PCI GP

User's Guide

Revision: B
July 2005

Control Information

Control Item	Details
Document Owner	Hardware: Jerry Gaffke Software: Chet Britten
Information Label	**EDT Public**
Supersedes	None
File Location	c:/pcigp/pcigp.doc
Document Number	008-00965

Revision History

Revision	Date	Revision Description	Originator
A	04-March-02	Convert from FrameMaker	S Vasil
B	05-Jul-05	Update “Verifying the Installation” and “Upgrading the Firmware” sections.	M Mason

Contents

Overview

The PCI Bus General Purposes (PCI GP) interface is a single-slot, configurable DMA board for PCI bus-based computer systems. The PCI GP supports, but is not limited to, 4- and 16-channel serial operation. 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, satellite receivers, or research prototypes. The PCI GP uses a simple synchronous protocol for transferring data.

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

The PCI GP is extremely flexible, and daughter boards and other add-ons can be used to configure the PCI GP for a variety of uses. See the addenda for more information on specific configurations.

The input and output FIFO buffers are used to 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 GP with the UNIX-based and Windows NT/2000/XP operating systems.

Installation

Uninstall previous driver if there is one.

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 framegrabbers) be placed in one of the first two slots (closest to the AGP connector). The lower two PCI bus slots are only recommended for lower speed devices such as audio/modems/etc.

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

After installing the PCI GP, verify the installation, configure the device, and build the sample programs, if you wish. Instructions for uninstalling the software or upgrading the firmware on page 10.

Verifying the Installation

To verify that installation was successful and that the PCI GP is operating correctly:

1. Run `pciload`.
2. Verify that you are running `pcd64.bit`.
   Note: gp_xtest will only work with systems running `pcd64.bit`. To use `gp_xtest` with a different configuration file, run `pciload pcd64.bit`, cycle power, run `gp_xtest`, then re-run `pciload` with the correct configuration file for your board.
3. In Windows, run PCD Utilities. On Linux, bring up an xterm and CD to /opt/EDTpcd.
4. At the command prompt, enter:

```
gp_xtest 4096
```

GP_xtest returns test status information. You will be prompted to press **Return** at certain steps. The following is an example of proper behavior, although details will vary (see addenda for information specific to your configuration):

```
red# gp_xtest -u 1024
file <./gp_xtest.bit>
id: "gp_xtest.ncd 4036xlabg352 2002/04.01 10:18:13" loaded
reading 1024 words
buf at 26000
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
return to read:
Done.
0000 0100 0200 0300 0400 0500 0600 0700
Notice: need to swap bytes (big endian)
checking data
1024 words 0 errors loop 0
```

```
start speedtest? : y
reading 100 buffers of 1048576 bytes from unit 1 with 4 bufs
return to start:

19999610.794337 bytes/sec
time 5 .242982
Hit return to exit

red#
```

Configuring the PCI GP

The PCI GP can be configured to run in several modes. There are three different PCI interfaces that need a matching user interface: PCD64 (single-channel DMA, 64-word burst); PCDSSD (4-channel DMA); PCD16 (16-channel DMA). PCD64 uses gp_pcd.bit or gp_pcd8.bit, PCDSSD uses gpssd4.bit, and PCD16 uses gp_ssd16.bit. Before running the device, decide which of these modes is appropriate for your application and configure it according to the following directions:

PCD64

You configure the PCI GP by downloading a configuration file (.bit file) to the Xilinx field-programmable gate array.

To configure the PCI GP:

1. Run PCD Utilities.
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/2000/XP/2000/XP systems).

pcdrequest creates a script or batch file *pcdload*, which contains the commands needed to download the appropriate firmware file into the on-board gate array. *pcdload* runs immediately, as well as whenever the computer is rebooted (except on NT systems).

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

Note *gp_xtest downloads its own test bit file automatically; after running gp_xtest, run pcdload to reload the default bit file.*

PCDSSD or PCD16

The register set of PCDSSD and PCD16 are complex and numerous; therefore, the user must configure the board using `initpcd`.

The following commands will load the gpssd16 configuration file and configure unit 0 for 16 inputs and unit 1 for 16 outputs:

```
% cd ./opt/EDTpcd
% ./initpcd –u 0 –v –f rd16.cfg
% ./initpcd –u 1 –v –f wr16.cfg
```

Without the –v option, initpcd is silent.

Complete documentation for initpcd can be found as as a man(1) formatted comment in /opt/EDTpcd/initpcd.c (UNIX) or \edt\pcd (Windows).

Several standard configurations for the CHEN, CHDIR, CHEDGE, and PCD Direction registers are provided as initpcd configuration files:

rd16.cfg	Channels 0-15 configured as inputs
wr16.cfg	Channels 0-15 configured as outputs
rdwr16.cfg	Channels 0-7 configured as inputs, 8-15 as outputs

In the following standard configuration file listings, the command_reg has the ENABLE bit set to enable the PCD, and the funct_Preg has the PLL_CLK_SELECT bit set to use the internal PLL clock instead of the PCD clock (the PLL clock is used only for driving output clocks). Note that rd16.cfg uses the falling clock edge, while wr16.cfg uses the rising clock edge.

Standard configuration file listings:

rd16.cfg

bitfile	gpssd16.bit
command_reg	0x08
funct_reg	0x80
ssd16_chen_reg	0xFFFF
ssd16_chdir_reg	0x0000
ssd16_chedge_reg	0x0000
direction_reg	0xFCF0
flush_fifo	1

wr16.cfg

bitfile	gpssd16.bit
command_reg	0x08
funct_reg	0x80
ssd16_chen_reg	0xFFFF
ssd16_chdir_reg	0xFFFF
ssd16_chedge_reg	0xFFFF
direction_reg	0xC30F
flush_fifo 1	

rdwr16.cfg

bitfile	gpssd16.bit

command_reg	0x08
funct_reg	0x80
ssd16_chen_reg	0xFFFF
ssd16_chdir_reg	0xFF00
ssd16_chedge_reg	0xFF00

Example

The following configuration file will program the GPSSD16 with channels 0–7 as inputs and channels 8–15 as outputs.

rdwr16.cfg

bitfile	gpssd16.bit
command_reg	0x08
funct_reg	0x80
ssd16_chen_reg	0xFFFF
ssd16_chdir_reg	0xFF00
ssd16_chedge_reg	0xFF00
direction_reg	0xC3F0
flush_fifo	1

In this example:

- The lowest nibble enables inputs for channels 0–7
- The second lowest nibble disables outputs for channels 0–7
- The second highest nibble disables inputs and enables outputs for channels 8–11
- The two lower bits of the highest nibble enables outputs for channels 12–15. The two highest bits are always high.

Note: “flush_fifo: 1” should always be included as the FIFO flush enables the Xilinx state machine. The gpssd16 won’t run without this.

Building the Sample Programs

UNIX-based Systems

- To build any of the example programs on UNIX-based systems, bring up cd and navigate to /opt/EDTpcd, then enter the command:

  ```
  make file
  ```

  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
```

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

Windows NT/2000/XP Systems

To build any of the example programs on Windows NT/2000/XP systems, you must have VC++ installed, then:

1. Run PCD Utilities.
2. Enter the command:

```
nmake file
```

   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
```

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

- You can also build the sample programs by setting up a project in Windows Visual C++.

Uninstalling

Solaris-based Systems

To remove the PCI CD driver on Solaris-based 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 driver on Linux systems:

1. Become root or superuser.
2. Enter:

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

Windows NT/2000/XP Systems

To remove the PCI CD toolkit on Windows NT/2000/XP systems, use the Windows NT/2000/XP Add/Remove utility. For further details, consult your Windows NT/2000/XP 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

After upgrading to a new device driver, it may sometimes also be necessary to upgrade the PCI interface Xilinx PROM. If so, the *readme* file will say so. This is not necessary with first-time installations.

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

1. If necessary, navigate to the directory in which you installed the driver (for UNIX-based systems, usually */opt/EDTpcd*; for Windows NT/2000/XP, usually *C:\EDT\pcd*).
2. At the prompt, enter (depending on which EDT board you have):

```
pciload -u 0 pcd64.bit        /*For single-channel*/
or
pciload –u 0 pcdssd.bit       /* For 4-channel */
or
pciload -u 0 pcd16.bit        /* For 16-channel */
```

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.

pciload can also be used to detect which boards are in the system and verify their firmware levels. To check the board's Xilinx PROM against the Xilinx file in the current package, run

```
pciload –u 0 –v pcd64.bit      /* For single-channel*/
or
pciload –u 0 –v pcdssd.bit     /* For 4-channel */
or
pciload –u 0 -v pcd16.bit      / *For 16-channel */
```

To obtain a list of all the EDT boards currently installed in the system with their unit numbers and PCI firmware dates and revisions, simply enter the *pciload* command without arguments:

```
pciload
```

Input and Output

The EDT Product device driver can perform two kinds of DMA transfers: continuous and noncontinuous. For noncontinuous transfers, the driver uses DMA system calls read() and write(). *Each* read() *and* write() *system* call *allocates* kernel resources, during which time DMA transfers are interrupted.

To perform continuous transfers, use the 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 you can configure. See the sample programs simple_getdata.c and simple_putdata.c distributed with the driver for examples of using the ring buffers.

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 using 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("edt", 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) ;

      edt_start_buffers(edt_p, 0) ; /* 0 starts unlimited buffer
DMA*/

      /* 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_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("edt", 1, 2) ;
      char buf[1024] ;
      int numbytes, outfd = open("outfile", 1) ;
      /*
       * Becase 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 (typically 1 MB) buffers configured in a round-robin data FIFO. During capture, the application must be able to transfer or process the data before data acquisition wraps around and overwrites the buffer currently being processed.

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

```
#include "edtinc.h"

main()
{
      EdtDev *edt_p = edt_open("edt", 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, 0x100000, 4, EDT_READ,
NULL) ;

        edt_start_buffers(edt_p, 0) ; /* 0 starts unlimited buffer
DMA */

        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) ;
        }
}
```

If running under Solaris 2.x, use the "-D_REENTRANT -ledt -lthread" options to compile and link the library file libedt.a with your program. See the makefile and example programs provided for examples of compiling programs using the library routines.

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.

If driver-specific library routines exist, they can be found in the following section.

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.

Routine	Description
edt_wait_for_buffers	Blocks until the specified number of buffers have completed.
edt_wait_for_next_buffer	Waits for the next buffer that completes DMA.
edt_wait_buffers_timed	Blocks until the specified number of buffers have completed; returns a pointer to the time that the last buffer finished.
edt_next_writebuf	Returns a pointer to the next buffer scheduled for output DMA.
edt_set_buffer	Sets which buffer should be started next.
edt_set_buffer_size	Used to change the size or direction of one of the ring buffers.
edt_last_buffer	Waits for the last buffer that has been transferred.
edt_last_buffer_timed	Like edt_last_buffer but also returns the time at which the dma was complete on this buffer.
edt_configure_ring_buffers	Configures the ring buffers.
edt_buffer_addresses	Returns an array of addresses referencing the ring buffers.
edt_disable_ring_buffers	Stops DMA transfer, disables ring buffers and releases resources.
edt_ring_buffer_overrun	Detects ring buffer overrun which may have corrupted data.
edt_reset_ring_buffers	Stops DMA in progress and resets the ring buffers.
edt_configure_block_buffers	Configures ring buffers using a contiguous block of memory.
edt_startdma_action	Specifies when to perform the action at the start of a dma transfer as set by edt_startdma_reg().
edt_enddma_action	Specifies when to perform the action at the end of a dma transfer as set by edt_ednddma_reg().
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

Routine	Description
	as the buffer_timed function.
edt_ref_tmstamp	Used for debugging. Able to see a history with `setdebug -g` with an application defined event in the same timeline as driver events.
edt_get_burst_enable	Returns a value indicating whether PCI Bus burst transfers are enabled during DMA.
edt_set_burst_enable	Turns on or off PCI Bus burst transfers during DMA.
edt_get_firstflush	Returns the value set by edt_set_firstflush(). This is an obsolete function.
edt_set_firstflush	Tells whether and when to flush FIFOs before DMA.
edt_flush_fifo	Flushes the EDT Product FIFOs.
edt_get_goodbits	Returns the number of bits from the remote device.
Control	
edt_set_event_func	Defines a function to call when an event occurs.
edt_remove_event_func	Removes a previously set event function.
edt_reg_read	Reads the contents of the specified EDT Product register.
edt_reg_write	Writes a value to the specified EDT Product register.
edt_reg_and	ANDs the value provided with the value of the specified EDT Product register.
edt_reg_or	ORs the value provided with the value of the specified EDT Product 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	Sleeps for the specified number of microseconds.
edt_alloc	Allocate page-aligned memory in a system-independent way.
edt_free	ree 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.

For an example of real-time data capture using ring buffers, see the example on page XX.

See Also

edt_set_rtimeout, edt_set_wtimeout

edt_wait_for_next_buffer

Description

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

Syntax

#include "edtinc.h"

void *edt_wait_for_next_buffer(EdtDev *edt_p) ;

Arguments

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

Return

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

edt_wait_buffers_timed

Description

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

Syntax

```
#include "edtinc.h"

void *edt_wait_buffers_timed (EdtDev *edt_p, int count, uint
*timep);
```

Arguments

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

count buffer number for which to block. Completed buffers are numbered cumulatively starting with 0 when the EDT Product is opened.

timep pointer to an array of two unsigned integers. The first integer is seconds, the next integer is microseconds representing the system time at which the buffer completed.

Return

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

Note

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

edt_next_writebuf

Description

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

Syntax

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

Arguments

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

Return

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

edt_set_buffer

Description

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

Arguments

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

Syntax

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

Example

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

Return

0 on success, -1 on failure.

See Also

edt_stop_buffers(), edt_done_count(), edt_get_todo()

edt_set_buffer_size

Description

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

more cleanly using seperate, simultaneous, read and write dma transfers using different dma channels.

Arguments

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

Syntax

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

Example

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

Return

0 on success, -1 on failure.

See Also

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

edt_last_buffer

Description

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

Arguments

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

Syntax

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

Example

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

Return

Address of the image.

See Also

edt_wait_for_buffers, edt_last_buffer_timed

edt_last_buffer_timed

Description

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

Arguments

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

Syntax

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

Example

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

Return

Address of the image.

See Also

edt_wait_for_buffers(), edt_last_buffer(), edt_wait_buffers_timed

edt_configure_ring_buffers

Description

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

Syntax

```
#include "edtinc.h"

int edt_configure_ring_buffers(EdtDev *edt_p, int bufsize, int
nbufs,
                    int data_output, void *bufarray[]);
```

Arguments

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

Return

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

edt_buffer_addresses

Description

Returns an array containing the addresses of the ring buffers.

Syntax

```
#include "edtinc.h"

void **edt_buffer_addresses(EdtDev *edt_p);
```

Arguments

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

Return

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

edt_disable_ring_buffers

Description

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

Syntax

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

Arguments

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

Return

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

edt_ring_buffer_overrun

Description

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

Syntax

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

Arguments

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

Return

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

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

edt_reset_ring_buffers

Description

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

Syntax

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

Arguments

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

Return

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

edt_configure_block_buffers

Description

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

Syntax

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

Arguments

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

Return

0 on success, -1 on failure.

See Also

edt_configure_ring_buffers

edt_startdma_action

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_startdma_reg(), edt_reg_write(), edt_reg_read()

edt_enddma_action

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

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

edt_startdma_reg

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_startdma_action()

edt_abort_dma

Description

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

Syntax

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

Arguments

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

Return

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

edt_abort_current_dma

Description

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

Syntax

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

Arguments

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

Return

0 on success, -1 on failure

edt_get_bytecount

Description

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

Syntax

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

Arguments

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

Return

The number of bytes transferred, as described above.

edt_timeouts

Description

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

Syntax

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

Arguments

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

Return

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

edt_get_timeout_count

Description

Returns the number of bytes transferred at last timeout.

Syntax

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

Arguments

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

Return

The number of bytes transferred at last timeout.

edt_set_timeout_action

Description

Sets the driver behavior on a timeout.

Syntax

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

Arguments

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

Return

No return value.

edt_get_timeout_goodbits

Description

Returns the number of good bits in the last long word of a read buffer after the last timeout. This routine is called after a timeout, if the timeout action is set to EDT_TIMEOUT_BIT_STROBE. (See edt_set_timeout_action on page **Error! Bookmark not defined..)**

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 Code Fontparatextefault ¶ Fonton page **Error! Bookmark not defined..**

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 Code Fontparatextefault ¶ Fonton page **Error! Bookmark not defined..**

Return

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

edt_reg_read

Description

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

Syntax

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

Arguments

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

Return

The value of the register.

edt_reg_write

Note

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

Description

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

Syntax

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

Arguments

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

Return

No return value.

edt_reg_and

Note

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

Description

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

Syntax

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

Arguments

edt_p device handle returned from edt_open or edt_open_channel

address The name of the register to modify. Use the names provided in the register descriptions in the section entitled "Hardware."

mask The value to AND with the register.

Return

The new value of the register.

edt_reg_or

Note

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

Description

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

Syntax

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

Arguments

edt_p device handle returned from edt_open or *edt_open_channel*

address The name of the register to modify. Use the names provided in the register descriptions in the section entitled "Hardware."

mask The value to OR with the register.

Return

The new value of the register.

edt_get_foicount

Description

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

Syntax

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

Arguments

edt_p device struct returned from edt_open

Example

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

Return

Integer

See Also

edt_set_foiunit(), edt_get_foiunit(), edt_set_foicount()

edt_set_foicount

Description

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

Syntax

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

Arguments

edt_p device struct returned from edt_open

unit unit number of RCI unit

Example

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

Return

0 on success, -1 on failure

See Also

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

edt_intfc_write

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_intfc_read(), edt_reg_write(), edt_intfc_write_short()

edt_intfc_read

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_intfc_write(), edt_reg_read(), edt_intfc_read_short()

edt_intfc_write_short

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_intfc_write(), edt_reg_write()

edt_intfc_read_short

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_intfc_read(), edt_reg_read()

edt_intfc_write_32

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_intfc_read_32(), edt_reg_write()

edt_intfc_read_32

Description

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

Syntax

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

Arguments

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

Example

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

Return

void

See Also

edt_intfc_write_32(), edt_reg_read()

edt_msleep

Description

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

Syntax

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

Arguments

usecs	The number of microseconds for the process to sleep.

Return

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

edt_alloc

Description

Convenience routine to allocate memory in a system-independent way. The buffer returned is page aligned. Uses VirtualAlloc on Windows 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 Code Fontparatextefault ¶ Fonton page **Error! Bookmark not defined.** 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 Code Fontparatextefault ¶ Font 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);
See Also
edt_msg_output
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(

See Also

edt_msg_init, edt_msg_set_function, edt_msg_set_level, edt_msg

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

See Also

edt_msg_init

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

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

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'll need to understand how the clock generator operates. Figure diagrams how the final value is generated.

Figure 1. Legend. Frequency values:

f_{xtal} - PCI GP-20 is 10 MHz or PCI GP-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.

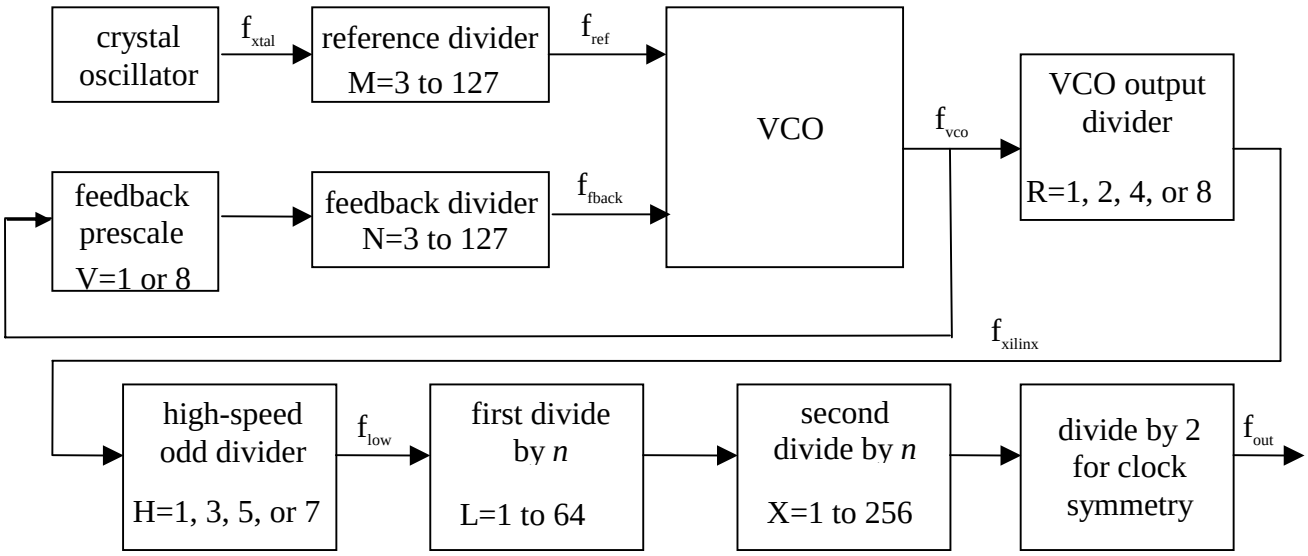

Figure 2. Output Clock Generation Block Diagram

The formula for calculating f_{out} is:

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

edt_find_vco_frequency_ics307

Description

Computes the phased-lock loop parameter for the ICS 307 chip, based on an input clock frequency (xtal) and a target frequency (target). The EdtDev pointer is not required; itc can be set to NULL. If the xtal value is 0, there should be an EdtDev pointer which 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 ouput frequency in Hz
xtal	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 which 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 SS board, using parameters computed by edt_find_vco_frequency_ics307. The clock_channel selects the clock channel to which this should be applied.

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	edt_pll structure filled in by edt_find_vco_frequency_ics307 or edt_find_vco_frequency_ics307_nodivide
clock_channel	channel ID (0-3)

edt_set_frequency_ics307

Description

This is a convenience function which first calls edt_find_vco_frequency_ics307 to compute the parameters for the ICS 307 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.

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	desired output frequency in Hz
xtal	base frequency of the PCI SS board. Default is 10.3681 MHz
clock_channel	channel ID (0-3)

Registers

The PCI SS 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 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-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 is as shown in Figure 2 and described below.

Address Bits	31 16	15 0
0x00	Device ID = 0x40 (for pciss1) 0x41 (for pciss16) 0x42 (for pciss4)	Vendor ED = 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)

Values for the status and command fields are shown in Tables 3 and 4. For complete descriptions of the bits in the status and command fields, see the *PCI Local Bus Specification*, Revision 2.2, 1998, 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 3. Configuration Space Status Field Values

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

Table 4. Configuration Space Status Field Values

PCI Local Bus Addresses

Table 3 describes the PCI SS 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.

See the addenda for registers specific to your configuration.

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.*

A 4-channel PCI uses 0, 20, 40, 60; a 16-channel PCI uses 200, 220, 240, etc. (200 + [channel number x 20]).

The appropriate PCI Xilinx bitfile to be used is determined by your particular application. If you use pciss1 or pciss4, refer to the 4-channel table below; if you use pciss16, refer to the 16-channel table below.

4-Channel

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	
0x60 – 7c (channel 3) 0x40 – 5c (channel 2) 0x20 –3c (channel 1)	Channels 1, 2, and 3 are set up the same as Channel 0, starting at their respective offset.	
0x1C	scatter-gather DMA next count and control (channel 0)	
0x18	scatter-gather DMA current count and control (channel 0)	
0x14	scatter-gather DMA next address (channel 0)	
0x10	scatter-gather DMA current address (channel 0)	
0x0C	main DMA next count and control (channel 0)	
0x08	main DMA current count and control (channel 0)	
0x04	main DMA next address (channel 0)	
0x00	main DMA current address (channel 0)	

Byte Word	3 2	1 0
	1	0

Table 5. 4-Channel PCI Local Bus Addresses

16-Channel

Address Bits	31 16	15 0
0x3E0 – 3FC (channel 15) 0x3CO – 3DC (channel 14) 0x3A0 – 3BC (channel 13) 0x380 – 39C (channel 12) 0x360 – 37C (channel 11) 0x340 – 35C (channel 10) 0x320 – 33C (channel 9) 0x300 – 31C (channel 8) 0x2E0 – 2FC (channel 7) 0x2C0 – 2DC (channel 6) 0x2A0 – 2BC (channel 5) 0x280 – 29C (channel 4) 0x260 – 27C (channel 3) 0x240 – 25C (channel 2)	Channels 1 through 15 are set up the same as Channel 0, starting at their respective offset.	

Address Bits	31 16	15 0
0x220 – 23C (channel 1)		
0x21C	scatter-gather DMA next count and control (channel 0)	
0x218	scatter-gather DMA current count and control (channel 0)	
0x214	scatter-gather DMA next address (channel 0)	
0x210	scatter-gather DMA current address (channel 0)	
0x20C	main DMA next count and control (channel 0)	
0x208	main DMA current count and control (channel 0)	
0x204	main DMA next address (channel 0)	
0x200	main DMA current address (channel 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	

Byte Word	3 2	1 0
	1	0

Table 6. 4-Channel 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 PCI SS 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 6):

- 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 7. 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 68.

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 68.

Bit	EDT_	Description
D31	EN_RDY	Enable scatter-gather next empty interrupt. A value of 1 enables DMA_START (bit 29 of this register) to set DMA_INT (bit 12 of the Status register), thus causing an interrupt if the PCI_EN_INTR bit is set (bit 15 of the Main DMA Command and Configuration register). A value of 0 disables the DMA_START from causing an interrupt.
D30	DMA_DONE	Read-only: a value of 0 indicates that a scatter-gather DMA transfer is currently in progress. A value of 1 indicates that the current scatter-gather DMA is complete.
D29	DMA_START	Write a 1 to this bit to indicate that the values of this register and the SG DMA Next Address register are valid; this sets this bit to 0, indicating either that the copy is in progress, or that the device is waiting for the current DMA to complete. In either case, this register and the SG DMA Next Address register are not available for writing. Reading a value of 1 indicates that the SG DMA Next Count and SG DMA Next Address registers have been copied into the SG DMA Current Count and SG DMA Current Address registers and that the Next Count and Next Address registers are once more available for writing.
D28	EN_MN_DONE	A value of 1 enables the main DMA page done interrupt (bit 18).
D27	EN_SG_DONE	Enable scatter-gather DMA done interrupt. A value of 1 enables DMA_DONE (bit 30 of this register) to set DMA_INT (bit 12 of the Status register), thus causing an interrupt if the PCI_EN_INTR bit is set (bit 15 of the Main DMA Command and Configuration register). A value of 0 disables the DMA_DONE from causing an interrupt.
D26	DMA_ABORT	A value of 1 stops the DMA transfer in progress and cancels the next one, clearing bits 29 and 30. Always 0 when read.
D25	DMA_MEM_RD	A value of 1 specifies a read operation; 0 specifies write.

D24	BURST_EN	A value of 0 means bytes are written to memory as soon as they are received. A value of 1 means bytes are saved to write the most efficient number at once.
D23	MN_DMA_DONE	Read only: a value of 1 indicates that the main DMA is not active.
D22	MN_NXT_EMP	Read only: a value of 1 indicates that the main DMA next address and next count registers are empty.
D21–19		Reserved for EDT internal use.
D18	PG_INT	Read-only: a value of 1 indicates that the page interrupt is set (enabled by bit 28 of this register), and that the main DMA has completed transferring a page for which bit 16 (the page interrupt bit) was set in the scatter-gather DMA list (see Figure 6). If the PCI interrupt is enabled (bit 15 of the PCI interrupt and remote Xilinx configuration register), this bit causes a PCI interrupt. Clear this bit by disabling the page done interrupt (bit 28 of this register).
D17	CURPG_INT	Read-only: a value of 1 indicates that bit 16, the page interrupt bit, was set in the scatter-gather DMA list entry for the current main DMA page.
D16	NXTPG_INT	Read-only: a value of 1 indicates that bit 16, the page interrupt bit, was set in the scatter-gather DMA list entry for the next main DMA page.
D15–0		The number of bytes in the next scatter-gather DMA list.

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: 1. Clear, then set bit D3 of the remote Xilinx command register. 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	RMT_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.

Specifications

PCI Bus Compliance	
Number of Slots	1
Transfer Size	Maximum 64 bytes per transfer
DVMA master	Yes
PCI Bus memory space	Approximately 66 KB
Clock Rate	33 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 Nonoperating: -20 to 60° C
Humidity	Operating: 20 to 80% noncondensing at 40° C Nonoperating: 95% noncondensing at 40°C
Physical	
Dimensions	3.3” x 5.78” x 0.5”
Weight	3.5 oz

Table 8. PCI Bus Configurable DMA Interface Specifications