# PCI CD and PCI CDa

**Revision: I**
**December 2004**

## Control Information

| Control Item | Details |
|---|---|
| Document Owner | Mark Mason/Dan Boer |
| Information Label | EDT Public |
| Supersedes | None |
| File Location | frm:/pcicd/pcicd.doc |
| Document Number | 008-00965-06 |

## Revision History

| Revision | Date | Revision Description | Originator |
|---|---|---|---|
| Draft | 20-Aug-03 | Convert from FrameMaker to Word | S Vasil |
| A | 11-Jan-04 | Added Input/Output section; general cleanup | S Vasil |
| B | 18-Mar-04 | Updated “Upgrading the Firmware” section. Added CDa pinout. | R Henderson<br>D Boer |
| C | 29-Mar-04 | Fixed PCI CD pinout (pcd8_src.bit) | D Boer |
| D | 19-Apr-04 | Deleted Output Disable on page 64 | D Boer |
| E | 21-Apr-04 | Fixed PCI CDa pinouts | D Boer |
| F | 05-May-04 | Added PIO registers, 0x08 and 0x09 | D Boer |
| G | 09-Sep-04 | Fixed pinout | D Boer |
| H | 07-Oct-04 | Updated Input/Output section | M Mason |
| I | 16-Dec-04 | Minor fixes; added line drawing for jumpers | 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.

# Contents

**Overview ..... 5**
Features ..... 5
**Installation ..... 6**
**Verifying the Installation ..... 7**
Configuring the PCI CD ..... 7
Configuration Utility: initpcd ..... 8
PCI CD ..... 9
PCI CDa ..... 9
Building the Sample Programs ..... 9
Uninstalling ..... 9
Upgrading the Firmware ..... 10
**Input and Output ..... 12**
Elements of EDT Interface Applications ..... 12
FIFO Flushing ..... 14
Multi-Threaded Programming ..... 14
DMA Library Routines ..... 15
EDT Message Handler Library ..... 49
Message Definitions ..... 49
Files ..... 50
**PCI CD Output Clock Generation ..... 55**
**PCI CDa Output Clock Generation ..... 59**
**Hardware Interface Protocol ..... 61**
Electrical Interface ..... 61
Signals ..... 63
Timing ..... 64
**Connector Pinout ..... 65**
**Registers ..... 71**
Configuration Space ..... 71
PCI Local Bus Addresses ..... 72
Scatter-gather DMA ..... 73
Performing DMA ..... 74
Flash ROM Access Registers ..... 79
Interrupt Registers ..... 81
Remote Xilinx Registers ..... 82
**Specifications ..... 91**

# Overview

The PCI Bus Configurable DMA (PCI CD) board 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. This interface is typically used to move data between a PCI bus host computer and devices such as scanners, plotters, imaging devices, or research prototypes. The PCI CD uses a simple synchronous protocol for transferring data.

The CDa is very similar to the PCI CD, with the following differences:

- The PCI CDa uses the PCI SS-style phase-locked loop (PLL) and DMA engine.
- The PCI CDa comes standard with an XC2S100e user interface Xilinx; an XC2S600e Xilinx is optional.
- The PCI CD I/O is either RS422 or LVDS.
- FIFOs are internal to the Xilinx FPGA.
- The PCI CDa will operate in 66 MHz PCI slots with transfer rates up to 210 megabytes per second. It uses the same host software as the PCI CD.

## Features

- The PCI CD/CDa 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 that enable applications to access the PCI CD/CDa and transfer data continuously or in bursts across the PCI CD/CDa interface using EDT library calls.
- The interface is synchronous, meaning all data and control signals are transmitted with a clock signal. Either the PCI CD/CDa interface or the external device can generate receive or transmit timing, or each can generate its own transmit timing.
- The PCI CD/CDa RS422 oscillator operates at 10 MHz or 20 megabytes per second. The PLL can be selected as the clock source, which can be programmed at various speeds. The transfer rate is limited by the RS422 interface.
- The PCI CD PECL (PCI CD-40) oscillator operates at 20 MHz or 40 megabytes per second. The PLL can be selected as the clock source, which can be programmed at various speeds. The transfer rate is limited by the PECL interface.
- The PCI CD/CDa LVDS oscillator operates at 30 MHz or 60 megabytes per second. The clock operates at various speeds depending on programming. The transfer rate is limited by the LVDS interface.
- The DMA Engine on all PCI CD boards is up to 80 megabytes per second. The DMA engine on all CDa boards is up to 210 megabytes per second.
- All CD/CDa boards use 4 KB input and output FIFOs, and all signals are differential.
- The input and output FIFO buffers smooth data transfer between the PCI bus and the user device, accommodating data during the transition from one DMA to the next. DMA transfers are queued in hardware, minimizing the amount of FIFO required.

This manual describes the operation of the PCI CD/CDa with the UNIX-based and Windows operating systems.

# Installation

If you are using a Dell computer, you should be aware that for some models, Dell recommends high data rate cards (such as video and frame grabbers) be placed in one of the first two slots (closest to the AGP connector). The other two PCI bus slots are only recommended for lower speed devices such as audio devices or modems.

**Caution:** Be sure to use appropriate static protection to prevent damaging your PCI CD/CDa.

Other computer manufacturers may have similar requirements as Dell. Consult your computer manufacturer's documentation for more information.

After installing the PCI CD/CDa, verify the installation, configure the device, and build the sample programs (optional). Instructions for uninstalling the software or upgrading the firmware are on page 9.

For complete instructions regarding PCI board installation, see your computer manufacturer’s manual.

# Verifying the Installation

To verify that installation was successful and that the PCI CD/CDa is operating correctly:

1. Run `Pcicd Utilities` (Windows NT) or cd to `/opt/EDTpcd` (UNIX).
2. At the command prompt, enter:

```
xtest 4096
```

**Outcome:** The PCI CD/CDa returns test status information. You will be prompted to press **Return** at certain steps. The following is an example of typical behavior, although details will vary:

```
reading 4096 words
buf at 820000
testing dirreg at 4 4
testing dirout at 8 8
testing dirin at 8 c
testing ctlout at a a
testing ctlin at a e
Calling DMA read 8192 at 820000
return to do read:
read returned length 8192
Done.
checking data
4096 words 0 errors
buf 0 820000
buf 1 920000
reading 100 buffers of 1048576 bytes from unit 0 with 2 bufs
return to start: starting read at 820000
starting read at 920000
hit return to continue:
counter0 4628 2362958141 counter1 4628 3122437420 freq 0    266230000
dtime 759479279.000000 ticktime 266230000.000000
time is 2.852719 sec
36757077.671371 bytes/sec
```

## Configuring the PCI CD

The PCI CD/CDa can be configured to run in several modes. Before running the device, decide which of these modes is appropriate for your application and configure it accordingly. Configure the PCI CD/CDa by downloading a bitfile to the Xilinx field-programmable gate array (FPGA).

To configure the PCI CD/CDa:

1. Run `Pcicd Utilities` (Windows NT) or cd to `/opt/EDTpcd` (UNIX).
2. At the command prompt, enter:

```
pcdrequest
```

3. Read the description of the signals and options.
4. Enter the associated option number (UNIX-based systems), or click the radio button next to the desired mode of operation and then click **OK** (Windows NT systems).

The `pcdrequest` command creates a script or batch file `pcdload` that contains the commands needed to download the appropriate bitfile into the FPGA. The `pcdload` command runs immediately, as well as whenever the computer is rebooted.

To reselect the default Xilinx bitfile at a later time, rerun `pcdrequest`, or edit the `pcdload` script file by hand.

**Note:** `xtest` downloads its own test bitfile automatically; after running `xtest`, run `pcdload` to reload the default bitfile.

The PCI CD/CDa boards are synchronous interfaces—they send a clock signal with all data and control signals. The PCI CD stores inputs only at the rising edge of the receive timing (RXT) signal that comes from the user device. The user device stores outputs from the PCI CD only at the rising edge of the transmit timing (TXT) signal. The PCI CD/CDa always outputs the TXT signal, but the internal source of the signal can be either the RXT from the user device or an internal oscillator on PLL in the PCI CD/CDa. If an internal oscillator is used, its rate is 10 MHz for the PCI CD/CDa RS422, 20 MHz for the PCI CD PECL, and 30 MHz for the PCI CD/CDa LVDS.

## Configuration Utility: initpcd

EDT supplies a board configuration utility with your PCD driver called `initpcd`. This utility inputs a simple text configuration file to set up the PCI CD/CDa boards, as well as any other board that uses the PCD driver. Use `initpcd` to reliably and consistently configure your PCD boards and to eliminate messy configuration code from your applications.

Complete documentation of the `initpcd` command set is included as the first comment in the source file `initpcd.c`. Several .cfg files are also included in the PCD driver directory as examples.

A typcial initpcd .cfg script performs the following operations:

- Loads a specified firmware bitfile.
- Sets clock speeds on available PLL clocks.
- Sets any configuration registers, such as channel enable and direction bits, DMA mapping registers, and any other arbitrary register, including customer-defined registers.
- Sets bit order, byte order, and short word order supported in the EDT firmware.
- Flushes the user interface FIFO to initialize the user interface firmware.

The following is an example of a simple PCI CDa configuration file:

```
bitfile: pcda.bit

# Enable interface Xilinx
command_reg:     0x08

# Enable and set the onboard PLL clock generator to 30MHz
funct_reg:       0x80
run_command:     set_ss_vco -F 30000000.0 0

# Alternatively setting interface register 0x0f to 0x02 sends
RXT to TXT.
intfc_reg:       0x0f 0x00
#intfc_reg:      0x0f 0x02

# Byteswap and shortswap usually 1 for Sun, 0 for Intel.
byteswap_sun:    1
shortswap_sun:   1
byteswap_x86:    0
shortswap_x86:   0

flush_fifo:      1
```

## PCI CD

You must choose the source for the TXT signal when the driver is loaded. We recommend using the RXT input for TXT timing. Even if the PCI CD internal oscillator is used to time the user device, the user device can use the PCI CD signal SENDT (send timing) to drive device outputs. It can then be sent back as the RXT signal.

For most installations, choose either external clock with TXT looped from RXT, or internal clock from the PCI CD. For the SSD4 option, choose from one, two, or four input channels, or one input and one output channel.

## PCI CDa

PCI CDa currently has only a 16-bit parallel interface and a 16-channel synchronous serial interface.

# Building the Sample Programs

### UNIX-based Systems

To build any of the example programs on UNIX-based systems, 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 NT Systems

To build any of the example programs on Windows NT systems:

1. Run `pcicd 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++.

# Uninstalling

### Solaris Systems

To remove the PCI CD/CDa driver on Solaris systems:

1. Become root or superuser.
2. Enter:

```
pkgrm EDTpcd
```

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

### Linux Systems

To remove the PCI CD/CDa driver on Linux systems:

1. Enter: `cd /opt/EDTpcd`
2. Enter: `make unload`
3. Enter: `cd /`
4. Enter: `rm -rf /opt/EDTpcd`

### Windows NT Systems

To remove the PCI CD/CDa toolkit on Windows NT systems, use the Windows NT Add/Remove utility. For further details, consult your Windows NT 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/EDTpcd*; for Windows, usually *C:\EDT\pcd*).
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
```

# Input and Output

The PCI CD/CDa 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 of buffers (usually four of 1 MB) 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.

# FIFO Flushing

First-in, first-out (FIFO) memory buffers are used to smooth data transmission between different types of data sinks internal to PCI DV boards. For instance, the FIFO stores information processed by the user interface Xilinx until the PCI Xilinx retrieves it across the PCI bus. The PCI bus normally sends information in bursts, so the FIFO allows this same information to be sent smoothly.

When acquiring or sending data, you should flush the FIFO immediately before performing DMA. This also resets the FIFO to an empty state. The following subroutines either flush the FIFO or set it to flush automatically at the start of DMA. Complete descriptions are available on page 38.

### void edt_flush_fifo (EDTDev *edt_p)

This routine flushes the global board FIFO. It is called just before the first call to `edt_start_buffers`. It affects all channels on a multi-channel board. See `edt_flush_channel` for per-channel FIFO flushing. Use caution when calling edt_flush_fifo immediately before edt_read or edt_write, because there is a delay allocating kernel DMA resources during thich the FIFO may overflow with data. See edt_set_firstflush below.

### int edt_set_firstflush (EDTDev *edt_p, int flag)

This routine performs a global FIFO flush just before DMA starts in the driver when the flag is nonzero. This routine is useful when using `edt_read` and `edt_write`. The default setting is OFF.

### void edt_flush_channel (EDTDev *edt_p)

This routine flushes only the DMA channel associated with `edt_p`. It requires firmware support from the user interface Xilinx and is not implemented on all firmware configurations.

### void pcd_pio_flush_fifo (EDTDev *edt_p)

After calling pcd_pio_init(EdtDev *edt_p), this routine may be used to flush the global board FIFO by using memory-mapped register access instead of using a system call to the driver. This method is slightly faster.

# Multi-Threaded Programming

EDT recognizes there are many ways to write multi-threaded programs that work, so the following guidelines are provided to help with your initial programming efforts or as a means of troubleshooting applications that use EDT boards.

The EDT driver is thread-safe with the following constraints:

1. All DMA operations must be performed in the same thread as edt_open and edt_close with respect to each channel. Other threads may open the same channel concurrently with DMA, but should perform no DMA-related operations. This is because kernel DMA resources are allocated on a per-thread basis and must be allocated and released in the same thread.

2. When exiting the program, one of the following two conditions must be met before the EDT driver is used again:

- All threads spawned by a main program must be joined with the main program after they exit and before the main program exits; or
- If the main program does not wait for the child threads to exit, then any program that is run following the main program must wait for all the child threads to exit. This waiting period depends on system load and availability of certain system resources, such as a hardware memory management unit.

Failure to meet one of these conditions results in undefined program and system behavior and program and system crashes can occur.

# 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. Table 1, DMA Library Routines, 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. |

| Routine | Description |
|---|---|
| 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(). |
| 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. |

| Routine | Description |
|---|---|
| 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 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 susubroutine 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:<br>EDT_READ = 0<br>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 *edt_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 32.)

***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 37.

***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 11W |
| EDT_EVENT_P11W_CNT | Count interrupt occurred | PCI 11W |
| 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 39. |

***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 by 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_msleep(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 NT 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 47 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

# PCI CD Output Clock Generation

The output clock is generated from a phase-locked loop (PLL) oscillator, a reference crystal, and programmable dividers. Because each of these components has physical limits to its operation, it may not be possible to get exactly the frequency desired. To get the expected results, you need to understand how the clock generator operates. Figure 1 diagrams how the final value is generated.

***Figure 1 Legend***

Frequency values:

| | |
|---|---|
| $f_{xtal}$ | PCI CD-20 is 10 MHz, PCI CD-40 is 20 MHz, and PCI CD-60 is 30 MHz. |
| $f_{ref}$ | The PLL reference frequency must be between 200 KHz and 5.0 MHz. |
| $f_{vco}$ | The VCO output frequency must be between 50 MHz and 250 MHz. |
| $f_{fback}$ | The VCO varies $f_{vco}$ until the feedback frequency matches the PLL reference frequency. |
| $f_{xilinx}$ | The input frequency into the high speed odd divider must be less than 100 MHz. |
| $f_{low}$ | The divide by *n* counter input frequency must be less than 30 MHz. If L and X are both set to 1, then frequencies to 100 MHz may be passed to the divide by 2. |
| $f_{out}$ | This final divide by 2 assures a 50% output clock duty cycle. |

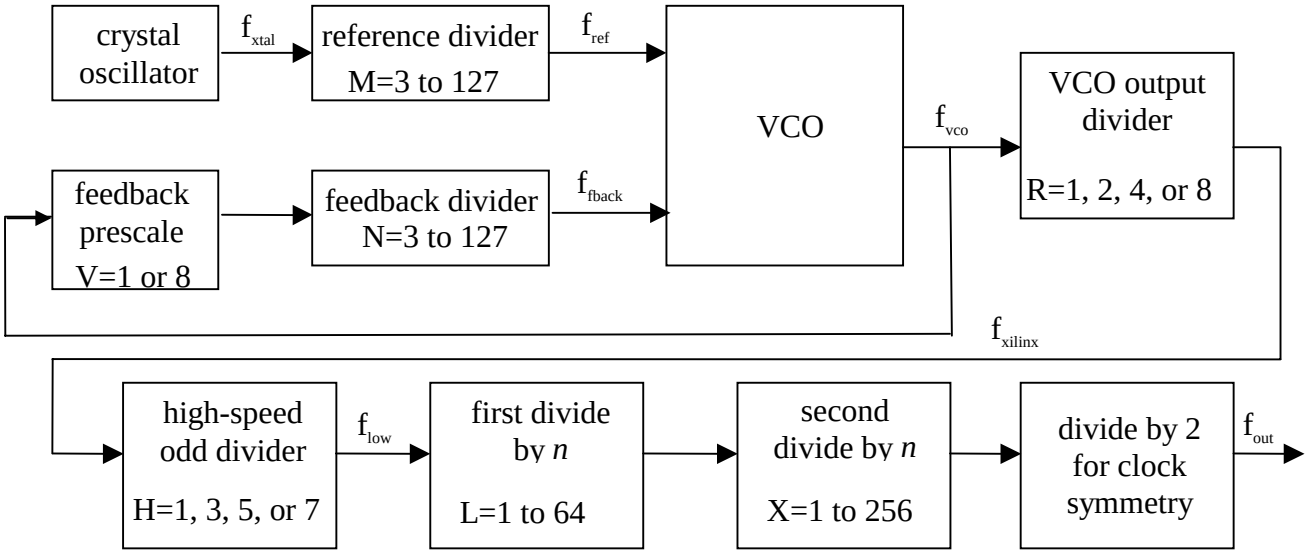



**Figure 1. Output Clock Generation Block Diagram**

The formula for calculating $f_{out}$ is:

$$f_{out} = (N * V * f_{xtal})/(m * R * H * L * X * 2)$$

For example, to achieve 15 MHz using a PCI CD-20 ($f_{xtal}$ = 10 MHz), the following numbers work (although they are not unique):

N=60, V=1, M=10, R=2, H=1, L=1, X=1

To achieve 125 Hz in a PCI CD-20 ($f_{xtal}$ = 10 MHz), the following numbers work (although they are not unique):

N=50, V=1, M=10, R=8, H=5, L=50, X=100

The output clock has a wide range of values, but the frequency limitations at different stages limits the ultimate ability to exactly hit any specific frequency.

An example, the PLL reference frequency can be as low as 200 KHz: that would seem to allow steps of 200,000 in the VCO output. Unfortunately, since the maximum VCO output is 50 MHz and the *n*programmable divider only goes to 127, the loop cannot lock unless V is set to 8, giving 1.6 MHz minimum steps. Now if R is set to 8, we *can* get 200 KHz steps at $f_{xilinx}$. The lowest frequency in this case is at N=32 (6.4 MHz) to N=127 (25.4 MHz), in 200 KHz steps.

The following three library routines help compute the output clock frequency:

| Routine | Description |
|---|---|
| edt_find_vco_frequency | Computes the phase-locked loop parameters necessary to match (or approximate) the supplied target frequency. |
| edt_set_pll_clock | Sets the phase-locked loop circuit to the value computed by edt_find_vco_frequency. Includes debugging information. |
| edt_set_out_clock | Sets the phase-locked loop circuit to the value computed by edt_find_vco_frequency. Does not include debugging information. |

**Table 1. Output Clock Generation Library Routines**

## edt_find_vco_frequency

### Description

Computes the phase-lock loop parameters described in Figure 1 that are necessary to match or closely approximate the supplied target frequency, and builds the edt_pll structure with these computed values. Because of the hardware constraints described in the Legend to Figure 1, the target frequency may not be possible. This routine returns the actual frequency calculated.

**Note** This routine computes the necessary values but does not set the phase-locked loop circuit. To set the circuit, use edt_set_out_clock or edt_set_pll_clock, described next.

### Syntax

```
#include "edtinc.h"

double edt_find_vco_frequency (EdtDev *edt_p, double target,
double xtal, edt_pll *pll, int verbose);
```

### Arguments

| | |
|---|---|
| *edt_p* | device handle returned from edt_open |
| *target* | desired output frequency in Hz |
| *xtal* | base frequency of the PCI CD board:<br>PCI CD use XTAL20<br>PCI CD use XTAL60 |
| *Verbose* | a value of 1 prints a description to stdout, useful for debugging ; a value of 0 turns off verbose output |

### Return

The actual frequency, in Hertz, computed for the edt_pll structure.

## edt_set_pll_clock

### Description

Sets the phase-locked loop circuit to the parameters described in Figure  and computed using `edt_find_vco_frequency`, above. This routine enables the phase-locked loop by setting the appropriate bits in the FUNCT register, and then calls `edt_set_out_clock` to set the circuit as required.

### Syntax

```
#include "edtinc.h"
void edt_set_pll_clock(EdtDev *edt_p, double xtal, edt_pll
*pll, int verbose);
```

### Arguments

| | |
|---|---|
| *edt_p* | device handle returned from *edt_open* |
| *xtal* | base frequency of PCI CD/CDa board:<br>PCI CD-20 use `XTAL20`<br>PCI CD-60 use `XTAL60` |
| *pll* | the structure containing the values necessary to set the phase-locked loop circuit, as described in Figure 1. |
| *verbose* | a value of 1 prints a description to *stdout*, useful for debugging; a value of 0 turns off verbose output |

### Return

None.

## edt_set_out_clock

### Description

Sets the phase-locked loop circuit to the parameters described in Figure 1 and computed using `edt_find_vco_frequency`.

### Syntax

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

```
void edt_set_out_clock(EdtDev *edt_p, edt_pll *pll);
```

### Arguments

*edt_p* device handle returned from `edt_open`.

*pll* the structure containing the values necessary to set the phase-locked loop circuit, as described in Figure 1.

### Return

None.

# PCI CDa Output Clock Generation

The PCI CDa has one programmable clock with a range of 168 Hz to 100 MHz. Most frequencies between these extremes can be achieved with little error. (Some FPGA bitfiles may allow an extended range of up to 200 MHz.)

The programmable clock has an ICS307-02 clock generator (from Integrated Circuit Systems, www.icst.com) followed by a 14-bit programmable divider. The reference clock to the ICS307-02 is 10.3861 MHz.

**Note:** Seee `set_ss_vco.c` as an example.

The following three routines provided by EDT handle the clock generators:

## edt_find_vco_frequency_ics307

### Description

Computes the PLL parameter for the ICS307 chip, based on an input clock frequency (`xtal`) and a target frequency (`target`). The EdtDev pointer is not required; it can be set to `NULL`. If the `xtal` value is 0, there should be an EdtDev pointer that can be used to determine the reference clock frequency.

The `nodivide` version turns off the final divide by 2 in the FPGA code; if the current bitfile supports that, frequencies greater than 100 MHz can be targeted.

### Syntax

```
#include "edtinc.h"

#include "edt_ss_vco.h"

double edt_find_vco_frequency_ics307(EdtDev *edt_p, double
target, double xtal, edt_pll *pll, int verbose)

double edt_find_vco_frequency_ics307_nodivide(EdtDev *edt_p,
double target, double xtal, edt_pll *pll, int verbose)
```

### Arguments

| | |
|---|---|
| *edt_p* | Device handle returned from `edt_open`. |
| *target* | Desired output frequency in Hz. |
| *xtal* | The base frequency of the PCI SS board. Default is 10.3681 MHz. |
| *verbose* | A value of 1 prints a summary of the results to `stdout`. A value of 0 turns off output. |

### Return

The return value is the actual frequency found that comes closest to the target frequency. The PLL structure returns the values required for `edt_set_frequency_ics307`.

## edt_set_out_clk_ics307

### Description

Sets the frequency output on the PCI CDa board, using parameters computed by `edt_find_vco_frequency_ics307`. The `clock_channel` command selects the clock channel to which this should be applied. The CDa has only one clock channel; therefore, the channel ID should always be set to 0.

### Syntax

```
#include "edtinc.h"

#include "edt_ss_vco.h"

void edt_set_out_clk_ics307(EdtDev *edt_p, edt_pll *clk_data,
int clock_channel);
```

### Arguments

| | |
|---|---|
| *edt_p* | Device handle returned from `edt_open`. |
| *clk_data* | The `edt_pll` structure filled in by `edt_find_vco_frequency_ics307` or `edt_find_vco_frequency_ics307_nodivide`. |
| *clock_channel* | Channel ID (0) |

## edt_set_frequency_ics307

### Description

This is a convenience function that first calls `edt_find_vco_frequency_ics307` to compute the parameters for the ICS307 PLL chip, then calls `edt_set_out_clk_ics307` to set the frequency on the desired channel. If the target frequency is greater than 100 MHz, `edt_find_vco_frequency_ics307_nodivide` is used instead of `edt_find_vco_frequency_ics307`. The CDa has only one clock channel; therefore, the channel ID should always be set to 0.

### Syntax

```
#include "edtinc.h"

#include "edt_ss_vco.h"

double edt_set_frequency_ics307(EdtDev *edt_p, double ref_xtal,
double target, int clock_channel)
```

### Arguments

| | |
|---|---|
| *edt_p* | Device handle returned from `edt_open`. |
| *target* | The desired output frequency in Hz. |
| *xtal* | The base frequency of the PCI CDa board. Default is 10.3681 MHz |
| *clock_channel* | Channel ID (0) |

# Hardware Interface Protocol

This section describes how to connect your device to an PCI CD/CDa interface, including the electrical characteristics of the signal, the signal descriptions, the timing specifications, and the connector pinout.

## Electrical Interface

The PCI CD/CDa uses differential data transmission to transmit data at very high rates over long distances through noisy environments. Differential transmission nullifies the effects of ground shifts and noise. These effects appear as common mode voltages on the transmission line and are rejected by the receiver. A typical balanced differential circuit is shown below.

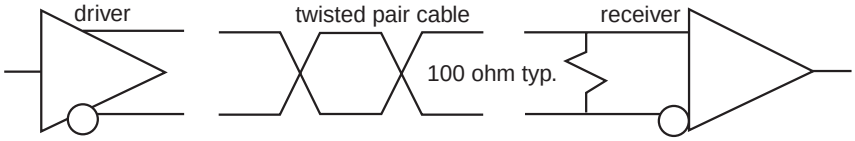



**Figure 2. Balanced Differential Circuit**

The interface is implemented with 32 signals, each implemented as a differential pair of wires.

### RS422

The PCI CD-20 DMA interface protocol uses RS422 signal levels. RS422 is defined by the Electronic Industries Association to provide robust high-speed data transmission. It allows signaling rates up to 10 MHz for short cables of up to 12 meters (40 feet) and cables of up to 1219 meters (4000 feet) at 100 KHz, as shown in Figure 3.

For further information, see the EIA RS422-A standard, *Electrical Characteristics of Balanced Voltage Digital Interface Circuits*, Dec. 1978, available from the Electronic Industries Association, Engineering Department, 2001 Eye St. N.W., Washington, D.C. 20006.

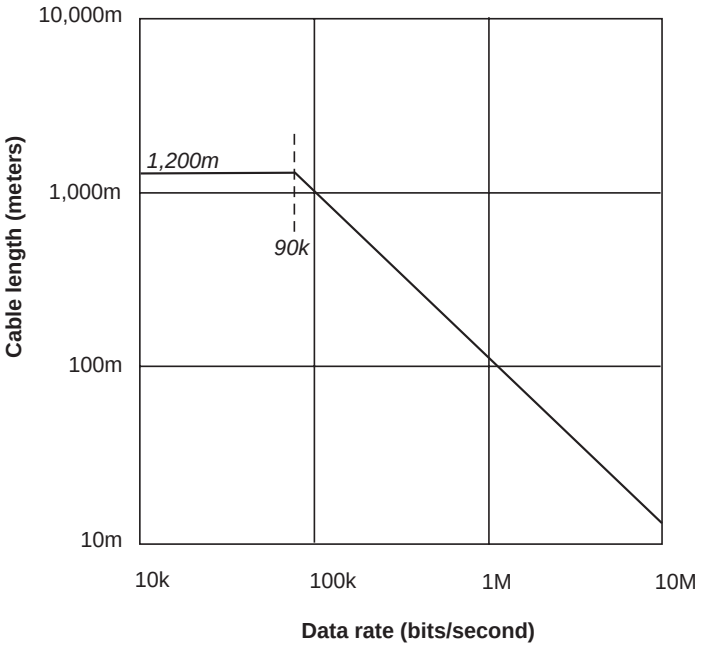
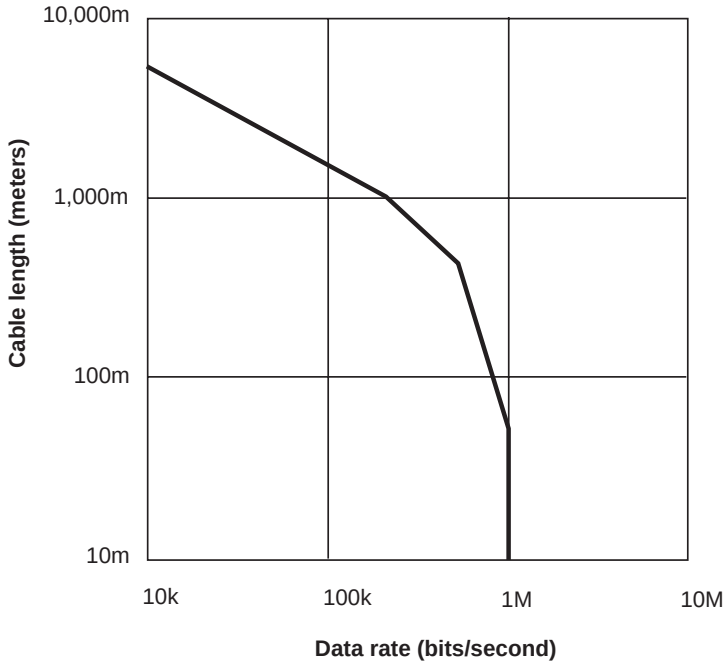



**Figure 3. RS422 Data Signalling Rate and Cable Length**

### Pseudo-ECL

The PCI CD-40 DMA interface protocol uses AT&T pseudo-emitter-coupled logic (PECL) signal levels. Pseudo-ECL levels are ECL levels that are shifted by 5 volts to run on a single +5 volts power supply. These devices are excellent for minimizing electromagnetic interference where data must be transmitted at very high speeds.




**PECL Data Signaling Rate and Cable Length**

For further information, see *The 41 Series of High Performance Line Drivers, Receivers, and Transceivers*, Jan. 1991 Data Book and Designers Guide, available from AT&T Microelectronics, CA91-001DBIP.

### LVDS

The PCI CD-60 DMA interface protocol uses low-voltage differential signaling, a new standard for faster signaling at lower power, compatible with IEEE 1596.3 SCI LVDS standard, and conforming to ANSI/TIA/EIA-644 LVDS standard.

# Signals

The hardware flow control protocol assumes that FIFO or memory buffers on both ends implement almost-full and almost-empty thresholds. Therefore, when a “not ready to accept data” signal is sent to the transmitting device, the receiver can still accommodate enough data to allow for cable delay and synchronization.

| Signal | PCI CD I/O | Description |
|---|---|---|
| DAT(15:0) | I/O | Sixteen bi-directional data lines for DMA data. |
| STAT(3:0) | I | Four general-purpose control inputs. Any can be enabled to interrupt the PCI bus host. |
| FUNCT(3:0) | O | Four general-purpose program control outputs. Can be used to reset the user device or indicate DMA direction for bi-directional devices. |
| SENDT | O | Send Timing is a constant clock driven by the DMA interface that can be used by the user device to generate the receive timing. This signal does not hae to be used. The PCI CD-20 outputs a 10 MHz clock. The PCI CD-40 outputs a 20 MHz clock. |
| RXT | I | Receive Timing is an input to the DMA device. When the pcd_looped.bit configuration is used, the RXT signal frequency must be generated by the external device. When the pcd_src.bit is used, the RXT signal must be looped back from the SENDT signal. It is best, although not required, that this signal is a continuous clock. Data clocked into the DMA interface must propagate through pipelining registers before it can be transferred into PCI bus memory. If the RXT clock stops, data is left in this pipe instead of being transferred to host memory. |
| TXT | O | Transmit Timing is an output from the DMA interface. TXT synchronizes the DMA output data and control signals. TXT is either internally generated from the same source as SENDT, or looped back from RXT. |
| IDV | I | Input Data Valid is asserted by the device synchronous with RXT, to tell the DMA interface that data on the DATA(15:0) signals are valid and must be registered and transferred to the PCI bus memory. The DMA interface will accomplish this unless the BNR signal has been asserted for >32 IDV signals. |
| BNR | O | Bus Not Ready is asserted by the DMA interface synchronous with TXT when 32 bytes or fewer of data space remains for input data from the device. This warns you to stop the data transfer or prepare for overflow. |
| ODV | O | Output Data Valid is asserted by the DMA interface when it has placed valid output data on DATA(15:0). ODV is asserted synchronously with the TXT clock, and only if the DNR signal is not asserted. |
| DNR | I | Device Not Ready is asserted by the device synchronous with the RXT clock when the user device is about to run out of space for storing data from the DMA interface. The amount of overrun buffer required in the device varies according to the cable length. The PCI CD may produce four or more words of valid |

| Signal | PCI CD I/O | Description |
|---|---|---|
| | | data after DNR is presented to the input pins. |

**Table 4. Signals**

# Timing

The clock and data output timing is specified at the pins of the PCI CD/CDa connector.

| | PCI CD-20 | PCI CD-40 | PCI CD-60 |
|---|---|---|---|
| Clock frequency | 0-10 MHz | 0-20 MHz | 0-30 MHz |
| Clock duty cycle | 50% ± 10 ns | 50% ± 5 ns | 50% ± 5 ns |
| Input minimum setup time | 20 ns | 5 ns | 5 ns |
| Input minimum hold time | 25 ns | 6 ns | 6 ns |
| Output maximum propagation delay | 20 ns | 10 ns | 10 ns |

**Table 5. Timing Specifications**

The following figure shows the PCI CD timing:

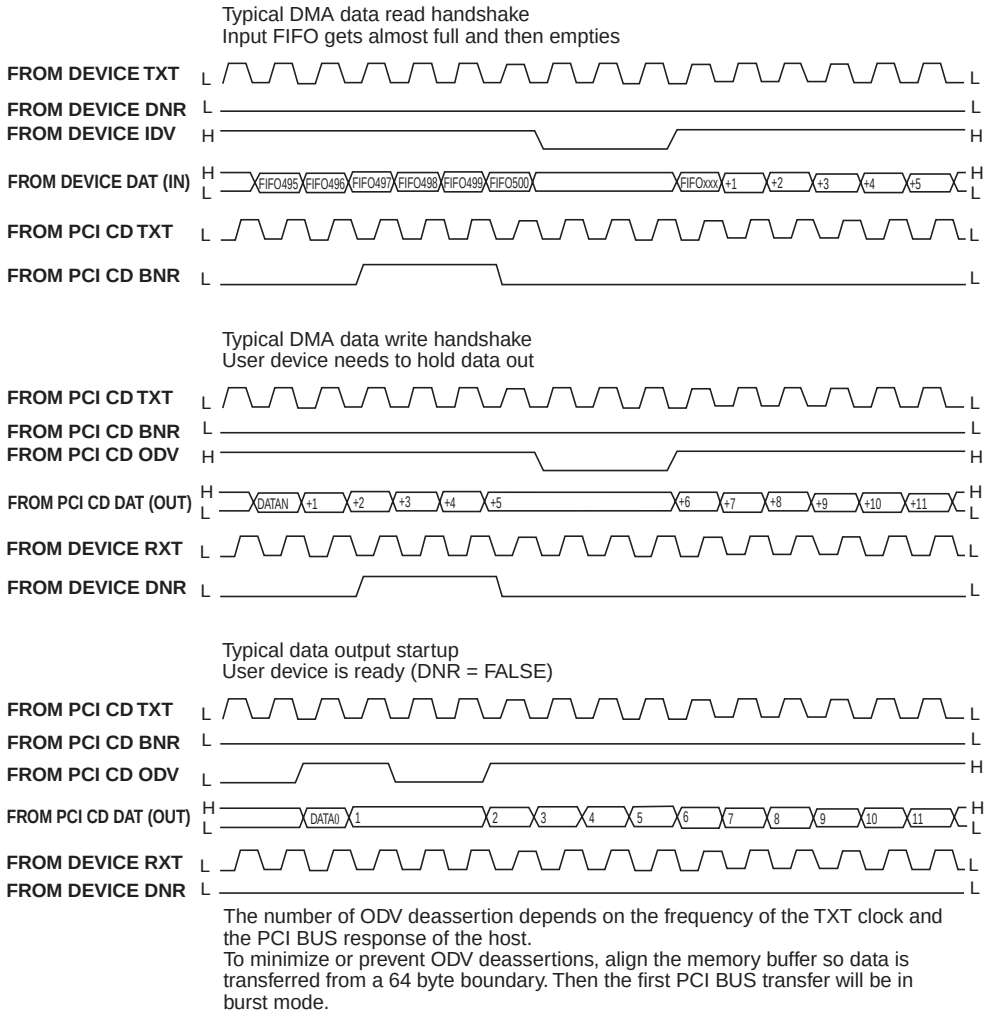



**Figure 6. Timing Specifications**

# Connector Pinout

The PCI CD/CDa board uses a high-density 80-pin I/O connector, AMP part number 787190-8. The cable uses a straight-shielded backshell (AMP P/N 749196-2) or right angle backshell (AMP P/N 749621-8).

The following pinout describes the connection from the PCI CD/CDa board to the cable.

**Note:** Do not connect your own circuits to the unused pins, as they may be internally connected to the PCI CD/CDa.

**Note:** If you're using the PMC version of the PCI CD, consult the PMC Supplement for the PCI CD and PCI DVK, EDT part number 008-01356, for the correct pinout.

| AMP | Signal | AMP | Signal |
|---|---|---|---|
| 1 | Ground | 41 | Ground |
| 2 | Ground | 42 | Ground |
| 3 | DATI4+ | 43 | DATI0+ |
| 4 | DATI4– | 44 | DATI0– |
| 5 | DATI5+ | 45 | DATI1+ |
| 6 | DATI5– | 46 | DATI1– |
| 7 | DATI6+ | 47 | DATI2+ |
| 8 | DATI6– | 48 | DATI2– |
| 9 | DATI7+ | 49 | DATI3+ |
| 10 | DATI7– | 50 | DATI3– |
| 11 | DATO4+ | 51 | DATO0+ |
| 12 | DATO4– | 52 | DATO0– |
| 13 | DATO5+ | 53 | DATO1+ |
| 14 | DATO5– | 54 | DATO1– |
| 15 | DATO6+ | 55 | DATO2+ |
| 16 | DATO6– | 56 | DATO2– |
| 17 | DATO7+ | 57 | DATO3+ |
| 18 | DATO7– | 58 | DATO3– |
| 19 | SPARE 0+ | 59 | SPARE0– |
| 20 | +5V | 60 | +5V |
| 21 | SPARE 1+ | 61 | SPARE1– |
| 22 | SPARE 2+ | 62 | SPARE2– |
| 23 | Ground | 63 | Ground |
| 24 | STAT0+ | 64 | RXT+ |
| 25 | STAT0– | 65 | RXT– |
| 26 | STAT1+ | 66 | IDV+ |
| 27 | STAT1– | 67 | IDV– |
| 28 | STAT2+ | 68 | DNR+ |
| 29 | STAT2– | 69 | DNR– |
| 30 | STAT3+ | 70 | Reserved+ |
| 31 | STAT3– | 71 | Reserved– |
| 32 | FUNCT0+ | 72 | SENDT+ |
| 33 | FUNCT0– | 73 | SENDT– |
| 34 | FUNCT1+ | 74 | ODV+ |
| 35 | FUNCT1– | 75 | ODV– |
| 36 | FUNCT2+ | 76 | BNR+ |
| 37 | FUNCT2– | 77 | BNR– |
| 38 | FUNCT3+ | 78 | TXT+ |
| 39 | FUNCT3– | 79 | TXT– |
| 40 | Ground | 80 | Ground |

**Table 7. PCI CD (pcd8_src.bit)**

| AMP | Signal | AMP | Signal |
|---|---|---|---|
| 1 | Ground | 41 | Ground |
| 2* | Ground | 42* | Ground |
| 3 | DAT4+ | 43 | DAT0+ |
| 4 | DAT4– | 44 | DAT0– |
| 5 | DAT5+ | 45 | DAT1+ |
| 6 | DAT5– | 46 | DAT1– |
| 7 | DAT6+ | 47 | DAT2+ |
| 8 | DAT6– | 48 | DAT2– |
| 9 | DAT7+ | 49 | DAT3+ |
| 10 | DAT7– | 50 | DAT3– |
| 11 | DAT12+ | 51 | DAT8+ |
| 12 | DAT12– | 52 | DAT8– |
| 13 | DAT13+ | 53 | DAT9+ |
| 14 | DAT13– | 54 | DAT9– |
| 15 | DAT14+ | 55 | DAT10+ |
| 16 | DAT14– | 56 | DAT10– |
| 17 | DAT15+ | 57 | DAT11+ |
| 18 | DAT15– | 58 | DAT11– |
| 19 | SPARE 0+ | 59 | SPARE0– |
| 20 | +5V | 60 | +5V |
| 21 | SPARE 1+ | 61 | SPARE1– |
| 22 | SPARE 2+ | 62 | SPARE2– |
| 23 | Ground | 63 | Ground |
| 24 | STAT0+ | 64 | RXT+ |
| 25 | STAT0– | 65 | RXT– |
| 26 | STAT1+ | 66 | IDV+ |
| 27 | STAT1– | 67 | IDV– |
| 28 | STAT2+ | 68 | DNR+ |
| 29 | STAT2– | 69 | DNR– |
| 30 | STAT3+ | 70 | Reserved+ |
| 31 | STAT3– | 71 | Reserved– |
| 32 | FUNCT0+ | 72 | SENDT+ |
| 33 | FUNCT0– | 73 | SENDT– |
| 34 | FUNCT1+ | 74 | ODV+ |
| 35 | FUNCT1– | 75 | ODV– |
| 36 | FUNCT2+ | 76 | BNR+ |
| 37 | FUNCT2– | 77 | BNR– |
| 38 | FUNCT3+ | 78 | TXT+ |
| 39 | FUNCT3– | 79 | TXT– |
| 40 | Ground | 80 | Ground |

**Table 8. PCI CD and PCI CDa (pcd_src.bit, pcd_looped.bit, pcda.bit)**

*On PCI CDa, these pins are Spare.

| P3 | Channel–Pin | Signal | P3 | Channel–Pin | Signal |
|---|---|---|---|---|---|
| 1 | CH1–25 | Ground | 41 | CH2–25 | Ground |
| 2 | CH1–25 | Ground | 42 | CH2–25 | Ground |
| 3 | CH2–3 | CH2D0+ | 43 | CH1–3 | CH1D0+ |
| 4 | CH2–4 | CH2D0– | 44 | CH1–4 | CH1D0– |
| 5 | CH2–5 | CH2D1+ | 45 | CH1–5 | CH1D1+ |
| 6 | CH2–6 | CH2D1– | 46 | CH1–6 | CH1D1– |
| 7 | CH2–7 | CH2D2+ | 47 | CH1–7 | CH1D2+ |
| 8 | CH2–8 | CH2D2– | 48 | CH1–8 | CH1D2– |
| 9 | CH2–9 | CH2D3+ | 49 | CH1–9 | CH1D3+ |
| 10 | CH2–10 | CH2D3– | 50 | CH1–10 | CH1D3– |
| 11 | CH4–3 | CH4D0+ | 51 | CH3–3 | CH3D0+ |
| 12 | CH4–4 | CH4D0– | 52 | CH3–4 | CH3D0– |
| 13 | CH4–5 | CH4D1+ | 53 | CH3–5 | CH3D1+ |
| 14 | CH4–6 | CH4D1– | 54 | CH3–6 | CH3D1– |
| 15 | CH4–7 | CH4D2+ | 55 | CH3–7 | CH3D2+ |
| 16 | CH4–8 | CH4D2– | 56 | CH3–8 | CH3D2– |
| 17 | CH4–9 | CH4D3+ | 57 | CH3–9 | CH3D3+ |
| 18 | CH4–10 | CH4D3– | 58 | CH3–10 | CH3D3– |
| 19 | | Not used | 59 | | Not used |
| 20 | | 5V DC (fused) | 60 | | 5V DC (fused) |
| 21 | | Not used | 61 | | Not used |
| 22 | | Not used | 62 | | Not used |
| 23 | CH3–25 | Ground | 63 | CH4–25 | Ground |
| 24 | | Reserved | 64 | CH1–1 | CH1CLK+ |
| 25 | | Reserved | 65 | CH1–2 | CH1CLK– |
| 26 | | Reserved | 66 | | Reserved |
| 27 | | Reserved | 67 | | Reserved |
| 28 | | Reserved | 68 | | Reserved |
| 29 | | Reserved | 69 | | Reserved |
| 30 | | Reserved | 70 | CH3–1 | CH3CLK+ |
| 31 | | Reserved | 71 | CH3–2 | CH3CLK– |
| 32 | CH2–1 | CH2CLK+ | 72 | | Reserved |
| 33 | CH2–2 | CH2CLK– | 73 | | Reserved |
| 34 | | Reserved | 74 | | Reserved |
| 35 | | Reserved | 75 | | Reserved |
| 36 | | Reserved | 76 | | Reserved |
| 37 | | Reserved | 77 | | Reserved |
| 38 | CH4–1 | CH4CLK+ | 78 | | Reserved |
| 39 | CH4–2 | CH4CLK– | 79 | | Reserved |
| 40 | CH3–25 | Ground | 80 | CH4–25 | Ground |

**Table 9. PCI CD (ssd.bit, ssd2.bit, ssd4.bit)**

| P3 | Channel–Pin | Signal | P3 | Channel–Pin | Signal |
|---|---|---|---|---|---|
| 1 | CH1–25 | Ground | 41 | CH2–25 | Ground |
| 2 | CH1–25 | Ground | 42 | CH2–25 | Ground |
| 3 | CH2–3 | CH2D0+ | 43 | CH1–3 | CH1D0+ |
| 4 | CH2–4 | CH2D0– | 44 | CH1–4 | CH1D0– |
| 5 | CH2–5 | CH2D1+ | 45 | CH1–5 | CH1D1+ |
| 6 | CH2–6 | CH2D1– | 46 | CH1–6 | CH1D1– |
| 7 | CH2–7 | CH2D2+ | 47 | CH1–7 | CH1D2+ |
| 8 | CH2–8 | CH2D2– | 48 | CH1–8 | CH1D2– |
| 9 | CH2–9 | CH2D3+ | 49 | CH1–9 | CH1D3+ |
| 10 | CH2–10 | CH2D3– | 50 | CH1–10 | CH1D3– |
| 11 | CH4–3 | CH4D0+ | 51 | CH3–3 | CH3D0+ |
| 12 | CH4–4 | CH4D0– | 52 | CH3–4 | CH3D0– |
| 13 | CH4–5 | CH4D1+ | 53 | CH3–5 | CH3D1+ |
| 14 | CH4–6 | CH4D1– | 54 | CH3–6 | CH3D1– |
| 15 | CH4–7 | CH4D2+ | 55 | CH3–7 | CH3D2+ |
| 16 | CH4–8 | CH4D2– | 56 | CH3–8 | CH3D2– |
| 17 | CH4–9 | CH4D3+ | 57 | CH3–9 | CH3D3+ |
| 18 | CH4–10 | CH4D3– | 58 | CH3–10 | CH3D3– |
| 19 | | Not used | 59 | | Not used |
| 20 | | 5V DC (fused) | 60 | | 5V DC (fused) |
| 21 | | Not used | 61 | | Not used |
| 22 | | Not used | 62 | | Not used |
| 23 | CH3–25 | Ground | 63 | CH4–25 | Ground |
| 24 | | Reserved | 64 | CH1–1 | CH1CLK+ |
| 25 | | Reserved | 65 | CH1–2 | CH1CLK– |
| 26 | | Reserved | 66 | | Reserved |
| 27 | | Reserved | 67 | | Reserved |
| 28 | | Reserved | 68 | | Reserved |
| 29 | | Reserved | 69 | | Reserved |
| 30 | | Reserved | 70 | CH3–1 | CH3CLK+ |
| 31 | | Reserved | 71 | CH3–2 | CH3CLK– |
| 32 | CH2–1 | CH2CLK+ | 72 | | Reserved |
| 33 | CH2–2 | CH2CLK– | 73 | | Reserved |
| 34 | | Reserved | 74 | | Reserved |
| 35 | | Reserved | 75 | | Reserved |
| 36 | | Reserved | 76 | | Reserved |
| 37 | | Reserved | 77 | | Reserved |
| 38 | CH4–1 | CH4CLK+ | 78 | | Reserved |
| 39 | CH4–2 | CH4CLK– | 79 | | Reserved |
| 40 | CH3–25 | Ground | 80 | CH4–25 | Ground |

**Table 10. PCI CD (ssdout1.bit, ssdio.bit, ssdio_neg.bit)**

| P3 | Channel–Pin | P3 | Channel–Pin |
|---|---|---|---|
| 1 | Ground | 41 | Ground |
| 2 | Ground | 42 | Ground |
| 3 | CH2D+ | 43 | CH0D+ |
| 4 | CH2D– | 44 | CH0D– |
| 5 | CH2CLK+ | 45 | CH0CLK+ |
| 6 | CH2CLK– | 46 | CH0CLK– |
| 7 | CH3D+ | 47 | CH1D+ |
| 8 | CH3D– | 48 | CH1D– |
| 9 | CH3CLK+ | 49 | CH1CLK+ |
| 10 | CH3CLK– | 50 | CH1CLK– |
| 11 | CH6D+ | 51 | CH4D+ |
| 12 | CH6D– | 52 | CH4D– |
| 13 | CH6CLK+ | 53 | CH4CLK+ |
| 14 | CH6CLK– | 54 | CH4CLK– |
| 15 | CH7D+ | 55 | CH5D+ |
| 16 | CH7D– | 56 | CH5D– |
| 17 | CH7CLK+ | 57 | CH5CLK+ |
| 18 | CH7CLK– | 58 | CH5CLK– |
| 19 | EXTCLKIN+ | 59 | EXTCLKIN– |
| 20 | +5V | 60 | +5V |
| 21 | Spare | 61 | Spare |
| 22 | Spare | 62 | Spare |
| 23 | Spare | 63 | Spare |
| 24 | CH8D+ | 64 | CH10D+ |
| 25 | CH8D– | 65 | CH10D– |
| 26 | CH8CLK+ | 66 | CH10CLK+ |
| 27 | CH8CLK– | 67 | CH10CLK– |
| 28 | CH9D+ | 68 | CH11D+ |
| 29 | CH9D– | 69 | CH11D– |
| 30 | CH9CLK+ | 70 | CH11CLK+ |
| 31 | CH9CLK– | 71 | CH11CLK– |
| 32 | CH12D+ | 72 | CH14D+ |
| 33 | CH12D– | 73 | CH14D– |
| 34 | CH12CLK+ | 74 | CH14CLK+ |
| 35 | CH12CLK– | 75 | CH14CLK– |
| 36 | CH13D+ | 76 | CH15D+ |
| 37 | CH13D– | 77 | CH15D– |
| 38 | CH13CLK+ | 78 | CH15CLK+ |
| 39 | CH13CLK– | 79 | CH15CLK– |
| 40 | GROUND | 80 | GROUND |

**Table 11. PCI CDa (ssd16io.bit)**

# Registers

The PCI CD/CDa 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 CD/CDa 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 CD/CDa-specific parameters, as defined in the file *pcd.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 CD/CDa is as shown in Figure 6 and described below.

| **Address Bits** | 31 16 | | 15 0 | |
|---|---|---|---|---|
| 0x00 | Device ID: PCD.SSD = 26<br>PCD64 = 26<br>PCD16 = 27 | | 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) | | | |
| 0x14 | Remote Xilinx Memory-Mapped IO 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 Tables 6 and 7. For complete descriptions of the bits in the status and command fields, see the *PCI Local Bus Specification*, Revision 2.1, 1995, 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 | 1 | 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 12. 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 | implemented | 10–15 | reserved | 0 |
| 5 | VGA Palette Snoop | 0 | | | |

**Table 13. Configuration Space Status Field Values**

# PCI Local Bus Addresses

Figure  describes the PCI CD/CDa 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 |
|---|---|---|
| 0xCC | remote Xilinx data | |
| 0xC8 | PCI interrupt status | |
| 0xC4 | PCI interrupt and remote Xilinx configuration | |
| 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 | |
| 0x00 | main DMA current address | |

| Byte Word | 3 2 | 1 0 |
|---|---|---|
| | 1 | 0 |

**Table 14. 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 Figure 1):

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 |

**Table 15. Scatter-gather DMA List Format**

# 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 Figure 6.
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 74. |

| 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 74. |

| 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).<br>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.<br>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).<br>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.<br>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). |

# Interrupt Registers

## PCI Interrupt and Remote Xilinx Configuration Register

| | |
|---|---|
| Size | 32-bit |
| I/O | read-write |
| Address | 0xC4 |
| Access | EDT_REMOTE_OFFSET |
| Comment | Remote Xilinx is also referred to as Interface or User Xilinx. |

| Bit | EDT_ | Description |
|---|---|---|
| D31–22 | | Not used. |
| D21 | RMT_STATE | Remote Xilinx INIT pin state. This bit is read-only. |
| D20 | RMT_DONE | Remote Xilinx DONE pin. |
| D19 | RMT_PROG | Remote Xilinx PROG pin. |
| D18 | RMT_INIT | Remote Xilinx INIT pin. |
| D17 | EN_CCLK | Enable one configuration clock cycle to remote Xilinx. |
| D16 | RMT_DATA | Remote Xilinx program data. |
| D15 | PCI_EN_INTR | Enable PCI interrupt. |
| D14 | RMT_EN_INTR | Enable Remote Xilinx interrupt. |
| D13–9 | | Not used. |
| D8 | RFIFO_ENB | After the remote Xilinx has been programmed to your satisfaction:<br>1. Clear, then set bit D3 of the remote Xilinx command register.<br>2. Set this bit to enable the burst data FIFO. |
| D7 | | Not used. |
| D6–0 | RMT_ADDR | 128-byte address of remote Xilinx register. |

To program the remote Xilinx:

1. Set the PROG and INIT pins low.
2. Wait for DONE (D20) to be low.
3. Set the PROG and INIT pins high.
4. Loop until INIT state (D21) goes high.
5. Wait four µs.
6. Write programming date, one bit at a time, to D16 with D17 high.
7. After all data is written, continue writing ones to D16 until DONE (D20) goes high.

The programming has failed if it has not completed after 32 clock cycles.

### PCI Interrupt Status Register

| | |
|---|---|
| Size | 32-bit |
| I/O | read-only |
| Address | 0xC8 |
| Access | EDT_DMA_STATUS |
| Comments | The driver uses this register initially to determine the source of a PCI interrupt. |

| Bit | EDT_ | Description |
|---|---|---|
| D16–31 | | Not used. |
| D15 | PCI_INTR | PCI interrupt. When asserted, the PCI CD is asserting an interrupt on the PCI bus. |
| D14 | | Not used. |
| D13 | RMT_INTR | Remote Xilinx interrupt. When asserted, the remote Xilinx interrupt is set. If bits 14 and 15 of the the PCI interrupt and remote Xilinx configuration register are aserted, the remote Xilinx causes a PCI interrupt. |
| D12 | DMA_INTR | End of DMA interrupt. Asserted when at least one of the DMA interrupts is asserted in the scatter-gather DMA next count and control register. Causes a PCI interrupt if bit 15 of the PCI interrupt and remote Xilinx configuration register is enabled. |
| D11–0 | | Not used. |

# Remote Xilinx Registers

The Xilinx chip is a programmable integrated circuit used to implement the PCI CD interface or to test the board. The Xilinx programmable IC is programmed serially, using the PCI interrupt and remote Xilinx configuration register as described on page 81.

***Note***

The following registers are defined to control the interface and reside in the remote Xilinx IC. In order to access those registers, the PCI CD requires that the remote Xilinx be loaded with a program that defines them. If the Xilinx is not loaded, or loaded with a different program, these registers are inaccessible.

The Xilinx IC is programmed when the PCI CD driver is loaded, or by the application program. If you have received a customized application program for your PCI CD from Engineering Design Team,some or all of these registers may not be defined. Consult the documentation that came with your customized program instead.

## Command Register

| | |
|---|---|
| Size | 8-bit |
| I/O | read-write |
| Address | 0x00 |
| Access | PCD_CMD |

| Bit | EDT_ | Description |
|---|---|---|
| D0 | DIR | A value of 1 indicates data is coming in to the PCI CD. |
| D1 | FORCEBNR | A value of 1 tells device that board is not ready. |
| D2 | DATA_INV | If this bit is set, the PCI CD inverts the data. |
| D3 | ENABLE | Set to 1 to enable the PCI CD interface. This bit is set after the direction is chosen and typically after the first DMA buffer is ready. To reset direction or flags this bit must be reset.<br>To flush the DMA FIFOs, clear then set this bit. |
| D4-7 | STAT_INT_EN | A value of 1 enables the corresponding STAT bit to cause an interrupt when it is asserted. |

## Data Path Status Register

| | |
|---|---|
| Size | 8-bit |
| I/O | read-only |
| Address | 0x01 |
| Access | PCD_DATA_PATH_STAT |

| Bit | EDT_ | Description |
|---|---|---|
| D0 | OF_NOT_EMP | If this bit is set, the output FIFO is not empty. |
| D1 | IF_NOT_EMP | If this bit is set, the input FIFO is not empty. |
| D2 | UNDERFLOW | If the DNR signal is low and the ODV signal goes low because the output FIFO runs out of data, this bit is asserted and remains so throughout the data transfer. Reset this bit with the ENABLE bit in the Command register. |
| D3 | OVERFLOW | This bit is asserted when the input FIFO is full and the IDV signal is high. Reset this bit with the ENABLE bit in the Command register. |
| D4-5 | INFFULL | If set, input FF is full. |
| D6 | INFFAFULL | If set, input FF is almost full. |
| D7 | IDV | Reflects IDV state. |

## Funct Register

| | |
|---|---|
| Size | 8-bit |
| I/O | read-write |
| Address | 0x02 |
| Access | PCD_FUNCT |

| Bit | PCD_ | Description |
|---|---|---|
| D0-3 | FUNCT | Sets the state of the user-definable FUNCT outputs. |
| D4-6 | FREQ6<br>FREQ7<br>SELAV | **PCI CD:** Used to program the PLL and select clock with the EDT library routine `edt_set_out_clock`.<br>**PCI CDa:** Not used. |
| D7 | PLLCLK | Set to enable PLL. |

## Stat Register

| | |
|---|---|
| Size | 8-bit |
| I/O | read-only |
| Address | 0x03 |
| Access | PCD_STAT |

| Bit | PCD_ | Description |
|---|---|---|
| D0-3 | STAT | The state of user-definable STAT input signals as last sampled by the RXT clock signal. |
| D4-7 | STAT_INT | Interrupt bits for the status bits. If the following conditions are both true, then the corresponding bit of these four can be asserted to cause a PCI Bus interrupt:<br>• The device interrupt is enabled using the RMT_EN_INTR bit in the PCI Interrupt and Remote Xilinx Configuration register.<br>• The corresponding bit is asserted in the command register (one of bits 4–7, named STAT_INT_EN).<br>The PCI Bus interrupt is then caused when the corresponding STAT signal is asserted according to the polarity specified in the stat polarity register. To reset the interrupt, disable and re-enable the appropriate STAT_INT_EN bit in the command register. |

## Stat Polarity Register

| | |
|---|---|
| Size | 8-bit |
| I/O | read-write |
| Address | 0x04 |
| Access | PCD_STAT_POLARITY |

| Bit | PCD_ | Description |
|---|---|---|
| D0-3 | POLARITY | A value of 0 indicates that a change from 0 to 1 from one clock cycle to the next causes an interrupt in the corresponding bit of the STAT_INT register, if the corresponding bit is also enabled in STAT_INT_EN.<br>A value of 1 causes the same event when the STAT_INT bit changes from 1 to 0 from one clock cycle to the next. |
| D4 | STAT_INT_ENA | Provides global enable or disable for all interrupt bits in Stat register, allowing the driver to disable and re-enable them in one operation, without altering the state of the Stat register. This bit is used mainly by the driver to disable the Stat interrupts to determine which other interrupts are pending. A value of 1 enables the interrupts. |
| D5 | ENA_OUT_CTRL | When set, enables the OUTPUT DISABLE signal on pin 22. Not used on CDa boards. |
| D6-7 | | Not used. |

## Direction Control Registers

| | |
|---|---|
| Size | 8-bit each |
| I/O | read-write |
| Address | A at 0x06, B at 0x07 |
| Access | PCD_DIRA and PCD_DIRB |
| Comments | These registers determine whether the physical drivers or receivers on the interface are inputs or outputs. Each pin on the PCI CD can be programmed as either an input or an output. Pins are normally configured for inputs or outputs as documented in the connector pinout shown on page 65. The PCI CD driver modifies the data bits appropriately for a *read* or *write* system call. Setting the same pins to serve as both input and output is useful for testing. Setting the same pins to serve as neither input nor output is not useful. Direction Control Register A controls the data direction; Direction Control Register B controls the direction of the control bit.<br>For example, to implement the interface documented in the connector pinout shown on page 65 for a write operation, set A to 0x0F and B to 0xCC. For a read, set A to 0xF0 and B to 0xCC. |

| Register | Bit | Description |
|---|---|---|
| B | DC15 | Pins 22 and 62 are outputs when this bit is low. |
| | DC14 | Pins 19, 59, 21, and 61 are outputs when this bit is low. |
| | DC13 | Pins 32–39 are outputs when this bit is low. |
| | DC12 | Pins 72–79 are outputs when this bit is low. |
| | DC11 | Pins 24–31 are outputs when this bit is low. |
| | DC10 | Pins 64–71 are outputs when this bit is low. |
| | DC9 | Pins 24–31 are inputs when this bit is low. |
| | DC8 | Pins 64–71 are inputs when this bit is low. |
| A | DC7 | Pins 11–18 are outputs when this bit is low. |
| | DC6 | Pins 51–58 are outputs when this bit is low. |
| | DC5 | Pins 3–10 are outputs when this bit is low. |
| | DC4 | Pins 43–50 are outputs when this bit is low. |
| | DC3 | Pins 11–18 are inputs when this bit is low. |
| | DC2 | Pins 51–58 are inputs when this bit is low. |
| | DC1 | Pins 3–10 are inputs when this bit is low. |
| | DC0 | Pins 43–50 are inputs when this bit is low. |

## Programmed I/O Low Register

Size 8-bit

I/O read-write

Address 0x08

Access PCD_PIO_OUTLO

| Bit | PCD_ | Description |
|---|---|---|
| 7–0 | | Outputs data on the low 8 bits of the 16-bit word. |

## Programmed I/O High Register

Size 8-bit

I/O read-write

Address 0x09

Access PCD_PIO_OUTHI

| Bit | PCD_ | Description |
|---|---|---|
| 7–0 | | Outputs data on the high 8 bits of the 16-bit word. |

## Interface Configuration Register

Size 8-bit

I/O read-write

Address 0x0F

Access PCD_CONFIG

| Bit | PCD_ | Description |
|---|---|---|
| 0 | BYTESWAP | A value of 1 swaps the order of bytes in a 16-bit word of data coming in from the data source. |
| 1 | SELRXT | A value of 1 selects RXT as source for TXT. |
| 2 | SETAV64 | Debug only. A value of 1 tells PCI Xilinx there is always data available. |
| 3 | SHORTSWAP | Set to 1 if the host computer writes the first 16-bit word on bits 16-31 of the PCI data bus (big-endian format) instead of bits 0-15 as defined in the PCI bus specification. |
| 4 | DED | Disable delay on start of outputting. If set, may cause |

| | | |
|---|---|---|
| | | ragged ODV on start and underflows. |
| 5 | SETDNR | A value of 1 stops transfer to device as if device set to DNR. |
| 6 | PIOEN | A value of 1 translates DMA channel buffers and enables programmed I/O registers at 8 and 9. A write to 9 generates a 1 clock inside ODV. |
| 7 | SETIDV | Set input data valid (used for debug). |

## PLL Programming Register (PCI CDa only)

| | |
|---|---|
| Size | 8-bit |
| I/O | read-write |
| Address | 0x20 |
| Access | EDT_SS_PLL_CTL |
| Comments | The program `set_ss_vco` uses this register to program the serial interface of the PLL. |

| Bit | Name | Description |
|---|---|---|
| 3-0 | PLL_STROBE | Connected to the strobe inputs of PLL 3 to 0, respectively. |
| 5-4 | | Not used. |
| 6 | PLL_DATA | Connected to PLL serial data input. |
| 7 | PLL_SCLK | Connected to PLL serial clock input. |

## PLL Divider Register (PCI CDa only)

| | |
|---|---|
| Size | 8-bit |
| I/O | read-only |
| Address | 0x24, 0x25 |
| Access | EDT_SS_PLL0_CLK, EDT_SS_PLL0_X |
| Comments | This register is set by `set_ss_vco`. It is a post-scalar divider used to achieve lower frequencies than those at which the PLLs can be programmed. After this division, the clocks are divided by two again to even the duty cycle. The `set_ss_vco` considers all these effects. Set D7 in the Funct register to enable. |

| Bit | Name | Description |
|---|---|---|
| 15-0 | PLLDIV | Programmable post divider |

## Output Data Valid Delay Register (PCI CDa only)

| | |
|---|---|
| Size | 8-bit |
| I/O | read-write |
| Address | 0x28 |
| Access | ODV_DELAY |

| Bit | Name | Description |
|---|---|---|
| 7-0 | ODV_DELAY | Set this register to the number of 16-bit words to hold off output. Outgoing data backs up in FIFO, reducing or eliminating ODV transitioning on start up. |

## LED Control Register

| | |
|---|---|
| Size | 8-bit |
| I/O | read-write |
| Address | 0x30 |
| Access | LED_CTL |

| Bit | Name | Description |
|---|---|---|
| 1-0 | LED0<br>LED1 | Set according to the following table to select what drives the LED: |
| 2 | LED2 | Set to 1 to turn LED on, only when LED0 and LED1 are set to 0 (see above chart). |

| LED1 | LED0 | Source |
|---|---|---|
| 0 | 0 | From LED signal. If you choose this source, you must also set bit 2 in this register. |
| 0 | 1 | IDV |
| 1 | 0 | DNR |
| 1 | 1 | RXT |

# Specifications

| | |
|---|---|
| **PCI Bus Compliance** | |
| Number of Slots | 1 |
| Transfer Size | PCI CD: Maximum 64 bytes per transfer<br>PCI CDa: Maximum 1024 bytes per transfer |
| DVMA master | Yes |
| PCI Bus memory space | Approximately 66 KB |
| Clock Rate | PCI CD: 33 MHz<br>PCI CDa: 66 MHz |
| **Device Data Transfer** | |
| Protocol | Synchronous stream |
| Buffers | Application specific |
| **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** | 5 V at 1.5 A |
| **Environmental** | |
| Temperature | Operating: 10 to 40° C<br>Nonoperating: -20 to 60° C |
| Humidity | Operating: 20 to 80% noncondensing at 40° C<br>Nonoperating: 95% noncondensing at 40°C |
| **Physical** | |
| Dimensions | 3.3” x 5.78” x 0.5” |
| Weight | 3.5 oz |

**Table 16. PCI Bus Configurable DMA Interface Specifications**