to 1. The DEVINIT signal is not a pulse; the software determines the length of DEVINIT according to the requirements of the device.
D6-4	FNCT3 FNCT2 FNCT1	These function control bits are passed directly to the device interface. They are true when p ositive; write a 1 to these bits to see a logic high on the cable. When the loopback connector is installed, FNCT2 is connected to the device interrupt input.
D3		Not used.
D2	FLUSH	FLUSH is set to 1 when a DMA transfer is aborted and the register fi8les still contain data awaiting a burst transfer. FLUSH causes a final 16-byte transfer to the current DMA buffer. This value is not stored and need not be reset.
D1	DIR	Select the direction of the DMA transfer. DIR is set to 1 when the PCI 16D writes DMA data. When the loopback connector is installed, toggling this bit changes the DEVINT

signal.

D0	Not used.

Configuration Register

Size	16-bit
I/O	read-write
Address	0xC6
Access	P16_CONFIG

Bit	P11W_	Description
D15-11		Not used.
D10	SSWAP	When set to 1, Short SWAP swaps 16-bit words within a 32-bit memory word, for hosts that require this reordering.
D9	INV	When set to 1, inverts the data.
D8	SWAP	SWAP determines which byte of PCI Local Bus half-word ends up on which half of the PCI 16D 16-bit bus. When SWAP is 0, byte swapping is disabled; this is the default—the upper byte of an Sbus half word appears on the upper half of the PCI 16D 16-bit bus.
D7-4		Not used.
D3	DFRST	Set DFRST to disable a FIFO reset. When set to 1, DFRST prevents BCLR from resetting and emptying the FIFO.
D2	DBMODE	Set DBMODE to disable Sbus burst DMA transfers. Use DBMODE for applications in which data must be immediately transferred to memory.
D1	CLKP	CLKP, the data clock polarity bit, determines which edge of the DCLK inpu7t initiates the transfer. A 0 on CLKP requires a negative edge to initiate the transfer; this is the default. A 1 on CLKP requires a positive DCLK edge to start a tranfer. We recommend using the negative edge to initiate transfers, so that unplugging the device cable does not produce a clock edge.
D0	DACKP	DACKP determines the polarity of the Data Acknowledge signal. If DACKP is zero, the external DACK is asserted low (negative is true); this is the default. If DACKP is set, the external DACK is asserted high (positive is true). If you remove the hardware jumper on the PCI 16D, the DACK polarity is forced to positive true, and DACKP is ignored. Use the jumper for devices that require a positive value to be true DACK from powerup. We recommend the negative is true setting for DACK, so that DACK is not asserted when the cable is unplugged from the device.

Status Register

Size	16-bit
I/O	read-only
Address	0xC8
Access	P16_STATUS

Bit	P16D_	Description
D15	INT	A value of 1 indicates the PCI interrupt is asserted.
D14		Not used.
D13	DEVINT	Shows when a device interrupt is pending. DEVINT is set to show a pending device interrupt when the external DINT input is asserted (low) and latched, and the ENDINT bit is set high in the Configuration register. DEVINT causes a PCI bus interrupt when EN_INT (bit 15 of the Command register) is 1, which enables interrupts. Clear DEVINT by setting EN_DINT (bit 13 of the Command registers) to 0, and then to 1.
D12	DMA_INT	A value of 1 indicates the end of DMA interrupt is asserted.
D11	DINT_S	Reflects the state of the external PCI 16D DINT (Device Interrupt) input. DINT is not latched.
D10-8	STATC_S STATB_S STATA_S	Reflects the state of the external PCI 16D inputs STATA, STATB, and STATC.
D7	DINIT_S	Reads back the state of the Command register INIT bit.
D6-4	FNCT3_S FNCT2_S FNCT1_S	Reads back the state of Command register bits FNCT3, FNCT2, and FNCT1.
D3	ENEOD_S	End of DMA interrupt enabled; reads back the ones complement of the ENEOD bits in the Configuration register.
D2	ENDINT_S	Device interrupt enabled; reads back the ones complement of the ENDINT bits in the Configuration register.
D1	DIR_S	Reads back the ones complement of the DIR bit in the Command register.
D0		Not used.

Data Register

Size	16-bit
I/O	read-write
Address	0xCA
Access	P16_DATA
Comments	Reading this register yields the current contents of the PCI 16D input data bus.

Bit	Description
D15-0	When DMA is in progress, DMA writes to this register; otherwise, data written to this register appears on the PCI 16D output data bus to this register.

Specifications

PCI Local Bus Compliance	
Number of Slots	1
Transfer Size	1, 2, 4, and 16 bytes
DVMA master	Yes
PCI Local Bus width	32 bits
Clock rate	16 MHz to 25 MHz
Device Data Transfer	
Format	16-bit parallel word
Handshake	2-wire asynchronous handshake
Transfer types	Input or output
Signal polarity	Data is true. Control signals are programmable; the default is true when zero.
Signals	16 data inputs, 16 data outputs 3 status inputs, 3 control outputs, 1 device interrupt
Buffers	128-byte FIFO for input, 128-byte FIFO for output
Software	Drivers for Solaris 2.6+ (Intel and SPARC platforms), Windows NT/2000/XP Version 4.0, AIX Version 4.3, IRex 6.5, and Linux Red Hat Version 5.1
Power	5V at 1.5A
Environmental	
Temperature	Operating: 10 to 40° C Nonoperating: -20 to 60° C
Humidity	Operating: 20 to 80% noncondensing at 40° C Nonoperating: 95% noncondensing at 40°C
Physical	
Dimensions	3.3” x 5.78” x 0.5”
Weight	6 oz

References

See the UNIX manual pages for `ioctl(2)`, `select(2)`, `read(2)`, `open(2)`, `write(2)`, `aioread(2)`, `aiowrite(2)`, and `aiowait(2)` for specific information about these system calls.

See the Solaris Installation documents for more information about configuring the kernel for asynchronous I/O.

PCI Local Bus Specification, Revision 2.1, 1995, is available from:

PCI Special Interest Group
P.O. Box 14070
Portland, OR 97214

Phone:
800/433-5177 (United States)
503/797-4207 (International)

Fax: 503/234-6762

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP