EDT Application Programming Interface

Engineering Design Team, Inc.
http://www.edt.com

April 30, 2018

Contents

Introduction

The EDT C routines are separated into a few general libraries that work across all boards, as well as some board-specific libraries.

EDT API reference in PDF format

Libraries

EDT DMA Library : Low-level DMA routines for all boards.

EDT Digital Imaging Library : Routines for image capture, save, and device control for EDT Digital Imaging boards.

EDT Camera Link Simulator Library : Routines for camera Link simulation (output) for EDT CLS series boards.

EDT Message Handler Library : Generalized error- and message-handling for all boards. board.

OCM/OC192 Library : Routines and registers specific to the OCM and OC192 mezzanine boards.

EDT Time Library : For controlling the EDT Time functions with certain SS/GS bitfiles.

Note: If you are viewing this document on a CD or mirror site, please note that the latest version of this document (which also includes a search feature) is available at http://www.edt.com/manuals/api.

Terms of Use

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

Engineering Design Team, Inc. ("EDT"), makes no warranties, express or implied, including without limitation the implied warranties of merchantibility and fitness for a particular purpose, regarding the software described in this document ("the software"). EDT does not warrant, guarantee, or make any representations regarding the use or the results of the use of the software in terms of its correctness, accuracy, reliability, currentness, or otherwise. The entire risk

as to the results and performance of the software is assumed by you. The exclusion of implied warranties is not permitted by some jurisdictions. The above exclusion may not apply to you.

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

Copyright, Trademarks

EDT DMA Library

The DMA library provides a set of consistent routines across many of the EDT products, with simple yet powerful ring-buffered DMA (Direct Memory Access) capabilities.

A DMA transfer can be continuous or noncontinuous:

> For noncontinuous transfers, the driver uses DMA system calls read() and write(). Each read() or write() system call performs one DMA transfer. These calls allocate kernel resources, during which time DMA transfers are interrupted.
>
> To perform continuous transfers, use the ring buffers – 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 configurable ring buffer parameters. See the sample programs simple_getdata.c and simple_putdata.c (in the installation directory) for examples of using the ring buffers.

Note:
When developing applications for EDT digital image capture boards such as the PCIe8 DV C-Link, programmers should avoid direct access to the edt library (edt_ subroutines) and instead use the higher level EDT Digital Imaging Library. Some limited use of edtlib calls may be necessary in DV applications, however we can not provide support for applications that directly call edtlib subroutines for data acquisition (e.g. edt_configure_ring_buffers, edt_start_buffers); instead use the pdvlib corollaries (e.g. pdv_multibuf, pdv_start_images).
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.

Building and using the Library, Utilities and Example Applications By default, EDT's pcd installation package is copied into c: (Windows), or /opt/EDTpdv (Linux / Macos). For pdv packages, see the EDT Digital Imaging Library.

Note:
Applications using EDT boards must be linked with the appropriate (32 or 64-bit) for the platform in use. Applications linked with 32-bit EDT libraries will not run correctly on 64-bit systems, or vice-versa.

To rebuild a program or library, you'll need to use a compiler and either the `nmake` application that comes with Visual Studio, or the Unix `make` utility, as described below.

1. Do one of the following:

   For Linux or MacOS, navigate to the installation directory in a terminal window.

   For Windows, click on the **PCD** Utilities or **PDV** Utilities desktop icon to bring up a command window in the installation directory. If Visual Studio environment variables aren't set, you will need to do something like the following. This example assumes Visual Studio 8; consult Microsoft's documentation for other versions:

   ```
   c:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\vcvarsall.bat amd64
   ```

   to build for 64-bit, or if you are building for 32-bit,

   ```
   c:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\vcvarsall.bat x86
   ```

   (Hint: you may find it convenient to configure a Windows Command prompt to open and run the above automatically, e.g. by modifying the Properties >> Target: to be *comspec% /k ""c: Files (x86) Visual Studio 9.0.bat"" amd64*.)

2. Enter

```
make file
```

where *file* is the name of the example program you wish to build. 3. To rebuild all the libraries, examplees, utilities and diagnostics, run

```
make
```

Alternately, on Windows you can use a Visual Studio. Releases are all built using makefiles; reference `includes.mk` and `makefile.def` for lists of the library objects (which all have .c source files), applications, and header files.

Elements of EDT Interface Applications Applications that perform continuous transfers typically include the following elements:

1. The preprocessor statement:

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

2. A call to edt_open to open the device. This returns a pointer to a structure that represents the EDT board in software. All subsequent calls will use this pointer to access the board.

3. Optionally, setup for writing a file or some other target for the data to be acquired.
4. A call to edt_configure_ring_buffers to configure the ring buffers.
5. A call to start the DMA, such as edt_start_buffers.
6. Data processing calls, as required.
7. A call to edt_close to close the device.
8. Appropriate settings in your makefile or C workspace to compile and link the library file libedt.c.

Example

```
#include "edtinc.h"
main()
{
    EdtDev *edt_p = edt_open("pcd", 0) ;
    char *buf_ptr; int outfd = open("outfile", 1) ;
    // Configure a ring buffer with four 1MB buffers
    edt_configure_ring_buffers(edt_p, 1024*1024, 4, EDT_READ, NULL) ;
    edt_start_buffers(edt_p, 0) ; // 0 starts unlimited buffer DMA

    // This loop will capture data indefinitely, but the write() (or
    // other data processing) 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 that perform noncontinuous transfers typically include the following elements:

1. The preprocessor statement:

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

2. A call to edt_open to open the device. This returns a pointer to a structure that represents the EDT board in software. All subsequent calls will use this pointer to access the board.
3. Optionally, setup for writing a file or some other target for the data to be acquired.
4. A system read() or write() call to cause one DMA transfer.
5. Data processing calls, as required.
6. A call to edt_close to close the device.

7. Appropriate settings in your makefile or C workspace to compile and link the library file libedt.c.

Assuming that a multichannel FPGA configuration file has been loaded, this example opens a specific DMA channel with edt_open_channel:

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

    // Because read()s are noncontinuous, without hardware
    // handshaking, the data will have gaps between each read().
    while ((numbytes = edt_read(edt_p, buf, 1024)) > 0)
        write(outfd, buf, numbytes) ;

    edt_close(edt_p) ;
}
```

You can use ring buffer mode for real-time data capture using a small number of buffers (typically 1 MB) configured in a round-robin data FIFO. During capture, the application must be able to transfer or process the data before data acquisition wraps around and overwrites the buffer currently being processed. The example below shows real-time data capture using ring buffers, although it includes no error-checking. In this example, `process_data(bufptr)` must execute in the same amount of time it takes DMA to fill a single buffer, or faster.

```
#include "edtinc.h"
main()
{
  EdtDev *edt_p = edt_open("pcd", 0) ;

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

Check compiler options in the EDT-provided makefiles.

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

1. Because kernel DMA resources are allocated on a per-thread basis and must be allocated and released in the same thread, perform all DMA operations in the same thread as edt_open and edt_close with respect to each channel. Other threads can open the same channel concurrently with DMA, but must perform no DMA-related operations.
2. To avoid undefined application or system behavior, or even system crashes, when exiting the program:

   Join all threads spawned by a main program with the main program after they exit and before the main program exits; or:

   If the main program does not wait for the child threads to exit, then any program that is run following the main program must wait for all the child threads to exit. This waiting period depends on system load and availability of certain system resources, such as a hardware memory management unit.

Modules

Startup / Shutdown

These functions are used to open and close the EDT device.

Initialization
FIFO Flushing

First-in, first-out (FIFO) memory buffers are used to smooth data transmission between different types of data sinks internal to EDT boards.

Input/Output

These functions are used to perform and control DMA transfers.

Register Access

Register access functions.

Utility

Utility functions.

Startup / Shutdown

These functions are used to open and close the EDT device.

Functions

int edt_close (EdtDev *edt_p)
int edt_get_port (EdtDev *edt_p)
Routine to get the "port" number, as distinct from the dma channel.

EdtDev * edt_open (const char *device_name, int unit)
Opens the specified EDT Product and sets up the device handle.

EdtDev * edt_open_channel (const char *device_name, int unit, int channel)
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.

EdtDev * edt_open_device (const char *device_name, int unit, int channel, int verbose)
Opens an EDT device.

EdtDev * edt_open_quiet (const char *device_name, int unit)
Just a version of edt_open that does so quietly, so we can try opening the device just to see if it's there without a lot of printfs coming out.

---

void edt_set_port (EdtDev *edt_p, int port)
Routine to set the "port" number, as distinct from the dma channel.

Function Documentation

*EdtDev** edt_open (const char * device_name, int unit)

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

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. When finished, use edt_close to release any resources allocated during use.

Parameters:
device_name a string with the name of the EDT Product board; for example, "pcd". `EDT_INTERFACE` can also be used; it is defined as the name of the board type in `edtdef.h`.

unit Unit number of the device (if multiple devices). The first unit is always 0.

See also:
edt_open_channel, edt_open_quiet, edt_close

Returns:
A pointer to the EdtDev structure if successful. This data structure holds information about the device which is needed by library functions. User applications should avoid accessing structure elements directly. NULL is returned if unsuccessful, and the global variable errno is set. Use edt_perror to print an error message.

Definition at line 643 of file libedt.c.

*EdtDev** *edt_open_channel (const char* * **device_name***, int* **unit***, int* **channel***)*

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.

To open a device with only one channel, just use edt_open.

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. When finished, use edt_close to release any resources allocated during use.

Parameters:
device_name a string with the name of the EDT Product board; for example, "pcd". `EDT_INTERFACE` can also be used; it is defined as the name of the board type in `edtdef.h`.

unit Unit number of the device (if multiple devices). The first unit is always 0.

channel specifies DMA channel number (counting from zero).

Returns:
A pointer to the EdtDev structure if successful. This data structure holds information about the device which is needed by library functions. User applications should avoid accessing structure elements directly. NULL is returned if unsuccessful, and the global variable errno is set. Use edt_perror to print an error message.

See also:
edt_open, edt_open_quiet, edt_close

Definition at line 699 of file libedt.c.

EdtDev∗ *edt_open_device (const char* ∗ **device_name***, int* **unit***, int* **channel***, int* **verbose***)*

Opens an EDT device.

This call underlies the othere edt_open∗ calls, which basically just map to different variations on calls to this one. For example, edt_open_quiet calls edt_open_device with **verbose** set to 0 (false). User applications should typically use the higher-level calls rather than calling this directly, although there's no real harm in doing so either.

Parameters:

device_name a string with the name of the EDT Product board; for example, "pcd". `EDT_INTERFACE` can also be used; it is defined as the name of the board type in `edtdef.h`.

unit Unit number of the device (if multiple devices). The first unit is always 0.

channel specifies DMA channel number (counting from zero).

verbose when 0, produce no console output. When nonzero, outputs a message on successful open, or an error-specific message on failure

See also:

edt_open, edt_open_channel, edt_open_quiet

Returns:

A pointer to the EdtDev structure if successful. This data

Definition at line 589 of file libedt.c.

EdtDev∗ *edt_open_quiet (const char* ∗ **device_name***, int* **unit***)*

Just a version of edt_open that does so quietly, so we can try opening the device just to see if it's there without a lot of printfs coming out.

Parameters:

device_name a string with the name of the EDT Product board; for example, "pcd". `EDT_INTERFACE` can also be used; it is defined as the name of the board type in `edtdef.h`.

unit Unit number of the device (if multiple devices). The first unit is always 0.

See also:

edt_open, edt_open_channel, edt_close

Returns:

Pointer to EdtDev struct, or NULL if error.

Definition at line 665 of file libedt.c.

Initialization

Functions

int edt_bitload (EdtDev *edt_p, const char *basedir, const char *fname, int flags, int skip)

Searches for and loads a gate array bit file into an EDT PCI board.

---

int edt_bitload_from_prom (EdtDev *edt_p, u_int addr1, int size1, u_int addr2, int sized, int flags)

Bitload from a given address in the PCI PROM.

Function Documentation

*int edt_bitload (EdtDev * edt_p, const char * indir, const char * name, int flags, int skip)*

Searches for and loads a gate array bit file into an EDT PCI board.

Searches under <basedir>/bitfiles/xxx, or if a PCI DV, in the appropriate sub-directory (<basedir>/bitfiles/dv/.../<file>.bit or <basedir>/bitfiles/dvk/.../<file>.bit. '...' stands for all the subdirs found under the base path.) Quits after the first successful load.

Parameters:

edt_p device handle returned from edt_open

basedir base directory to start looking for the file

name name of the bitfile to load

flags misc flag bits – should be combination of BITLOAD_FLAGS_* which are defined in edt_bitload.h. (This variable was formerly rcam which is obsolete.)

skip if nonzero, don't actually load, just find the files (debugging)

Returns:

0 on success, -1 on failure

Definition at line 1460 of file edt_bitload.c.

FIFO Flushing

First-in, first-out (FIFO) memory buffers are used to smooth data transmission between different types of data sinks internal to EDT boards.

For instance, the FIFO stores information processed by the user interface Xilinx until the PCI Xilinx retrieves it across the PCI bus. The PCI bus normally sends information in bursts, so the FIFO allows this same information to be sent smoothly. When acquiring or sending data, flush the FIFO immediately before performing DMA. This also resets the FIFO to an empty state. The following subroutines either flush the FIFO or set it to flush automatically at the start of DMA.

Functions

int edt_disable_channel (EdtDev *edt_p, u_int channel)

Clears a specified mezzanine channel enable bit.

---

int edt_disable_channels (EdtDev *edt_p, u_int mask)

Clears specified mezzanine channel enable bits.

---

int edt_enable_channel (EdtDev *edt_p, u_int channel)

Sets a specified mezzanine channel enable bit.

---

int edt_enable_channels (EdtDev *edt_p, u_int mask)

Sets specified mezzanine channel enable bits.

---

void edt_flush_channel (EdtDev *edt_p, int channel)
void edt_flush_fifo (EdtDev *edt_p)

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

---

int edt_get_firstflush (EdtDev *edt_p)

OBSOLETE.

---

int edt_set_firstflush (EdtDev *edt_p, int val)

Tells whether and when to flush the FIFOs before DMA transfer.

Function Documentation

*int edt_disable_channel (EdtDev * edt_p, u_int channel)*

Clears a specified mezzanine channel enable bit.

Parameters:
edt_p
channel

Returns:
0 on success, -1 on failure

This function disables a DMA channel specified by the second argument.

Definition at line 9331 of file libedt.c.

int edt_disable_channels (EdtDev * **edt_p**, *u_int* **mask**)

Clears specified mezzanine channel enable bits.

Parameters:
edt_p
channel

Returns:
0 on success, -1 on failure

This function disables DMA channels specified by the bitmask in second argument.

Definition at line 9262 of file libedt.c.

int edt_enable_channel (EdtDev * **edt_p**, *u_int* **channel**)

Sets a specified mezzanine channel enable bit.

Parameters:
edt_p
channel

Returns:
0 on success, -1 on failure

This function enables a DMA channel specified by the second argument.

Definition at line 9295 of file libedt.c.

int edt_enable_channels (EdtDev * **edt_p**, *u_int* **mask**)

Sets specified mezzanine channel enable bits.

Parameters:
edt_p
channel

Returns:
0 on success, -1 on failure

This function enables DMA channels specified by the bitmask in second argument.

Definition at line 9229 of file libedt.c.

*void edt_flush_fifo (EdtDev * edt_p)*

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

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Definition at line 4023 of file libedt.c.

*int edt_get_firstflush (EdtDev * edt_p)*

OBSOLETE.

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).

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Example

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

Returns:
The value set by edt_set_firstflush().

See also:
edt_set_firstflush

Definition at line 4692 of file libedt.c.

*int edt_set_firstflush (EdtDev * edt_p, int flag)*

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.

Parameters:

edt_p pointer to edt device structure returned by 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

Returns:

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

Definition at line 4662 of file libedt.c.

Input/Output

These functions are used to perform and control DMA transfers.

Functions

int edt_abort_current_dma (EdtDev *edt_p)

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

---

int edt_abort_dma (EdtDev *edt_p)

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

---

unsigned int edt_allocated_size (EdtDev *edt_p, int bufnum)

Gets the allocated size of the specified buffer.

---

unsigned char ** edt_buffer_addresses (EdtDev *edt_p)

Returns an array containing the addresses of the ring buffers.

---

unsigned char * edt_check_for_buffers (EdtDev *edt_p, uint_t count)

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

---

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

Configures the EDT device ring buffers.

---

int edt_configure_block_buffers_mem (EdtDev *edt_p, int bufsize, int numbufs, int write_flag, int header_size, int header_before, u_char *user_mem)

Identical to edt_configure_block_buffers, with the additional parameter user_mem*, which allows the user to specify a block of pre-allocated memory to use (Note: this does not work on Linux).*

---

int edt_configure_ring_buffers (EdtDev *edt_p, int bufsize, int numbufs, int write_flag, unsigned char **bufarray)

Configures the EDT device ring buffers.

---

int edt_disable_ring_buffers (EdtDev *edt_p)

Disables the EDT device ring buffers.

---

int edt_do_timeout (EdtDev *edt_p)

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).

---

bufcnt_t edt_done_count (EdtDev ∗edt_p)

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

---

void edt_enddma_action (EdtDev ∗edt_p, uint_t val)

Specifies when to perform the action at the end of a dma transfer as specified by edt_enddma_reg.

---

void edt_enddma_reg (EdtDev ∗edt_p, uint_t desc, uint_t val)

Sets the register and value to use at the end of dma, as set by edt_enddma_action.

---

int edt_get_burst_enable (EdtDev ∗edt_p)

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.

---

uint_t edt_get_bytecount (EdtDev ∗edt_p)

OBSOLETE: Use edt_get_bufbytecount(edt_p, &bufnum) instead.

---

unsigned char ∗ edt_get_current_dma_buf (EdtDev ∗edt_p)

edt_current_dma_buf

---

unsigned short edt_get_direction (EdtDev ∗edt_p)

Gets the value of the PCD_DIRA and PCD_DIRB registers.

---

int edt_get_goodbits (EdtDev ∗edt_p)

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

---

int edt_get_numbufs (EdtDev ∗edt_p)

edt_get_numbufs

---

int edt_get_reftime (EdtDev ∗edt_p, u_int ∗timep)

Gets the seconds and nanoseconds timestamp in the same format as the buffer_timed functions.

---

int edt_get_rtimeout (EdtDev ∗edt_p)

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

---

uint_t edt_get_timeout_count (EdtDev ∗edt_p)

Returns the number of bytes transferred at last timeout.

---

int edt_get_timeout_goodbits (EdtDev *edt_p)

Returns the number of good bits in the last long word of a read buffer after the last timeout.

---

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

Gets the seconds and nanoseconds timestamp of when dma was completed on the buffer specified by bufnum*.*

---

uint_t edt_get_todo (EdtDev *edt_p)

Gets the number of buffers that the driver has been told to acquire.

---

int edt_get_total_bufsize (EdtDev *edt_p, int bufsize, int header_size)

edt_get_total_bufsize

---

int edt_get_wtimeout (EdtDev *edt_p)

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

---

unsigned char * edt_last_buffer (EdtDev *edt_p)

Waits for the last buffer that has been transferred.

---

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

Like edt_last_buffer but also returns the time at which the DMA was complete on this buffer.

---

caddr_t edt_map_dmamem (EdtDev *edt_p)
unsigned char * edt_next_writebuf (EdtDev *edt_p)

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

---

uint_t edt_next_writebuf_index (EdtDev *edt_p)

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

---

int edt_read (EdtDev *edt_p, void *buf, uint_t size)

Performs a read on the EDT Product.

---

void edt_read_end_action (EdtDev *edt_p, u_int enable, u_int reg_desc, u_char set, u_char clear, u_char setclear, u_char clearset, int delay1, int delay2)

Enables an action where a specified register will be programmed with a specified value at the end of a dma read operation.

---

void edt_read_start_action (EdtDev *edt_p, u_int enable, u_int reg_desc, u_char set, u_char clear, u_char setclear, u_char clearset, int delay1, int delay2)

Enables an action where a specified register will be programmed with a specified value at the start of a dma read operation.

---

int edt_ref_tmstamp (EdtDev *edt_p, u_int val)

Causes application-defined events to show up in the same timeline as driver events when the event history is listed by running `setdebug -g`.

---

int edt_remove_event_func (EdtDev *edt_p, int event_type)

Removes an event function previously set with edt_set_event_func.

---

int edt_reset_ring_buffers (EdtDev *edt_p, uint_t bufnum)

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

---

int edt_ring_buffer_overrun (EdtDev *edt_p)

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

---

int edt_set_buffer (EdtDev *edt_p, uint_t bufnum)

Sets which buffer should be started next.

---

int edt_set_buffer_physaddr (EdtDev *edt_p, uint_t index, uint64_t physaddr)
int edt_set_buffer_size (EdtDev *edt_p, uint_t which_buf, uint_t size, uint_t write_flag)

Used to change the size or direction of one of the ring buffers.

---

int edt_set_burst_enable (EdtDev *edt_p, int on)

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.

---

void edt_set_direction (EdtDev *edt_p, int direction)

On PCD cards, sets DMA direction to read or write.

---

void edt_set_dmy_reg_read_callback (EdtDev *edt_p, u_int(*callBack)(struct edt_device *edt_p, u_int reg_desc))

When "dmy" or "DMY" is passed to edt_open(), edt_p->devid is set to DMY_ID and all attempted interaction with EDT hardware is ignored.*

---

void edt_set_dmy_reg_write_callback (EdtDev *edt_p, void(*callBack)(struct edt_device *edt_p, u_int reg_desc, u_int reg_value))

When "dmy" or "DMY" is passed as the first argument to edt_open(), edt_p->devid is set to DMY_ID and all attempted interaction with EDT hardware is ignored.*

---

void edt_set_dmy_wait_for_buffers_callback (EdtDev *edt_p, void(*callBack)(struct edt_device *edt_p, u_char *buf))

When "dmy" or "DMY" is passed as the first argument to edt_open(), edt_p->devid is set to DMY_ID and all attempted interaction with EDT hardware is ignored.*

---

int edt_set_event_func (EdtDev *edt_p, int event_type, EdtEventFunc f, void *data, int continuous)

Defines a function to call when an event occurs.

---

int edt_set_rtimeout (EdtDev *edt_p, int value)

Sets the number of milliseconds for data read calls, such as edt_read, to wait for DMA to complete before returning.

---

int edt_set_timeout_action (EdtDev *edt_p, u_int action)

Sets the driver behavior on a timeout.

---

int edt_set_wtimeout (EdtDev *edt_p, int value)

Sets the number of milliseconds for data write calls, such as edt_write, to wait for DMA to complete before returning.

---

int edt_start_buffers (EdtDev *edt_p, uint_t count)

Starts DMA to the specified number of buffers.

---

void edt_startdma_action (EdtDev *edt_p, uint_t val)

Specifies when to perform the action at the start of a dma transfer as specified by edt_startdma_reg.

---

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

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

---

int edt_stop_buffers (EdtDev *edt_p)

Stops DMA transfer after the current buffer has completed.

---

int edt_timeouts (EdtDev *edt_p)

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

unsigned char * edt_wait_buffers_timed (EdtDev *edt_p, int count, u_int *timep)

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

unsigned char * edt_wait_for_buffers (EdtDev *edt_p, int count)

Blocks until the specified number of buffers have completed.

unsigned char * edt_wait_for_next_buffer (EdtDev *edt_p)

Waits for the next buffer that finishes DMA.

int edt_write (EdtDev *edt_p, void *buf, uint_t size)

Perform a write on the EDT Product.

void edt_write_end_action (EdtDev *edt_p, u_int enable, u_int reg_desc, u_char set, u_char clear, u_char setclear, u_char clearset, int delay1, int delay2)

Enables an action where a specified register will be programmed with a specified value at the end of a dma write operation.

void edt_write_start_action (EdtDev *edt_p, u_int enable, u_int reg_desc, u_char set, u_char clear, u_char setclear, u_char clearset, int delay1, int delay2)

Enables an action where a specified register will be programmed with a specified value at the start of a dma write operation.

Function Documentation

*int edt_abort_current_dma (EdtDev * edt_p)*

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

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:

0 on sucess, -1 on error. If an error occurs, call edt_perror to get the system error message.

See also:

edt_abort_dma

Definition at line 6004 of file libedt.c.

*int edt_abort_dma (EdtDev * edt_p)*

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

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
0 on sucess, -1 on error. If an error occurs, call edt_perror to get the system error message.

See also:
edt_abort_current_dma

Definition at line 5982 of file libedt.c.

*unsigned int edt_allocated_size (EdtDev * edt_p, int buffer)*

Gets the allocated size of the specified buffer.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

buffer the index of the buffer.

Returns:
The buffer size, in bytes, or 0 if the specified index is invalid.

Definition at line 1949 of file libedt.c.

*unsigned char** edt_buffer_addresses (EdtDev * edt_p)*

Returns an array containing the addresses of the ring buffers.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
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.

Definition at line 2202 of file libedt.c.

*unsigned char** edt_check_for_buffers (EdtDev * edt_p, uint_t count)

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

Parameters:

edt_p pointer to edt device structure returned by edt_open

count number of buffers. Must be 1 or greater. Four is recommended.

Returns:

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.

Definition at line 2749 of file libedt.c.

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

Configures the EDT device ring buffers.

Any previous configuration is replaced, and previously allocated buffers are released. Buffers are normally allocated and maintained within the EDT device library (bufarray = NULL).

Note:

bufarray can alternately point to an array of user buffers which will be used instead of the internally allocated ones, however **it will fail** (possibly with a system crash) if the system has more than 4 GBytes of memory. Since > 4 GBytes is becoming ubiquitous, providing user buffers has effectively been deprecated. The argument remains in order to maintain code constency, nevertheless **EDT can not provide support for any applications that provide a non-NULL argument in bufarray.**

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

bufsize size of each buffer, in bytes. 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.

numbufs number of buffers. Must be 1 or greater. Four is recommended for most applications.

write_flag 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 **(Deprecated – see note above)** this argument is an array of pointers to application-allocated and page aligned buffers (use edt_alloc to allocate page alligned buffers); these buffers must match the size and number of buffers specified in this call and will be used as the ring buffers.

Returns:
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.

Definition at line 1773 of file libedt.c.

int edt_configure_block_buffers_mem *(EdtDev* * edt_p*, int* bufsize*, int* numbufs*, int* write_flag*, int* header_size*, int* header_before*, u_char* * user_mem*)*

Identical to edt_configure_block_buffers, with the additional parameter *user_mem*, which allows the user to specify a block of pre-allocated memory to use (Note: this does not work on Linux).

Users are encourage to use edt_configure_block_buffers rather than this function, as that function handles allocation of memory and works on all systems.

Definition at line 1667 of file libedt.c.

int edt_configure_ring_buffers *(EdtDev* * edt_p*, int* bufsize*, int* numbufs*, int* write_flag*, unsigned char* ** bufarray*)*

Configures the EDT device ring buffers.

Any previous configuration is replaced, and previously allocated buffers are released. Buffers are normally allocated and maintained within the EDT device library (bufarray = NULL).

Note:
bufarray can alternately point to an array of user buffers which will be used instead of the internally allocated ones, however **it will fail** (possibly with a system crash) if the system has more than 4 GBytes of memory. Since > 4 GBytes is becoming ubiquitous, providing user buffers has effectively been deprecated. The argument remains in order to maintain code constency, nevertheless **EDT can not provide support for any applications that provide a non-NULL argument in bufarray.**

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

bufsize size of each buffer, in bytes. 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.

numbufs number of buffers. Must be 1 or greater. Four is recommended for most applications.

write_flag 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 **(Deprecated – see note above)** this argument is an array of pointers to application-allocated and page aligned buffers (use edt_alloc to allocate page alligned buffers); these buffers must match the size and number of buffers specified in this call and will be used as the ring buffers.

Returns:

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.

Definition at line 1641 of file libedt.c.

*int edt_disable_ring_buffers (EdtDev * edt_p)*

Disables the EDT device ring buffers.

Pending DMA is cancelled and all buffers are released.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:

0 on success, -1 on error. If an error occurs, call edt_perror to get the system error message.

Definition at line 1862 of file libedt.c.

*int edt_do_timeout (EdtDev * edt_p)*

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.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
0 on success; -1 on failure

See also:
ring buffer discussion

Definition at line 2366 of file libedt.c.

*bufcnt_t edt_done_count (EdtDev * edt_p)*

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

Parameters:
edt_p pointer to edt device structure returned by edt_open

Returns:
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.

Definition at line 2782 of file libedt.c.

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

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.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

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);
```

See also:

edt_startdma_action, edt_startdma_reg, edt_reg_write, edt_reg_read

Definition at line 3138 of file libedt.c.

*void edt_enddma_reg (EdtDev * edt_p, uint_t desc, uint_t val)*

Sets the register and value to use at the end of dma, as set by edt_enddma_action.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

desc register description of which register to use as in `edtreg.h`.

val value to write

See also:

edt_enddma_action for example

Definition at line 3188 of file libedt.c.

*int edt_get_burst_enable (EdtDev * edt_p)*

Returns the value of the burst enable flag, determining whether the DMA master transfers as many words as possible at once, or transfers them one at a time as soon as the data is acquired.

Burst transfers are enabled by default to optimize use of the bus. For more information, see edt_set_burst_enable.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:

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

Definition at line 3668 of file libedt.c.

*uint_t edt_get_bytecount (EdtDev * edt_p)*

OBSOLETE: Use edt_get_bufbytecount(edt_p, &bufnum) instead.

Obsoleted 04/2013 in favor of edt_get_buf_bytecount since it fails to identify offset and buffer atomically.

Returns the number of bytes read so far into the current buffer. Can be used to monitor how much data has been read into the buffer during acquisition.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
The number of bytes transferred, as described above.

Definition at line 4088 of file libedt.c.

unsigned char edt_get_current_dma_buf (EdtDev * edt_p)*

edt_current_dma_buf

Returns the address of the current active DMA buffer, for linescan cameras where the buffer is only partially filled. Note there is a possible error if this is called with normal DMA that doesn't time out, because the "current" buffer may change between a call to this function and the pointer's access.

Parameters:
edt_p,: device handle returned from edt_open

Definition at line 2705 of file libedt.c.

*unsigned short edt_get_direction (EdtDev * edt_p)*

Gets the value of the PCD_DIRA and PCD_DIRB registers.

The value from PCD_DIRB is shifted up 8 bits.

Definition at line 4122 of file libedt.c.

*int edt_get_goodbits (EdtDev * edt_p)*

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

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
Number 0-31 representing the number of good bits in the last 32-bit word of the current read buffer.

Definition at line 4999 of file libedt.c.

*int edt_get_numbufs (EdtDev * edt_p)*

edt_get_numbufs

returns the number of buffers allocated even if by other process (for monitoring from a separate call to edt_open)

Definition at line 980 of file libedt.c.

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

Gets the seconds and nanoseconds timestamp in the same format as the buffer_timed functions.

Used for debugging and coordinating dma completion time with other events.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

timep pointer to an unsigned integer array

Example

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

Returns:

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

See also:

edt_timestamp, edt_done_count, edt_wait_buffers_timed

Definition at line 6715 of file libedt.c.

*int edt_get_rtimeout (EdtDev * edt_p)*

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

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:

The number of milliseconds in the current read timeout period.

Definition at line 4522 of file libedt.c.

*uint_t edt_get_timeout_count (EdtDev * edt_p)*

Returns the number of bytes transferred at last timeout.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
The number of bytes transferred at last timeout.

Definition at line 4108 of file libedt.c.

*int edt_get_timeout_goodbits (EdtDev * edt_p)*

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.)

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

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

Definition at line 4969 of file libedt.c.

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

Gets the seconds and nanoseconds timestamp of when dma was completed on the buffer specified by *bufnum*.

bufnum is moduloed by the number of buffers in the ring buffer, so it can either be an index, or the number of buffers completed.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

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);
```

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

Definition at line 2573 of file libedt.c.

*uint_t edt_get_todo (EdtDev * edt_p)*

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.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

```
int curdone;
int curtodo;
curdone = edt_done_count(edt_p);
curtodo = edt_get_todo(edt_p);
// curtodo - curdone is how close the DMA is to catching up with our
// processing
```

Returns:
Number of buffers started via edt_start_buffers.

See also:
edt_done_count, edt_start_buffers, edt_wait_for_buffers

Definition at line 6663 of file libedt.c.

*int edt_get_total_bufsize (EdtDev * edt_p, int bufsize, int header_size)*

edt_get_total_bufsize

returns the total buffer size for block of buffers, in which the memory allocation size is rounded up so all buffers start on a page boundary. This is used to allocate a single contiguous block of DMA buffers.

Definition at line 1545 of file libedt.c.

*int edt_get_wtimeout (EdtDev * edt_p)*

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

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
The number of milliseconds in the current write timeout period.

Definition at line 4542 of file libedt.c.

unsigned char edt_last_buffer (EdtDev * edt_p)*

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.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
Address of the image.

See also:
edt_wait_for_buffers, edt_last_buffer_timed

Definition at line 2261 of file libedt.c.

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

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 two unsigned integers which will be filled in with the seconds and nanoseconds of the time the buffer was finished being transferred.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

timep pointer to an unsigned integer array

Example

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

Returns:
Address of the image.

See also:
edt_wait_for_buffers, edt_last_buffer, edt_wait_buffers_timed

Definition at line 2305 of file libedt.c.

unsigned char edt_next_writebuf (EdtDev * edt_p)*

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

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
A pointer to the buffer, or NULL on failure.

See also:
edt_next_writebuf_index

Definition at line 2156 of file libedt.c.

*uint_t edt_next_writebuf_index (EdtDev * edt_p)*

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

Increments the next buffer index, so subsequent calls to edt_next_writebuf will return subsequent buffers.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
Index of the buffer, as returned by edt_buffer_addresses, or -1 on failure. If an error occurs, call edt_perror to get the system error message.
Index of the buffer, as returned by edt_buffer_addresses.

Definition at line 2181 of file libedt.c.

*int edt_read (EdtDev * edt_p, void * buf, uint_t size)*

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 2^31 bytes does not fail. This call is not multibuffering, and no transfer is active when it completes.

Parameters:

edt_p pointer to edt device structure returned by edt_open

buf address of buffer to read into

size size of read in bytes

Returns:

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.

Definition at line 2011 of file libedt.c.

void *edt_read_end_action (EdtDev* * **edt_p**, *u_int* **enable**, *u_int* **reg_desc**, *u_char* **set**, *u_char* **clear**, *u_char* **setclear**, *u_char* **clearset**, *int* **delay1**, *int* **delay2**)

Enables an action where a specified register will be programmed with a specified value at the end of a dma read operation.

Enabled with EDT_ACT_ALWAYS and disabled with EDT_ACT_NEVER passed to the enable argument. A common use of this is to write to a register which signals an external device that dma has ended to notify the device to stop sending.

This routine is intended to work with edt_read(). It will not work well ring buffers since sequential dma operations are pipelined in hardware in the EDT dma engine.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

enable EDT_ACT_ALWAYS to enable, EDT_ACT_NEVER to disable.

reg_desc Register access description code.

set Register bits to be set.

clear Register bits to be cleared.

setclear Register value to be toggled up then down.

clearset Register value to be toggled down then up.

Example

```
edt_read_end_action(edt_p, EDT_ACT_ALWAYS, PCD_FUNCT, 0x8F, 0, 0x10, 0);
edt_read_end_action(edt_p, EDT_ACT_NEVER, dummy, dummy, dummy, dummy);
```

See also:
edt_read_start_action(), edt_write_start_action(), edt_write_end_action()

Definition at line 3285 of file libedt.c.

*void edt_read_start_action (EdtDev * edt_p, u_int enable, u_int reg_desc, u_char set, u_char clear, u_char setclear, u_char clearset, int delay1, int delay2)*

Enables an action where a specified register will be programmed with a specified value at the start of a dma read operation.

Enabled with EDT_ACT_ALWAYS and disabled with EDT_ACT_NEVER passed to the enable argument. 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.

This routine is intended to work with edt_read(). It will not work well ring buffers since sequential dma operations are pipelined in hardware in the EDT dma engine.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

enable EDT_ACT_ALWAYS to enable, EDT_ACT_NEVER to disable.

reg_desc Register access description code.

set Register bits to be set.

clear Register bits to be cleared.

setclear Register value to be toggled up then down.

clearset Register value to be toggled down then up.

Example

```
edt_read_start_action(edt_p, EDT_ACT_ALWAYS, PCD_FUNCT, 0x8F, 0, 0x10, 0);
edt_read_start_action(edt_p, EDT_ACT_NEVER, dummy, dummy, dummy, dummy);
```

See also:
edt_read_end_action(), edt_write_start_action(), edt_write_end_action()

Definition at line 3234 of file libedt.c.

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

Causes application-defined events to show up in the same timeline as driver events when the event history is listed by running `setdebug -g`.

This is useful for debugging.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

val an arbitrary value meaningful to the application

Example

```
#define BEFORE_WAIT 0x1212aaaa
#define AFTER_WAIT 0x3434bbbb
u_char *buf;
edt_ref_tmstamp(edt_p, BEFORE_WAIT);
buf=edt_wait_for_buffer(edt_p);
edt_ref_tmstamp(edt_p, AFTER_WAIT);

// now look at output of setdebug -g and you'll see something like:

// 0: 0001ca0 REFTMSTAMP           : 1212aaaa      324.422071 (0.004189)
// ... other events from edt_wait_for_buffer() shown, like START_BUF, SETUP_DMA, FLUSH, etc
// 0: 0001d08 REFTMSTAMP           : 3434bbbb        324.518885 (0.000045)
```

Returns:

0 on success, -1 on failure

See also:

setdebug –help

Definition at line 6336 of file libedt.c.

*int edt_remove_event_func (EdtDev * edt_p, int event_type)*

Removes an event function previously set with edt_set_event_func.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

event_type The event that causes the function to be called. Valid events are as listed in edt_set_event_func.

Returns:

0 on success, -1 on failure. If an error occurs, call edt_perror to get the system error message.

Definition at line 5825 of file libedt.c.

*int edt_reset_ring_buffers (EdtDev * edt_p, uint_t bufnum)*

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

Parameters:

edt_p pointer to edt device structure returned by 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 supplied modulo the number of buffers.

Returns:

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

Definition at line 5950 of file libedt.c.

*int edt_ring_buffer_overrun (EdtDev * edt_p)*

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.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:

1(true) when overrun has occurred, 0(false) otherwise.

Definition at line 6023 of file libedt.c.

*int edt_set_buffer (EdtDev * edt_p, uint_t bufnum)*

Sets which buffer should be started next.

Usually done to recover after a timeout, interrupt, or error.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

bufnum the index of the buffer to start next.

Example

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

Returns:

0 on success, -1 on failure.

See also:
edt_stop_buffers, edt_done_count, edt_get_todo

Definition at line 1981 of file libedt.c.

*int edt_set_buffer_size (EdtDev * edt_p, uint_t index, uint_t size, uint_t write_flag)*

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.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

index index of ring buffer to change

size size to change it to

write_flag direction

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);
```

Returns:
0 on success, -1 on failure

See also:
edt_open_channel, rdpcd8.c, rd16.c, rdssdio.c, wrssdio.c

Definition at line 6545 of file libedt.c.

*int edt_set_burst_enable (EdtDev * edt_p, int onoff)*

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.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

onoff a value of 1 turns the flag on (the default); 0 turns it off.

Definition at line 3648 of file libedt.c.

*void edt_set_direction (EdtDev * edt_p, int direction)*

On PCD cards, sets DMA direction to read or write.

Most users will not need to use this function, but instead can just set the direction when calling edt_configure_ring_buffers.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

direction one of EDT_READ or EDT_WRITE

Definition at line 4163 of file libedt.c.

*void edt_set_dmy_reg_read_callback (EdtDev * edt_p, u_int(*)(struct edt_device *edt_p, u_int reg_desc) callBack)*

When "dmy" or "DMY" is passed to edt_open*(), edt_p->devid is set to DMY_ID and all attempted interaction with EDT hardware is ignored.

This function registers a callback function which is invoked when edt_reg_read() is called to return a simulated register read value.

Example: See the rd16_dmy_register.c sample program.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

callBack callback function invoked when edt_p->devid is DMY_ID and edt_reg_read() is called.

synopsys: u_int (*callBack)(struct edt_device *edt_p, u_int reg_desc)

Definition at line 9698 of file libedt.c.

*void edt_set_dmy_reg_write_callback (EdtDev * edt_p, void(*)(struct edt_device *edt_p, u_int reg_desc, u_int reg_value) callBack)*

When "dmy" or "DMY" is passed as the first argument to edt_open*(), edt_p->devid is set to DMY_ID and all attempted interaction with EDT hardware is ignored.

This function registers a callback function which is invoked when edt_reg_write() is called to set a simulated register write value.

Example: See the wr16_dmy_register.c sample program.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

callBack callback function invoked when edt_p->devid is DMY_ID and edt_reg_write() is called.

synopsys: u_int (*callBack)(struct edt_device *edt_p, u_int reg_desc)

Definition at line 9724 of file libedt.c.

void *edt_set_dmy_wait_for_buffers_callback* *(EdtDev* * **edt_p**, *void(*)(struct edt_device *edt_p, u_char *buf)* **callBack***)*

When "dmy" or "DMY" is passed as the first argument to edt_open*(), edt_p->devid is set to DMY_ID and all attempted interaction with EDT hardware is ignored.

This function registers a callback function which is invoked when edt_wait_for_buffers() is called to populate the ring buffer with data.

Example: See the rd16_dmy_dma.c sample program.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

callBack callback function invoked when edt_p->devid is DMY_ID and edt_wait_for_buffers() is called.

synopsys: void (*callBack)(struct edt_device *edt_p, u_char *buf)

Definition at line 9674 of file libedt.c.

int edt_set_event_func (EdtDev * **edt_p**, *int* **event_type**, *EdtEventFunc* **func**, *void* * **data**, *int* **continuous***)*

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.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

event_type 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.

continuous Flag to enable or disable continued event notification. A value of 0 causes an implied edt_remove_event_func as the event is triggered.

Returns:

0 on success, -1 on failure. If an error occurs, call edt_perror to get the system error message.

*int edt_set_rtimeout (EdtDev * edt_p, int value)*

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 and edt_read.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

value The number of milliseconds in the timeout period.

Returns:

0 on success; -1 on error

Definition at line 4482 of file libedt.c.

*int edt_set_timeout_action (EdtDev * edt_p, u_int action)*

Sets the driver behavior on a timeout.

Parameters:

edt_p pointer to edt device structure returned by 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.

Returns:

0 on success, -1 on failure.

Definition at line 4940 of file libedt.c.

*int edt_set_wtimeout (EdtDev * edt_p, int value)*

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 and edt_write.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

value The number of milliseconds in the timeout period.

Returns:

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

Definition at line 4504 of file libedt.c.

*int edt_start_buffers (EdtDev * edt_p, uint_t count)*

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.

Parameters:

edt_p pointer to edt device structure returned by edt_open

count 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.

Returns:

0 on success, -1 on error. If an error occurs, call edt_perror to get the system error message.

Definition at line 1920 of file libedt.c.

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

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 at 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.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

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);
```

See also:

edt_startdma_reg, edt_reg_write, edt_reg_read

Definition at line 3102 of file libedt.c.

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

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

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

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);
```

See also:

edt_startdma_action

Definition at line 3163 of file libedt.c.

*int edt_stop_buffers (EdtDev * edt_p)*

Stops DMA transfer after the current buffer has completed.

Ring buffer mode remains active, and transfers will be continued by calling edt_start_buffers.

Parameters:

edt_p pointer to edt device structure returned by edt_open

Returns:

0 on success, -1 on failure. If an error occurs, call edt_perror to get the system error message.

Definition at line 5923 of file libedt.c.

*int edt_timeouts (EdtDev * edt_p)*

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

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:

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

Definition at line 4363 of file libedt.c.

*unsigned char** *edt_wait_buffers_timed (EdtDev * edt_p, int count, u_int * timep)*

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

Parameters:

edt_p pointer to edt device structure returned by 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 nanoseconds representing the system time at which the buffer completed.

Returns:

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.

Definition at line 2230 of file libedt.c.

*unsigned char** *edt_wait_for_buffers (EdtDev * edt_p, int count)*

Blocks until the specified number of buffers have completed.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel count: how many buffers to block for

count How many buffers to block for. Completed buffers are numbered relatively; start each call with 1.

Returns:

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.
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.

Definition at line 2426 of file libedt.c.

unsigned char edt_wait_for_next_buffer (EdtDev * edt_p)*

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.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

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

Definition at line 2618 of file libedt.c.

*int edt_write (EdtDev * edt_p, void * buf, uint_t size)*

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 2^31 bytes does not fail. This call is not multibuffering, and no transfer is active when it completes.

Parameters:
edt_p pointer to edt device structure returned by edt_open
buf address of buffer to write from
size size of write in bytes

Returns:
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.

Definition at line 2081 of file libedt.c.

*void edt_write_end_action (EdtDev * edt_p, u_int enable, u_int reg_desc, u_char set, u_char clear, u_char setclear, u_char clearset, int delay1, int delay2)*

Enables an action where a specified register will be programmed with a specified value at the end of a dma write operation.

Enabled with EDT_ACT_ALWAYS and disabled with EDT_ACT_NEVER passed to the enable argument. A common use of this is to write to a register which signals an external device that dma has ended to notify the device to stop sending.

This routine is intended to work with edt_write(). It will not work well ring buffers since sequential dma operations are pipelined in hardware in the EDT dma engine.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

enable EDT_ACT_ALWAYS to enable, EDT_ACT_NEVER to disable.

reg_desc Register access description code.

set Register bits to be set.

clear Register bits to be cleared.

setclear Register value to be toggled up then down.

clearset Register value to be toggled down then up.

Example

```
edt_write_end_action(edt_p, EDT_ACT_ALWAYS, PCD_FUNCT, 0x8F, 0, 0x10, 0);
edt_write_end_action(edt_p, EDT_ACT_NEVER, dummy, dummy, dummy, dummy);
```

See also:

edt_write_start_action(), edt_read_start_action(), edt_read_end_action()

Definition at line 3387 of file libedt.c.

*void edt_write_start_action (EdtDev * edt_p, u_int enable, u_int reg_desc, u_char set, u_char clear, u_char setclear, u_char clearset, int delay1, int delay2)*

Enables an action where a specified register will be programmed with a specified value at the start of a dma write operation.

Enabled with EDT_ACT_ALWAYS and disabled with EDT_ACT_NEVER passed to the enable argument. 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.

This routine is intended to work with edt_write(). It will not work well ring buffers since sequential dma operations are pipelined in hardware in the EDT dma engine.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

enable EDT_ACT_ALWAYS to enable, EDT_ACT_NEVER to disable.

reg_desc Register access description code.

set Register bits to be set.

clear Register bits to be cleared.

setclear Register value to be toggled up then down.

clearset Register value to be toggled down then up.

Example

```
edt_write_start_action(edt_p, EDT_ACT_ALWAYS, PCD_FUNCT, 0x8F, 0, 0x10, 0);
edt_write_start_action(edt_p, EDT_ACT_NEVER, dummy, dummy, dummy, dummy);
```

See also:

edt_write_end_action(), edt_read_start_action(), edt_read_end_action()

Definition at line 3336 of file libedt.c.

Register Access

Register access functions.

Functions

u_int edt_bar1_read (EdtDev *edt_p, u_int offset)

A convenience routine to access the EDT BAR1 registers.

---

void edt_bar1_write (EdtDev *edt_p, u_int offset, u_int val)

A convenience routine to access the EDT BAR1 registers.

---

uchar_t edt_intfc_read (EdtDev *edt_p, uint_t offset)

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

---

uint_t edt_intfc_read_32 (EdtDev *edt_p, uint_t offset)

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

---

u_short edt_intfc_read_short (EdtDev *edt_p, uint_t offset)

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

---

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

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

---

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

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

---

uint_t edt_reg_and (EdtDev *edt_p, uint_t desc, uint_t val)

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.

---

void edt_reg_clearset (EdtDev *edt_p, uint_t desc, uint_t val)

Toggles the bits specified in the mask argument off then on in a single ioctl call.

---

uint_t edt_reg_or (EdtDev *edt_p, uint_t desc, uint_t val)

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.

---

uint_t edt_reg_read (EdtDev *edt_p, uint_t desc)

Reads the specified register and returns its value.

---

void edt_reg_setclear (EdtDev *edt_p, uint_t desc, uint_t val)

Toggles the bits specified in the mask argument on then off in a single ioctl call.

---

void edt_reg_write (EdtDev *edt_p, uint_t desc, uint_t val)

Write the specified value to the specified register.

Function Documentation

*u_int edt_bar1_read (EdtDev * edt_p, u_int offset)*

A convenience routine to access the EDT BAR1 registers.

Passed the BAR1 byte address for a 32-bit word; note that the LS two bits of the address are ignored.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

offset integer byte offset into EDT BAR1 register memory, addressing a 32-bit value. Note that the LS two bits of the address are ignored.

Returns:

The value of the 32-bit register.

Example u_int reg24 = edt_bar1_read(edt_p, 0x24);

See also:

edt_bar1_write, edt_reg_read.

Definition at line 9369 of file libedt.c.

*void edt_bar1_write (EdtDev * edt_p, u_int offset, u_int data)*

A convenience routine to access the EDT BAR1 registers.

Passed the BAR1 byte address for a 32-bit word; note that the LS two bits of the address are ignored.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

offset integer byte offset into EDT BAR1 register memory, addressing a 32-bit value. Note that the LS two bits of the address are ignored.

data 32-bit value to set register with.

Example

```
u_int reg24 = 0xb01d_bee;
edt_bar1_write(edt_p, 0x24, reg24);
```

See also:
edt_bar1_read, edt_reg_write.

Definition at line 9398 of file libedt.c.

*uchar_t edt_intfc_read (EdtDev * edt_p, uint_t offset)*

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

The register descriptors used by edt_reg_read can also be used, since edt_intfc_read masks off the offset.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

offset integer offset into user interface XILINX, or edt_reg_read style register descriptor

Returns:
The value of the 8 bit register.

Example u_char func_reg = edt_intfc_read(edt_p, PCD_FUNC);

See also:
edt_intfc_write, edt_reg_read, edt_intfc_read_short

Definition at line 3427 of file libedt.c.

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

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

The register descriptors used be edt_reg_read can also be used, since edt_intfc_read_32 masks off the offset.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

offset integer offset into the user interface XILINX, or edt_reg_read style register descriptor.

Returns:
The value of the 32 bit register.

Definition at line 3548 of file libedt.c.

*u_short edt_intfc_read_short (EdtDev * edt_p, uint_t offset)*

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

The register descriptors used by edt_reg_read can also be used, since edt_intfc_read_short masks off the offset.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

offset integer offset into user interface XILINX, or edt_reg_read style register descriptor

Returns:
Value of the 16 bit register.

Example

```
int i;
puts("Directions for each channel of 16-channel card using
user interface xilinx 'ssdio.bit':");
u_short channel_direction_reg = edt_intfc_read_short(edt_p, SSD16_CHDIR);
for (i = 0; i < 16; ++i) {
    int dir = channel_dir_reg & (1 << i);
         printf("Channel %d configured for: ", i);
    if (dir == 0) {
        printf("input\n");
    } else if (dir == 1) {
        printf("output");
    }
}
```

See also:
edt_intfc_read, edt_reg_read

Definition at line 3497 of file libedt.c.

*void edt_intfc_write_32 (EdtDev * edt_p, uint_t offset, uint_t data)*

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

The register descriptors used by edt_reg_write can also be used, since edt_intfc_write_32 masks off the offset.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

offset integer offset into user interface XILINX, or edt_reg_write style register descriptor

data The 32 bit value to set the register to.

See also:

edt_intfc_read32, edt_reg_write

Definition at line 3572 of file libedt.c.

*void edt_intfc_write_short (EdtDev * edt_p, uint_t offset, u_short data)*

A convenience routine, partly for backward compatability, to access the user interface XILINX registers.

The register descriptors used by edt_reg_write() can also be used, since edt_intfc_write_short masks off the offset.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

offset,: integer offset into user interface XILINX, or edt_reg_write style register descriptor

data unsigned short integer value to set

Example

```
puts("Enabling all 16 DMA channels on PCDa with 'ssdio.bit'
loaded in user interface xilinx");
edt_intfc_write_short(edt_p, SSD16_CHEN, 0xffff);
```

See also:

edt_intfc_write, edt_reg_write

Definition at line 3528 of file libedt.c.

*uint_t edt_reg_and (EdtDev * edt_p, uint_t desc, uint_t mask)*

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.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

desc The name of the register to modify. Use the names provided in the register descriptions in Hardware Addendum for the card you are using (e.g. "PCI DV C-Link Hardware Addendum").

mask The value to AND with the register.

Returns:

The new value of the register

Definition at line 2961 of file libedt.c.

*void edt_reg_clearset (EdtDev * edt_p, uint_t desc, uint_t mask)*

Toggles the bits specified in the mask argument off then on in a single ioctl call.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

desc The name of the register to modify. Use the names provided in the register descriptions in Hardware Addendum for the card you are using (e.g. "PCI DV C-Link Hardware Addendum").

mask The value to XOR with the register.

Definition at line 2998 of file libedt.c.

*uint_t edt_reg_or (EdtDev * edt_p, uint_t desc, uint_t mask)*

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.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

desc The name of the register to modify. Use the names provided in the register descriptions in Hardware Addendum for the card you are using (e.g. "PCI DV C-Link Hardware Addendum").

mask The value to OR with the register.

Returns:
The new value of the register.

Definition at line 2920 of file libedt.c.

*uint_t edt_reg_read (EdtDev * edt_p, uint_t desc)*

Reads the specified register and returns its value.

Use this routine instead of using ioctls.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

desc The name of the register to read. Use the names provided in the register descriptions in Hardware Addendum for the card you are using (e.g. "PCI DV C-Link Hardware Addendum").

Returns:
The value of register.

Definition at line 2880 of file libedt.c.

*void edt_reg_setclear (EdtDev * edt_p, uint_t desc, uint_t mask)*

Toggles the bits specified in the mask argument on then off in a single ioctl call.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

desc The name of the register to modify. Use the names provided in the register descriptions in Hardware Addendum for the card you are using (e.g. "PCI DV C-Link Hardware Addendum").

mask The value to XOR with the register.

Definition at line 3024 of file libedt.c.

*void edt_reg_write (EdtDev * edt_p, uint_t desc, uint_t value)*

Write the specified value to the specified register.

Use this routine instead of using ioctls.

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

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

desc The name of the register to write. Use the names provided in the register descriptions in Hardware Addendum for the card you are using (e.g. "PCI DV C-Link Hardware Addendum").

value The desired value to write in register.

Definition at line 3055 of file libedt.c.

Utility

Utility functions.

Defines

#define edt_has_chanreg(edt_p) (ID_HAS_CHANREG(edt_p → devid))
#define edt_has_combined_fpga(edt_p) (ID_HAS_COMBINED_FPGA(edt_p → devid))
#define edt_has_irigb(edt_p) (ID_HAS_IRIGB(edt_p → devid))
#define edt_is_1553(edt_p) (ID_IS_1553(edt_p → devid))
#define edt_is_16bit_prom(edt_p) (ID_HAS_16BIT_PROM(edt_p → devid))
#define edt_is_16channel(edt_p) (ID_IS_16CHANNEL(edt_p → devid))
#define edt_is_1lane(edt_p) (ID_IS_1LANE(edt_p → devid))
#define edt_is_1or4channel(edt_p) (ID_IS_1OR4CHANNEL(edt_p → devid))
#define edt_is_2channel(edt_p) (ID_IS_2CHANNEL(edt_p → devid))
#define edt_is_32channel(edt_p) (ID_IS_32CHANNEL(edt_p → devid))
#define edt_is_3channel(edt_p) (ID_IS_3CHANNEL(edt_p → devid))
#define edt_is_4channel(edt_p) (ID_IS_4CHANNEL(edt_p → devid))
#define edt_is_4lane(edt_p) (ID_IS_4LANE(edt_p → devid))
#define edt_is_8lane(edt_p) (ID_IS_8LANE(edt_p → devid))
#define edt_is_dummy(edt_p) (ID_IS_DUMMY(edt_p → devid))
#define edt_is_dv_multichannel(edt_p) (edt_is_dvcl(edt_p) || edt_is_dvfox(edt_p) || edt_p → devid == PDVAERO_ID)
#define edt_is_dvcl(edt_p) (ID_IS_DVCL(edt_p → devid))
#define edt_is_dvcl2(edt_p) (ID_IS_CLSIM(edt_p → devid))
#define edt_is_dvcls(edt_p) (ID_IS_DVCLS(edt_p → devid))
#define edt_is_dvfox(edt_p) (ID_IS_DVFOX(edt_p → devid))
#define edt_is_fciusps(edt_p) (ID_IS_FCIUSPS(edt_p → devid))
#define edt_is_lcr_blade(edt_p) (ID_IS_LCRBLADE(edt_p → devid))
#define edt_is_micron_prom(edt_p) (ID_IS_MICRON_PROM(edt_p → devid))
#define edt_is_multichan(edt_p) (ID_IS_MULTICHAN(edt_p → devid))
#define edt_is_pcd(edt_p) (ID_IS_PCD(edt_p → devid))
#define edt_is_pcie_dvfox(edt_p) (ID_IS_PCIE_DVFOX(edt_p → devid))

#define edt_is_pdv(edt_p) (ID_IS_PDV(edt_p → devid))
#define edt_is_radio_blade(edt_p) (ID_IS_RADIOBLADE(edt_p → devid))
#define edt_is_simulator(edt_p) (ID_IS_CLSIM(edt_p → devid))
#define edt_is_unknown(edt_p) (ID_IS_UNKNOWN(edt_p → devid))

#define edt_pciload_info_na(edt_p) (ID_PCILOAD_INFO_NA(edt_p → devid))
#define edt_stores_macaddrs(edt_p) (ID_STORES_MACADDRS(edt_p → devid))
#define has_pcda_direction_bit(edt_p) (ID_HAS_PCD_DIR_BIT(edt_p → devid))

Functions

int edt_access (char *fname, int perm)

Determines file access, independent of operating system.

---

int edt_check_version (EdtDev *edt_p)

compares version strings between library and driver, returns 0 if they aren't the same

---

int edt_device_id (EdtDev *edt_p)

Gets the device ID of the specified device.

---

const char * edt_envvar_from_devstr (const char *devstr)
const char * edt_envvar_from_devtype (const int devtype)
u_int edt_errno (void)

Returns an operating system-dependent error number.

---

int edt_find_xpn (char *part_number, char *fpga)

Reads the default part number->fpga cross-reference file edt_parts.xpn in the current directory, and provides the FPGA if a match is found.

---

u_char edt_flipbits (u_char val)
int edt_get_bitname (EdtDev *edt_p, char *bitpath, int size)

Obtains the name of the currently loaded interface bitfile from the driver.

---

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

Obtains pathname to the currently loaded interface bitfile from the driver.

---

u_int edt_get_board_id (EdtDev *edt_p)

Gets the mezzanine id.

---

u_int edt_get_dma_info (EdtDev *edt_p, edt_dma_info *dmainfo)

Gets information about active dma.

---

int edt_get_driver_buildid (EdtDev *edt_p, char *build, int size)

Gets the full build ID of the EDT library.

---

int edt_get_driver_version (EdtDev *edt_p, char *versionstr, int size)

Gets the version of the EDT driver.

---

void edt_get_esn (EdtDev *edt_p, char *esn)

Retrieve the board's embedded information string from the PCI xilinx information header.

---

u_int edt_get_full_board_id (EdtDev *edt_p, int *extended_n, int *rev_id, u_int *extended_data)

Gets the mezzanine id including extended data.

---

char * edt_get_last_bitpath (EdtDev *edt_p)

int edt_get_library_buildid (EdtDev *edt_p, char *build, int size)

Gets the full build ID of the EDT library.

---

int edt_get_library_version (EdtDev *edt_p, char *versionstr, int size)

Gets the version (number and date) of the EDT library.

---

int edt_get_mezz_bitpath (EdtDev *edt_p, char *bitpath, int size)

Obtains pathname to the currently loaded mezzanine bitfile from the driver.

---

int edt_get_mezz_chan_bitpath (EdtDev *edt_p, char *bitpath, int size, int channel)

Obtains pathname to the currently loaded mezzanine bitfile from the driver.

---

u_int edt_get_mezz_id (EdtDev *edt_p)

void edt_get_osn (EdtDev *edt_p, char *osn)

Retrieve the board OEM's embedded information string from the PCI xilinx information header.

---

void edt_get_sns_sector (EdtDev *edt_p, char *esn, char *osn, int sector)

Retrieve the board's manufacturer and OEM embedded information strings strings from the PCI xilinx information header.

---

u_int edt_get_version_number ()

int edt_get_xref_info (const char *path, const char *pn, char *fpga, char *sn, char *mtype, char *moffs, char *mcount, char *desc, char *rsvd1, char *rsvd2)

Reads a part number->fpga cross-reference file and provides the fpga and base serial number if a match is found.

---

const char * edt_home_dir (EdtDev *edt_p)
char * edt_idstr (int id)

Converts the board ID returned by edt_device_id to a human readable form (original version, sans promcode).

---

char * edt_idstring (int id, int promcode)

Converts the board ID returned by edt_device_id to a human readable form (new version, with promcode).

---

uint_t edt_overflow (EdtDev *edt_p)
int edt_parse_devinfo (char *str, Edt_embinfo *ei)

Parse the board's embedded information string.

---

int edt_parse_esn (char *str, Edt_embinfo *ei)
int edt_parse_unit (const char *str, char *dev, const char *default_dev)

Parses an EDT device name string.

---

int edt_parse_unit_channel (const char *str, char *dev, const char *default_dev, int *channel)

parse -u argument returning the device and unit.

---

void edt_perror (char *str)

Formats and prints a system error.

---

int edt_set_bitpath (EdtDev *edt_p, const char *bitpath)

Sets pathname to the currently loaded user interface bitfile in the driver.

---

int edt_set_mezz_bitpath (EdtDev *edt_p, const char *bitpath)

Sets pathname to the currently loaded mezzanine bitfile in the driver.

---

int edt_set_mezz_chan_bitpath (EdtDev *edt_p, const char *bitpath, int channel)

Sets pathname to the currently loaded mezzanine bitfile in the driver.

---

u_int edt_set_mezz_id (EdtDev *edt_p)
int edt_system (const char *cmdstr)

Performs a UNIX-like system() call which passes the argument strings to a shell or command interpreter, then returns the exit status of the command or the shell so that errors can be detected.

---

char * edt_timestring (u_int *timep)

Function Documentation

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

Determines file access, independent of operating system.

This a convenience routine that maps to access() on Unix/Linux systems and _access() on Windows systems.

Parameters:

fname path name of file to check access permissions

perm permission flag(s) to test for. See documentation for access() (Unix/Linux) or _access() (Windows) for valid values.

Example

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

Returns:

0 on success, -1 on failure

Definition at line 452 of file edt_os_nt.c.

*int edt_device_id (EdtDev * edt_p)*

Gets the device ID of the specified device.

The board ID can be compared to specific board IDs listed in *edtreg.h*. To check if the specific board is part of a broader type, like PCD or PDV, macros such as edt_is_pcd and edt_is_pdv can be used.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

Definition at line 7235 of file libedt.c.

u_int edt_errno (void)

Returns an operating system-dependent error number.

Returns:

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

Example

```
if ((edt_p = edt_open("pcd", 0)) == NULL)
{
    edt_perror("edt_open failed");
    exit(edt_errno());
}
```

Definition at line 2853 of file libedt.c.

*int edt_find_xpn (char * part_number, char * fpga)*

Reads the default part number->fpga cross-reference file edt_parts.xpn in the current directory, and provides the FPGA if a match is found.

Equivalent to calling edt_find_get_xref_info with edt_parts.xpn as the filename. See /ref edt_find_xref_fpga for complete description.

Parameters:

fpga is a character array into which the fpga type will be stored (e.g. 'xc2s100e' will be returned for the part_number '01901933'). An array of 128 bytes will be more than enough for the foreseeable future.

Returns:

1 if found 8 or 10 digit match, 0 if not

Definition at line 8378 of file libedt.c.

*int edt_get_bitname (EdtDev * edt_p, char * bitpath, int size)*

Obtains the name of the currently loaded interface bitfile from the driver.

The program *bitload* sets this string in the driver when an interface bitfile is successfully loaded.

Parameters:

edt_p pointer to edt device structure returned by 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

Returns:

0 on success, -1 on failure.

See also:

edt_set_bitpath

Definition at line 7935 of file libedt.c.

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

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.

Parameters:

edt_p pointer to edt device structure returned by 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

Returns:

0 on success, -1 on failure.

See also:

edt_set_bitpath

Definition at line 7910 of file libedt.c.

*u_int edt_get_board_id (EdtDev * edt_p)*

Gets the mezzanine id.

Parameters:

edt_p

Returns:

mezzanine id

This function works on SS and GS boards to read the mezzanine board ids. It actually calls edt_get_full_board_id and ignores the extended data and rev_id returned from that function.

Definition at line 9117 of file libedt.c.

*u_int edt_get_dma_info (EdtDev * edt_p, edt_dma_info * dmainfo)*

Gets information about active dma.

Use this function to determine whether this or another open process has enabled DMA or image acquisition on any channel of a specific board (unit)

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

dmainfo pointer to struct of type edt_dma_info (defined in libedt.h), which has three fields containing hex values, with each bit in a field representing a channel that has been used, allocated or is currently active, as follows:

```
typedef struct
{
    uint_t    used_dma ;   // which channels have started dma within current open/close
    uint_t    alloc_dma ;  // which channels have has allocated > 1 ring buffer
    uint_t    active_dma ; // which channels have dma active right now
} edt_dma_info ;
```

Example

```
// this code checks whether this or some other process has done or is
// currently doing DMA on a given unit / channel, and prints out a warning
// if there is a possibility of a conflict based on the results

edt_dma_info tmpinfo ;
EdtDev *edt_p = edt_open_channel(EDT_INTERFACE, unit, channel);
u_int tmpmask = edt_get_dma_info(edt_p, &tmpinfo);

if (tmpinfo.active_dma & (1 << channel))
{
    printf("Warning: DMA is currently active on unit %d ch. %d.\n", unit, channel);
    printf("It is not safe to start another DMA on this unit/channel at this time.\n");
}
if (tmpinfo.used_dma & 1 << channel))
{
    printf("Warning: this or another process has already opened and done DMA on unit %d channel %d.
    printf("It may not be safe to start DMA on this unit/channel outside the currently opened proce
}
```

Returns:
mask of all of the above or'd together

See also:
setdebug.c Utility for example of use

Definition at line 8353 of file libedt.c.

int edt_get_driver_buildid (*EdtDev* * edt_p, *char* * build, *int* size)

Gets the full build ID of the EDT library.

The build ID string is the same format as that returned by edt_get_library_-buildid.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_-channel

build a string large enough to hold the build information (128 bytes is sufficient).

size the size, in bytes, of the user-allocated string

See also:
edt_get_library_buildid

Definition at line 7991 of file libedt.c.

*int edt_get_driver_version (EdtDev * edt_p, char * version, int size)*

Gets the version of the EDT driver.

The version string is the same format as that returned by edt_get_library_version.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel

version a string large enough to hold the version information (64 bytes is sufficient).

size the size, in bytes, of the user-allocated string

Definition at line 7967 of file libedt.c.

*void edt_get_esn (EdtDev * edt_p, char * esn)*

Retrieve the board's embedded information string from the PCI xilinx information header.

The EDT manufacturer's part numbers is embedded in an unused area of the Xilinx FPGA PROM, and is preserved across reloads (via pciload, hubload, etc.) unless options to overwrite are invoked in one of those utilities. This subroutine retrieves the EDT serial number portion of that information.

The data is an ASCII string, with the following colon-separated fields:

serial number:part number:clock speed:options:revision:interface xilinx:macaddrs:

(To see the information string, run *pciload* with no arguments.)

Note:
Information embedding was implemented in Sept. 2004; boards shipped before that time will yield a string with all NULLS unless the board's FPGA has since been updated with the embedded information.

Parameters:
edt_p pointer to edt device structure returned by edt_open

esn the EDT part number without dashes.

See also:
edt_get_sns, edt_get_osn, edt_parse_devinfo, edt_fmt_pn

Definition at line 857 of file edt_flash.c.

u_int edt_get_full_board_id (EdtDev * **edt_p***, int* * **extended_n***, int* * **rev_id***, u_int* * **extended_data***)*

Gets the mezzanine id including extended data.

Parameters:
edt_p

extended_n pointer to int to receive the number of extended data elements

rev_id pointer to int to fill in with the mezzanine rev_id

extended_data pointer to array to fill in with extended data elements

Returns:
mezzanine id

This function works on SS and GS boards to read the mezzanine board ids. If the id is an "extended" id, it reads the eeprom on the mezzanine including the extended data array.

The following values could be returned instead of the mezzanine id, if the mezzanine id couldn't be determined:

MEZZ_ERR_NO_BITFILE	Indicates that no UI bitfile is loaded, so the mezzanine id couldn't be determined.
MEZZ_ERR_BAD_BITSTREAM	Indicates an error while looking up the extended board info. Before EDT ticket #95 is fixed, this could also result when the ui bitfile is pciss4test and the mezz. board is 3X3G.
MEZZ_ERR_NO_REGISTER	Indicates that a bitfile has been loaded into the UI which doesn't support the extended board id register. All EDT UI bitfiles should support this, so contact EDT if this occurs.
MEZZ_UNKN_EXTBDID	Indicates that the board id is extended but the UI bitfile doesn't support this functionality. This is also unlikely - contact EDT if you see this.

If any of those values are returned, load a bitfile which supports the extended board id register, such as pciss1test, pciss4test, or pciss16test (depending on channels), or 3x3g.bit for the 3X3G board.

Definition at line 8953 of file libedt.c.

int edt_get_library_buildid (EdtDev * **edt_p**, *char* * **build**, *int* **size**)

Gets the full build ID of the EDT library.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

build a string large enough to hold the build information (128 bytes is sufficient).

size the size, in bytes, of the user-allocated string

See also:

edt_get_driver_buildid

Definition at line 8065 of file libedt.c.

*int edt_get_library_version (EdtDev * edt_p, char * version, int size)*

Gets the version (number and date) of the EDT library.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

version a string large enough to hold the version information (64 bytes is sufficient).

size the size, in bytes, of the user-allocated string

See also:

edt_get_driver_version

Definition at line 8015 of file libedt.c.

*int edt_get_mezz_bitpath (EdtDev * edt_p, char * bitpath, int size)*

Obtains pathname to the currently loaded mezzanine bitfile from the driver.

The edt_bitload sets this string in the driver when a mezzanine bitfile is successfully loaded.

Parameters:

edt_p pointer to edt device structure returned by 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

Returns:

0 on success, -1 on failure.

See also:

edt_get_bitpath

Definition at line 7856 of file libedt.c.

*int edt_get_mezz_chan_bitpath (EdtDev * edt_p, char * bitpath, int size, int channel)*

Obtains pathname to the currently loaded mezzanine bitfile from the driver.

The edt_bitload sets this string in the driver when a mezzanine bitfile is successfully loaded.

Parameters:

edt_p pointer to edt device structure returned by 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

Returns:

0 on success, -1 on failure.

See also:

edt_get_bitpath

Definition at line 7779 of file libedt.c.

*void edt_get_osn (EdtDev * edt_p, char * osn)*

Retrieve the board OEM's embedded information string from the PCI xilinx information header.

Some OEMs embed part number or other information about the board in an unused area of the Xilinx FPGA PROM. This information is preserved across reloads (via pciload, hubload, etc.) unless options to overwrite are invoked in one of those utilities. This subroutine retrieves the OEM serial number portion of that information.

Note:

Information embedding was implemented in Sept. 2004; boards shipped before that time will yield a string with all NULLS unless the board's FPGA has since been updated with the embedded information.

Parameters:

edt_p pointer to edt device structure returned by edt_open

osn the OEM's part number, if present.

See also:

edt_get_sns, edt_get_esn, edt_fmt_pn

Definition at line 879 of file edt_flash.c.

*void edt_get_sns_sector (EdtDev * edt_p, char * esn, char * osn, int sector)*

Retrieve the board's manufacturer and OEM embedded information strings strings from the PCI xilinx information header.

Certain information about the board, including manufacturer's part number, serial number, clock speed, Xilinx FPGA, and options, is embedded in an unused

area of the Xilinx FPGA PROM at the time of manufacture. This information is preserved across reloads (via pciload, hubload, etc.) unless options overwrite are invoked in the utility. This subroutine retrieves EDT and OEM (if present) information. The data is an ASCII string, with the following colon-separated fields:

serial number:part number:clock speed:options:revision:interface xilinx:

(To see the information string, run *pciload* with no arguments.)

Note:
Information embedding was implemented in Sept. 2004; boards shipped before that time will yield a string with all NULLS unless the board's FPGA has since been updated with the embedded information.

Parameters:
edt_p pointer to edt device structure returned by edt_open

esn the EDT part number without dashes.

osn the OEM's part number, if present.

See also:
edt_get_esn, edt_get_osn, edt_parse_devinfo, edt_fmt_pn

Definition at line 1044 of file edt_flash.c.

int edt_get_xref_info (const char * **path***, const char* * **pn***, char* * **fpga***, char* * **sn***, char* * **mtype***, char* * **moffs***, char* * **mcount***, char* * **desc***, char* * **rsvd1***, char* * **rsvd2***)*

Reads a part number->fpga cross-reference file and provides the fpga and base serial number if a match is found.

Opens the file specified in the *path* argument (e.g. *edt_parts.xpn*) and compares the entries with the provided part number. If a match is found, it will be copied to the *fpga* argument. Will also copy a serial number if found. Format of the file is ASCII text, one line per part number, as follows:

part_number fpga serial description (serial is optional and not present in earlier files)

Anything after the third item is ignored, and can be blank but should typically be the description (name of the device). Since files originally had only two fields and no serial number, an attempt is made to determine if the 3rd field looks like a serial # and copies that if so, otherwise sets the first character null.

Parameters:
path const character array containing path of the xref fpga file (typ. edt_parts.xpn).

part_number character array in which to store the part number, should be 8 or 10 digits. The last 2 digits of 10 digit part no. are the rev no. If a match with a 10-digit number is found, returns with the info from that one. If no 10-digit match is found but an 8-digit is found, returns with that info. That way we can have some numbers return a match regardless of rev, and others that cover a specific rev that takes precedence.

fpga a character array (64-bytes is sufficient) into which the fpga will be stored (e.g. xc2s100e' will be returned for the part_number '01901933'). If NULL this parameter will be ignored.

serial a character array into which the base serial number will be stored. An array of 64 bytes is sufficient, or NULL (ignored).

mac_type,: a character array (8 bytes is sufficient) into which the mac address board type (as a character string) will be stored. If NULL this parameter will be ignored.

mac_offset,: a character array (8 bytes is sufficient) into which the mac address offset will be stored. If NULL this parameter will be ignored.

nmacs,: a character array (8 bytes is sufficient) into which the number of mac addresses (as a character string) for this board type will be stored. If NULL this parameter will be ignored.

rsvd1,: reserved

rsvd2,: reserved

Returns:
number of parameters successfully assigned, or 0 if none.

Definition at line 8416 of file libedt.c.

char edt_idstr (int* id*)*

Converts the board ID returned by edt_device_id to a human readable form (original version, sans promcode).

For new 'a' boards that used the same devid as older versions (i.e. PCIe8 DV C-Link, PCIe4 DVa c-link, PCIe8 DV CLS) this subroutine will return without the 'a' suffix; therefore this subroutine should no longer be called directly; instead use edt_idstring() to make sure those boards get properly IDd.

Parameters:
id the board's hardware ID

Returns:
The id string of this board, with no check to see if it's an 'a board' (e.g. "pcie4 dv c-link")

Definition at line 7355 of file libedt.c.

*char** edt_idstring (int id, int promcode)

Converts the board ID returned by edt_device_id to a human readable form (new version, with promcode).

Supersedes edt_idstr which didn't take promcode now needed with new 'a' boards, some of which are detected via combination of ID and PROM code.

Parameters:

id the board's hardware ID

promcode the board's prom code, as defined in libedt.h

Returns:

The id string of this board, e.g. "pcie4 dva c-link"

Definition at line 7332 of file libedt.c.

*int edt_parse_devinfo (char * str, Edt_embinfo * ei)*

Parse the board's embedded information string.

During manufacturing programming, EDT embeds selected information is embedded into an unused area of the FPGA PROM. This information is preserved across reloads (via pciload, hubload, etc.) unless options to overwrite are invoked in one of those utilities. This subroutine takes as an argument the full information string, as retrieved from edt_get_esn, edt_get_osn or edt_get_sns, into the fields indicated by the Edt_embinfo structure.

(To see the information string, run *pciload* with no arguments.)

Note:

Information embedding was implemented in Sept. 2004; boards shipped before that time will yield a string with all NULLS unless the board's FPGA has since been updated with the embedded information.

Parameters:

str embedded inforamtion string, with information from one of the serial number retrieval subroutines (edt_get_esn, etc.)

ei Edt_embinfo structure into which the the parsed information will be put

See also:

edt_readinfo, edt_get_esn, edt_fmt_pn

Returns:

0 on success, -1 on error (input string not valid or too long)

Definition at line 3014 of file edt_flash.c.

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

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 and returns the device and unit number separately.

Parameters:

str device name string from command line. Should be either a unit number ("0" - "8") or device/unit concantenation ("pcd0," "pcd1," etc.)

dev array to hold the device device string; filled in by ths routine.

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."

Returns:

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

See also:

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

Definition at line 6279 of file libedt.c.

*int edt_parse_unit_channel (const char * instr, char * dev, const char * default_dev, int * channel_ptr)*

parse -u argument returning the device and unit.

Returns:

unit or -1 on failure (as well as device in dev, and channel in channel_ptr).

Parameters:

instr The input string. The argument of the -u option (like "0" or "pcd0" for example).

dev An array large enough to hold the device name, which is set by this function.

default_dev The default device to copy to dev if instr doesn't specify device. If NULL, EDT_INTERFACE will be used (which is "pcd" for pcd boards, "pdv" for dv boards, etc.).

channel_ptr The channel specified in instr, or 0 (set by this function). If channel_ptr is NULL or -1, it is ignored and unchanged.

Definition at line 6140 of file libedt.c.

*void edt_perror (char * **errstr**)*

Formats and prints a system error.

Parameters:
errstr Error string to include in printed error output.

See also:
edt_errno for an example

Definition at line 2815 of file libedt.c.

*int edt_set_bitpath (EdtDev * **edt_p**, const char * **bitpath**)*

Sets pathname to the currently loaded user interface bitfile in the driver.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel
bitpath address of a character buffer of at most 128 bytes

Returns:
0 on success, -1 on failure.

See also:
edt_get_bitpath, edt_set_mezz_bitpath

Definition at line 7879 of file libedt.c.

*int edt_set_mezz_bitpath (EdtDev * **edt_p**, const char * **bitpath**)*

Sets pathname to the currently loaded mezzanine bitfile in the driver.

Parameters:
edt_p pointer to edt device structure returned by edt_open or edt_open_channel
bitpath address of a character buffer of at most 128 bytes

Returns:
0 on success, -1 on failure.

See also:
edt_get_mezz_bitpath, edt_set_bitpath

Definition at line 7821 of file libedt.c.

*int edt_set_mezz_chan_bitpath (EdtDev * edt_p, const char * bitpath, int channel)*

Sets pathname to the currently loaded mezzanine bitfile in the driver.

Parameters:

edt_p pointer to edt device structure returned by edt_open or edt_open_channel

bitpath address of a character buffer of at most 128 bytes

channel which of two channels (0 or 1) this refers to

Returns:

0 on success, -1 on failure.

See also:

edt_get_mezz_bitpath, edt_set_bitpath

Definition at line 7719 of file libedt.c.

*int edt_system (const char * cmdstr)*

Performs a UNIX-like system() call which passes the argument strings to a shell or command interpreter, then returns the exit status of the command or the shell so that errors can be detected.

In WINDOWS spawnl() must be used instead of system() for this purpose.

Definition at line 7543 of file libedt.c.

EDT Digital Imaging Library

The PDV digital imaging library (pdvlib) provides a C language interface to the PDV device driver, including routines for image capture, save, and device control.

The library is designed for use with all EDT Digital Imaging boards, including the VisionLink series, PCIe and PCI DV, DVa and DV series Camera Link and legacy AIA interfaces, and PMC and Compact PCI variants. The library also has components to support the Camera Link simulator boards including the PCI DV CLS, PCIe8 DV CLS and PCIe8 DVa CLS.

The PDV library sits on top of the lower-level EDT DMA Library (edtlib). Library functions from both libraries operate on the same device handle, and routines from both libraries can be used in the same application. However pdvlib (`pdv_`) subroutines are designed to handle the extra bookeeping, error-recovery, triggering and timing functionality that is present on EDT Digital Imaging boards. Therefore direct calls to edtlib (`edt_`) subroutines should only be made when they provide functionality that is not present in an equivalent or similar pdvlib call. Most notable are the DMA image capture subroutines – pdvlib DMA should always be used (e.g. pdv_multibuf, pdv_start_images, pdv_wait_images), rather than calling the lower-level edtlib DMA subroutines directly (e.g. edt_configure_ring_buffers, edt_start_buffers, edt_wait_for_buffers.) However this restriction does not apply to the EDT Message Handler Library.

`Complete EDT API reference in PDF format`

Note:
Applications that access EDT boards must be linked with appropriate library (32 or 64-bit) for the platform in use. Applications linked with 32-bit EDT libraries will not run correctly on 64-bit systems, or vice-versa.

All routines access a specific device whose handle is created and returned by the pdv_open or pdv_open_channel routine. PDV applications typically include the following elements:

1. The preprocessor statement:

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

2. A call to pdv_open or pdv_open_channel, such as:

```
PdvDev *pdv_p = pdv_open_channel(EDT_INTERFACE, 0, 0);
```

   (EDT_INTERFACE is defined as `"pdv"` in edtdef.h.)

3. Device control or status calls, such as pdv_get_height, as in:

```
int height = pdv_get_height(pdv_p);
```

4. Ring buffer initialization code, such as:

```
pdv_multibuf(pdv_p, 4) ;
```

5. Data acquisition calls, such as pdv_image (which acquires an image and returns a pointer to it), as in:

```
unsigned char *image = pdv_image(pdv_p) ;
```

6. A check for timeouts (to flag a problem in the case of an unplugged camera, misconfiguration, or other reason for data loss), as in:

```
int t = pdv_timeouts() ;
```

   followed by appropriate action if new timeouts are detected, such as error output & timeout recovery code per simple_take.c

7. A call to pdv_close to close the device before ending the program, as in:

```
pdv_close(pdv_p) ;
```

8. Appropriate settings in your makefile or C workspace to compile and link the library files.
9. On Linux systems, the *-lpdv* and *-ledt* option to the compiler, to link the library file *libpdv.so* with your program.
10. On Linux systems, the *-L* and *-R* options to specify where to find the dynamic libraries at link- and runtime respectively. (See the makefile provided for examples.)
11. On Linux systems, the *-mt* option to the compiler (because the library uses multithreading), and the *-lm* option to the compiler (because it uses the math library).

To compile the library as a shared (.so) library on Linux, the following steps are necessary:

1. run

```
make clean
```

2. add

```
-fPIC
```

to the ALL_CFLAGS macro in the makefile

3. switch the library macro in the makefile:

```
LIBRARY=$(PDVLIB)
```

See the *makefile* and example programs provided in the install directory for examples of compiling code using the digital imaging library routines. Windows packages include a Visual Studio (8) solution and project files in *install_dir*\projects.vs2008.

Suggested starting points for acquisition are the simple_take.c, simplest_take.c and other **simple_*.c** example programs. For serial communication, see serial_cmd.c, a command line serial utility. Other **simple_*.c** example programs are provided to show specialized functionality.

The PdvDev device status structure is defined in the file libpdv.h. It includes the PdvDependent substructure, and other structure elements that describe the state of the board and camera, as initialized by the current camera configuration file (see the Camera Configuration Guide, at www.edt.com/manuals/PDV/camconfig.pdf) or modified by any subsequent API setup calls. These structure elements include values for things such as the current pixel re-order or color interpolation method, size and depth of the image, number and size of currently allocated buffers. To ensure compatibility with future versions of the library, programmers should always use the library calls for getting / setting any library values, and refrain from referencing the structure elements directly. Additionally, anything that can be queried via the subroutine calls such as currently set image width, height and depth should be done via subroutine calls rather than hard-coding specific values.

The PDV library source files are included in the installation. Most but not all routines are documented here. Undocumented routines include internals, custom, special purpose and obsoletes. Feel free, however, to look through the library source code and use (with caution) any routines that are appropriate. Email tech@edt.com if you have questions about specific routines.

Note:
When acquiring images in multithreaded applications, all routines that deal with starting, waiting for, or aborting images or buffers should be in the same thread.

Routines are divided into the following modules. You can use the Search button at the top of this page (in the HTML version of this doc) to search for specific library routines.

Modules

Startup / Shutdown

To open and close the EDT digital imaging device.

Settings

Get and set EDT interface board (register) values as well as device driver and camera settings.

Initialization

Read configuration files and initialize the board and camera.

Acquisition

Image acquisition subroutines.

Communications/Control

Serial communications and camera control subroutines.

Utility

Various utility subroutines.

Debug

Get and set flags that determine debug output from the library.

Startup / Shutdown

To open and close the EDT digital imaging device.

pdv_open and pdv_open_channel differ only in the *channel* argument. Since many applications are written for single channel boards (for example, the VisionLink F1) pdv_open will often suffice for opening a handle to the device. However it is just as easy to use pdv_open_channel with zero-assigned variable in the *channel* argument, providing for future possible expansion to multiple channel boards.

Functions

int pdv_close (PdvDev *pdv_p)

Closes the specified device and frees the device struct and image memory.

PdvDev * pdv_open (char *dev_name, int unit)

Opens channel 0 of an EDT Framegrabber for application access.

PdvDev * pdv_open_channel (const char *dev_name, int unit, int channel)

Opens an EDT Framegrabber channel for application access.

Function Documentation

*int pdv_close (PdvDev * pdv_p)*

Closes the specified device and frees the device struct and image memory.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

Returns:

0 if successful, -1 if unsuccessful

Definition at line 632 of file libpdv.c.

PdvDev pdv_open (char * dev_name, int unit)*

Opens channel 0 of an EDT Framegrabber for application access.

Opens the device, which is the first step in accessing the hardware. Allocates the memory for the device struct, as defined in libpdv.h (included through edtinc.h), and host memory required to store a captured image.

To open a specific channel on multi-channel device, see pdv_open_channel.

Parameters:

dev_name The name of the device, which for all EDT Digital Imaging boards is "pdv". If dev_name is NULL, "pdv" will be assumed. EDT_INTERFACE can also be used (recommended); it's defined as "pdv" in edtdef.h.

unit Unit number of the device (board). The first device is 0.

Returns:

A pointer to the PdvDev data structure, if successful. This data structure holds information about the device which is needed by library functions. User applications should avoid accessing structure elements directly. NULL is returned if unsuccessful.

See also:

pdv_open_channel

Definition at line 596 of file libpdv.c.

*PdvDev** *pdv_open_channel (const char* * **dev_name**, *int* **unit**, *int* **channel***)*

Opens an EDT Framegrabber channel for application access.

Opens the device, which is the first step in accessing the hardware. Allocates the memory for the device struct, as defined in libpdv.h (included through edtinc.h), and host memory required to store a captured image.

If you only want to use channel 0 on a multi-channel board, you can use pdv_open, but using pdv_open_channel with 0 in the **channel** argument is preferred.

pdv_open_channel provides for multiple cameras on separate channels, on boards that have multiple channels. Calling pdv_open_channel with using a specified board and channel returns a pointer to a software structure representing the connection to a specific camera – channel 0 for the camera on the connector closest to the PCI bus, channel 1 for the next connector up, and so on. Each call to pdv_open_channel with a unique channel number returns a discrete pointer, which is handled separately from any others, just as if each camera were connected to separate boards.

Example

```
// Example of opening and acquiring images from two cameras connected
// to separate channels 0 and 1 of a single VisionLink F4, PCIe8 DVa C-Link,
// or other multi-channel EDT Digital Imaging board

PdvDev *pdv_p0 = pdv_open_channel(EDT_INTERFACE, 0, 0);
PdvDev *pdv_p1 = pdv_open_channel(EDT_INTERFACE, 0, 1);

unsigned char *image_p0 = pdv_image(pdv_p0);
unsigned char *image_p1 = pdv_image(pdv_p1);
```

Note:
Acquiring data from multiple channels at the same time (or from multiple boards on the same bus) increases amount of data going across the bus. Unless the aggregate data is within the available bus bandiwdth, bus saturation (in the form of dropped data, broken images, overruns, or timeouts) is likely to occur. For more on bandwidth requirements, see EDT's `System Requirements web page`.

Parameters:

dev_name The name of the device, which for all EDT Digital Imaging boards is `"pdv"`. If dev_name is NULL, `"pdv"` will be assumed. EDT_INTERFACE can also be used (recommended); it's defined as `"pdv"` in edtdef.h.

unit Unit number of the device (board). The first device is 0.

channel The channel of the specified unit to open. The first channel is 0.

Returns:
A pointer to the PdvDev data structure, if successful. This data structure holds information about the device which is needed by library functions. User applications should avoid accessing structure elements directly. NULL is returned if unsuccessful.

See also:
pdv_open

Definition at line 473 of file libpdv.c.

Settings

Get and set EDT interface board (register) values as well as device driver and camera settings.

Most values get initialized from the config file, via the initcam program or the the camera configuration dialog (see pdv_initcam, initcam.c and the Camera Configuration Guide). In many cases the "get" routines are all that are used from an application, but sometimes it is useful for an application to make changes as well. These subroutines can be used to do both.

Functions

int pdv_auto_set_roi (PdvDev *pdv_p)

set ROI to camera width/height; adjust ROI width to be a multiple of 4, and enable ROI

---

char * pdv_camera_type (PdvDev *pdv_p)

Alias of pdv_get_cameratype.

---

int pdv_check_framesync (PdvDev *pdv_p, u_char *image_p, u_int *framecnt)

Checks for frame sync and frame count.

---

void pdv_cl_set_base_channels (PdvDev *pdv_p, int htaps, int vtaps)

Set the number of channels (taps) and horizontal and vertical alignment of the taps.

---

void pdv_enable_external_trigger (PdvDev *pdv_p, int flag)

Enables external triggering.

---

int pdv_enable_framesync (PdvDev *pdv_p, int mode)

Enables frame sync footer and frame out-of-synch detection.

---

int pdv_enable_lock (PdvDev *pdv_p, int flag)

Convenience routine to enable/disable shutter lock on/off on certain cameras.

---

int pdv_enable_roi (PdvDev *pdv_p, int flag)

Enables on-board region of interest.

---

int pdv_extra_headersize (PdvDev *pdv_p)

Return the header space allocated but not used for DMA.

---

int pdv_framesync_mode (PdvDev *pdv_p)

Returns the framesync mode.

---

int pdv_get_allocated_size (PdvDev *pdv_p)

Returns the allocated size of the image, including any header and pad for page alignment.

---

int pdv_get_blacklevel (PdvDev *pdv_p)

Gets the black level (offset) on the imaging device.

---

int pdv_get_bytes_per_image (PdvDev *pdv_p)

Gets the number of bytes per image, based on the set width, height, and depth.

---

int pdv_get_cam_height (PdvDev *pdv_p)

Returns the camera image height, in pixels, as set by the configuration file directive **height***, unaffected by changes made by setting a region of interest.*

---

int pdv_get_cam_width (PdvDev *pdv_p)

Returns the camera image width, in pixels, as set by the configuration file directive **width**.

---

char * pdv_get_camera_class (PdvDev *pdv_p)

Gets the class of the camera (usually the manufacturer name), as set by initcam *from the camera_config file* **camera_class** *directive.*

---

char * pdv_get_camera_info (PdvDev *pdv_p)

Gets the string set by the **camera_info** *configuration file directive.*

---

char * pdv_get_camera_model (PdvDev *pdv_p)

Gets the model of the camera, as set by initcam from the camera_config file **camera_model** *directive.*

---

char * pdv_get_cameratype (PdvDev *pdv_p)

Gets the type of the camera, as set by initcam *from the camera configuration file's camera description directives.*

---

int pdv_get_depth (PdvDev *pdv_p)

Gets the depth of the image (number of bits per pixel), as set in the configuration file for the camera in use.

---

int pdv_get_dmasize (PdvDev *pdv_p)

Returns the actual amount of image data for DMA.

---

int pdv_get_exposure (PdvDev *pdv_p)

Gets the exposure time on the digital imaging device.

---

int pdv_get_extdepth (PdvDev *pdv_p)

Gets the extended depth of the camera.

---

int pdv_get_firstpixel_counter (PdvDev *pdv_p)

Query state of the hardware first pixel counter register enable bit.

---

int pdv_get_frame_height (PdvDev *pdv_p)

Gets the camera image height.

---

int pdv_get_frame_period (PdvDev *pdv_p)

Get the frame period.

---

int pdv_get_gain (PdvDev *pdv_p)

Gets the gain on the device.

---

int pdv_get_header_dma (PdvDev *pdv_p)

Returns the current setting for flag which determines whether the header (or footer) size is to be added to the DMA size.

---

int pdv_get_header_offset (PdvDev *pdv_p)

Returns the byte offset of the header in the buffer.

HdrPosition pdv_get_header_position (PdvDev *pdv_p)

Returns the header or fotter position value.

---

int pdv_get_header_size (PdvDev *pdv_p)

Returns the currently defined header or footer size.

---

int pdv_get_header_within (PdvDev *pdv_p)

Tells if there is a header and it is within the data, and not extra data that gets added to the image DMA.

---

int pdv_get_height (PdvDev *pdv_p)

Gets the height of the image (number of lines), based on the camera in use.

---

int pdv_get_imagesize (PdvDev *pdv_p)

Returns the size of the image, absent any padding or header data.

---

int pdv_get_invert (PdvDev *pdv_p)

Get the state of the hardware invert register enable bit.

---

int pdv_get_max_gain (PdvDev *pdv_p)

Gets the maximum allowable gain value for this camera, as set by initcam *from the camera configuration file* **gain_max** *directive.*

---

int pdv_get_max_offset (PdvDev *pdv_p)

Gets the maximum allowable offset (black level) value for this camera, as set by initcam *from the camera configuration file* **offset_max** *directive.*

---

int pdv_get_max_shutter (PdvDev *pdv_p)

Gets the maximum allowable exposure value for this camera, as set by initcam *from the camera_config file* **shutter_speed_max** *directive.*

---

int pdv_get_min_gain (PdvDev *pdv_p)

Gets the minimum allowable gain value for this camera, as set by initcam *from the camera configuration file* **gain_min** *directive.*

---

int pdv_get_min_offset (PdvDev *pdv_p)

Gets the minimum allowable offset (black level) value for this camera, as set by initcam *from the camera configuration file* **offset_min** *directive.*

---

int pdv_get_min_shutter (PdvDev *pdv_p)

Gets the minimum allowable exposure value for this camera, as set by initcam *from the camera_config file* **shutter_speed_min** *directive.*

---

int pdv_get_pitch (PdvDev *pdv_p)

Gets the number of bytes per line (pitch).

---

int pdv_get_shutter_method (PdvDev *pdv_p, u_int *mcl)

Return shutter (expose) timing method and mode control (CC) state.

---

int pdv_get_width (PdvDev *pdv_p)

Gets the width of the image (number of pixels per line), based on the camera in use.

---

int pdv_image_size (PdvDev *pdv_p)

Returns the size of the image buffer in bytes, based on its width, height, and depth.

---

void pdv_invert (PdvDev *pdv_p, int val)

Tell the EDT framergrabber hardware to invert each pixel before transferring it to the host computer's memory.

---

void pdv_invert_fval_interrupt (PdvDev *pdv_p)

Set the Frame Valid interrupt to occur on the rising instead of falling edge of frame valid.

---

int pdv_picture_timeout (PdvDev *pdv_p, int value)

Sets the length of time to wait for data on acquisition before timing out.

---

int pdv_read_response (PdvDev *pdv_p, char *buf)

Read serial response, wait for timeout (or serial_term if specified), max is 2048 (arbitrary).

---

int pdv_set_binning (PdvDev *pdv_p, int xval, int yval)

Set binning on the camera to the specified values, and recalculate the values that will be returned by pdv_get_width, pdv_get_height, and pdv_get_imagesize.

---

int pdv_set_binning_dvc (PdvDev *pdv_p, int xval, int yval)

DVC 1312 binning.

---

int pdv_set_blacklevel (PdvDev *pdv_p, int value)

Sets the black level (offset) on the input device.

---

int pdv_set_cam_height (PdvDev *pdv_p, int value)

Sets placeholder for original full camera frame height, unaffected by ROI changes and usually only called by pdv_initcam.

---

int pdv_set_cam_width (PdvDev *pdv_p, int value)

Sets placeholder for original full camera frame width, unaffected by ROI changes and usually only called by pdv_initcam.

---

int pdv_set_cameratype (PdvDev *pdv_p, char *model)

Sets the camera's type (model) string in the dependent structure.

---

int pdv_set_depth (PdvDev *pdv_p, int value)

Deprecated – instead use the combined pdv_set_depth_extdepth_dpath.

---

int pdv_set_depth_extdepth (PdvDev *pdv_p, int depth, int extdepth)

Deprecated – instead use the combined pdv_set_depth_extdepth_dpath.

---

int pdv_set_depth_extdepth_dpath (PdvDev *pdv_p, int depth, int extdepth, u_int dpath)

Sets the bit depth, extended depth, and camera link data path.

---

int pdv_set_exposure (PdvDev *pdv_p, int value)

Sets the exposure time, using the method defined by the directives in the camera configuration file, if set.

---

int pdv_set_exposure_basler202k (PdvDev *pdv_p, int value)

set exposure (,gain, blacklevel) on basler A202K – ref BASLER A202K Camera Manual Document ID number DA044003.

---

int pdv_set_exposure_duncan_ch (PdvDev *pdv_p, int value, int ch)

Set exposure for Redlake (formerly Duncantech) DT and MS series cameras.

---

int pdv_set_exposure_mcl (PdvDev *pdv_p, int value)

Set the exposure when in pulse-width mode (also known as level trigger mode).

---

int pdv_set_extdepth (PdvDev *pdv_p, int value)

Deprecated – instead use the combined pdv_set_depth_extdepth_dpath.

---

void pdv_set_firstpixel_counter (PdvDev *pdv_p, int ena)

Enable hardware overwrite of first two bytes of the frame with a counter.

---

int pdv_set_frame_period (PdvDev *pdv_p, int period, int method)

Set the frame period counter and enable/disable frame timing.

---

void pdv_set_full_bayer_parameters (int nSourceDepth, double scale[3], double gamma, int nBlackOffset, int bRedRowFirst, int bGreenPixelFirst, int quality, int bias, int gradientcolor)

Sets the full bayer parameters for images for PCI DV library decoding of bayer formatted color image data.

---

int pdv_set_gain (PdvDev *pdv_p, int value)

Sets the gain on the input device.

---

int pdv_set_gain_duncan_ch (PdvDev *pdv_p, int value, int ch)

Set gain for Redlake (formerly Duncantech) DT and MS series cameras.

---

void pdv_set_header_dma (PdvDev *pdv_p, int header_dma)

Sets the boolean value for whether the image header is included in the DMA from the camera.

---

void pdv_set_header_offset (PdvDev *pdv_p, int header_offset)

Sets the byte offset of the header data in the allocated buffer.

---

void pdv_set_header_position (PdvDev *pdv_p, HdrPosition header_position)

Sets the header (or footer) position.

---

void pdv_set_header_size (PdvDev *pdv_p, int header_size)

Sets the header (or footer) size, in bytes, for the device.

---

pdv_set_header_type (PdvDev *pdv_p, int header_type, int irig_slave, int irig_offset, int irig_raw)

Sets the header (or footer) type.

---

int pdv_set_height (PdvDev *pdv_p, int value)

Sets height and reallocates buffers accordingly.

---

int pdv_set_roi (PdvDev *pdv_p, int hskip, int hactv, int vskip, int vactv)

Sets a rectangular region of interest, supporting cropping.

---

int pdv_set_shutter_method (PdvDev *pdv_p, int method, unsigned int mcl)

Set the device's exposure method and CC line state.

---

int pdv_set_width (PdvDev *pdv_p, int value)

Sets width and reallocates buffers accordingly.

---

int pdv_setsize (PdvDev *pdv_p, int width, int height)

Sets the width and height of the image.

---

int pdv_shutter_method (PdvDev *pdv_p)

Return shutter (expose) timing method.

Function Documentation

*int pdv_auto_set_roi (PdvDev * pdv_p)*

set ROI to camera width/height; adjust ROI width to be a multiple of 4, and enable ROI

mainly for use starting up with PCI DV C-Link which we want to use ROI in by default. But can be used for other stuff.

Definition at line 7994 of file libpdv.c.

char pdv_camera_type (PdvDev * pdv_p)*

Alias of pdv_get_cameratype.

This is the same as pdv_get_cameratype (but diff name) and exists for backward compatability.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
the camera type

See also:
pdv_get_cameratype

Definition at line 1725 of file libpdv.c.

*int pdv_check_framesync (PdvDev * pdv_p, u_char * image_p, u_int * framecnt)*

Checks for frame sync and frame count.

Framesync is hardware-enabled frame tagging via extra footer data on every frame. With framesync enabled, there are 16 bytes of extra footer data added to the frame DMA, with a magic number and frame count. If the magic number is not correct, framesync will return an error, allowing the calling function to handle the error. Typically this means stopping any continuous capture loop, resetting the DMA via pdv_timeout_restart, and re-starting continuous capture Or aborting altogether if repeated failures are detected (e.g. misconfiguration, cable unplugged, hardware failure.) The framecount argument allows users to ensure all frames are captured. It is not unusual for frames to be skipped but remain in sync; for example if blanking is very short between frames, or if the OS takes an extra long snooze to go do something else. Subroutine will return -1 if framesync is unsupported or not enabled, 0 if successful, or 1 if an out of sync condition is detected. If return code is 0, framecount will be updated with the current frame count, otherwise framecount will be 0.

Framesync functionality is available in PCIe Camera Link framegrabbers except the PCIe4 DV C-Link. This subroutine will return -1 if the device does not support this feature.

See also:
pdv_enable_framesync, pdv_framesync_mode;

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

image_p pointer to previously acquired image (via e.g. pdv_wait_image) for which you want the framesync to be checked.

framecnt pointer to location to put frame counter from this frame.

Returns:
result code (see description)

Definition at line 6080 of file libpdv.c.

*void pdv_cl_set_base_channels (PdvDev * pdv_p, int htaps, int vtaps)*

Set the number of channels (taps) and horizontal and vertical alignment of the taps.

Will set the number of Camera Link taps (channels) in the hardware by setting the left nibble of the PDV_CL_DATA_PATH register, and the htaps and vtaps PdvDev->dd_p structure elements.

For single-tap modes, htaps and vtaps should both be 1. For dual or 4-tap modes, most cameras output the data horizontally so htaps would be 2 or 4, and vtaps would remain 1. For RGB cameras (except bayer), htaps is usually 3 and vtaps 1.

Typically these are set via initcam or pdv_initcam; look at the various config files' htaps and vtaps directives. If a camera's output tap configuration is changed after after initialization, (usually via a serial command) this command can be used to update the framegrabber's registers to match.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_ope

htaps number of horizontal taps

htaps number of vertical taps

Returns:

void

See also:

pdv_set_depth_extdepth_dpath, **hskip**, **vskip** and **CL_DATA_PATH_NORM** directives in the Camera Configuration Guide

Definition at line 7931 of file libpdv.c.

*void pdv_enable_external_trigger (PdvDev * pdv_p, int flag)*

Enables external triggering.

One of several methods for external triggering. Calling this subroutine will enable the board's external trigger logic. When enabled via this subroutine, the hardware will queue any acquisition request made via pdv_start_image or similar subroutine, but will not service the request (that is, trigger the camera) until it sees a transition on the external trigger line coming in to the optical trigger pins (TTL level) on the board. If the camera is in freerun mode this of course won't have any effect.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

flag one of –

0 = turn off trigger

1 = turn on photo trigger

2 = turn on field ID trigger (through camera or cable). Does not apply to PCI C-Link.

Returns:
void

Definition at line 9742 of file libpdv.c.

*int pdv_enable_framesync (PdvDev * pdv_p, int mode)*

Enables frame sync footer and frame out-of-synch detection.

With framesync enabled, extra footer data is added to the frame DMA, enabling you to check for an out-of-synch condition using pdv_check_framesync or pdv_timeouts, and respond accordingly. The mode argument should be one of:

PDV_FRAMESYNC_OFF: Framesync functionality disabled.

PDV_FRAMESYNC_ON: Framesync functionality enabled, call pdv_check_framesync to check for out-of-synch data on a given frame.

PDV_FRAMESYNC_EMULATE_TIMEOUT: Framesync functionality enabled, framesync errors will be reflected as timeouts (see pdv_timeouts).

Framesync functionality is available in PCIe Camera Link framegrabbers except the PCIe4 (no 'a') DV C-Link. No PCI devices support this feature.

See also:
pdv_framesync, pdv_framesync_mode, pdv_timeouts;

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

mode framesync mode (see above)

Returns:
0 on success, -1 if not supported by the device in use.

Definition at line 6120 of file libpdv.c.

*int pdv_enable_lock (PdvDev * pdv_p, int flag)*

Convenience routine to enable/disable shutter lock on/off on certain cameras.

Obsolete routine, if camera can lock the shutter (currently only a few old Kodak Megaplus cameras) then just do it with pdv_serial_command.

Definition at line 8733 of file libpdv.c.

*int pdv_enable_roi (PdvDev * pdv_p, int flag)*

Enables on-board region of interest.

The rectangular region of interest parameters are set using pdv_set_roi; this subroutine is used to enable/disballe that region. Also calls pdv_setsize so subsequent calls to pdv_get_width or pdv_get_height return the values after region of interest is applied. Also resizes and reallocates any buffers allocated as a result of calling pdv_multibuf. Returns an error if the region of interest values are out of range.

The initial state of the region of interest can be controlled with directives in the configuration file. Most config files provided by EDT have ROI enabled by default. See the Camera Configuration Guide for more information.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

flag nonzero to enable region of interest; 0 to disable it.

Returns:

0 on success, -1 on failure.

See also:

pdv_set_roi for an example.

Definition at line 8035 of file libpdv.c.

*int pdv_extra_headersize (PdvDev * pdv_p)*

Return the header space allocated but not used for DMA.

Typically set via the **header_dma** and **header_size** directives in the configuration file.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

See also:

pdv_get_header_dma, pdv_set_header_size

Definition at line 5957 of file libpdv.c.

*int pdv_framesync_mode (PdvDev * pdv_p)*

Returns the framesync mode.

Can be one of:

PDV_FRAMESYNC_OFF: Framesync functionality disabled.

PDV_FRAMESYNC_ON: Framesync functionality enabled.

PDV_FRAMESYNC_EMULATE_TIMEOUT Framesync functionality enabled, and framesync errors will be reflected as timeouts.

See also:
pdv_framesync, pdv_check_framesync;

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 if enabled, 0 if not enabled;

Definition at line 6169 of file libpdv.c.

*int pdv_get_allocated_size (PdvDev * pdv_p)*

Returns the allocated size of the image, including any header and pad for page alignment.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
allocated size, in bytes.

See also:
pdv_image_size, pdv_get_header_dma

Definition at line 910 of file libpdv.c.

*int pdv_get_blacklevel (PdvDev * pdv_p)*

Gets the black level (offset) on the imaging device.

Applies only to cameras for which extended control capabilities have been written into the libaray, such as the Kodak Megaplus i series.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Black level value

Definition at line 3515 of file libpdv.c.

*int pdv_get_bytes_per_image (PdvDev * pdv_p)*

Gets the number of bytes per image, based on the set width, height, and depth.

Functionally equivalent to pdv_get_imagesize.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Definition at line 681 of file libpdv.c.

*int pdv_get_cam_height (PdvDev * pdv_p)*

Returns the camera image height, in pixels, as set by the configuration file directive **height**, unaffected by changes made by setting a region of interest.

See pdv_set_roi for more information.

Not to be confused with pdv_get_height; this subroutine gets the pdv_p->dd_p->cam_height value which only exists as a place to record the camera's (presumably) full height, as set by the config file 'height' directive and unaffected by any subsequent region of interest or pdv_setsize changes. This subroutine is just here to give applications a way to remember what that is. Doesn't change the buffer sizes or region of interest – for that, use pdv_set_roi or pdv_setsize.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Image height in pixels.

See also:
pdv_get_height, pdv_get_imagesize, **width** directive in the Camera Configuration Guide.

Definition at line 1177 of file libpdv.c.

*int pdv_get_cam_width (PdvDev * pdv_p)*

Returns the camera image width, in pixels, as set by the configuration file directive **width**.

Not to be confused with pdv_get_width; this subroutine gets the pdv_p->dd_p->cam_width value which only exists as a place to record the camera's (presumably) full width, as set by the config file 'width' directive and unaffected by any subsequent region of interest or pdv_setsize changes. Generally only useful to provide a hint to applications that want to know the original camera size since the value returned doesn't necessarily reflect the actual size of the buffers, frame passed in as modified by padding, headers or region of interest.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Image width in pixels.

See also:
pdv_get_dmasize, pdv_image_size, **width** directive in the Camera Configuration Guide

Definition at line 761 of file libpdv.c.

char pdv_get_camera_class (PdvDev * pdv_p)*

Gets the class of the camera (usually the manufacturer name), as set by *initcam* from the camera_config file **camera_class** directive.

Note:
the camera class is for application/GUI information only, and is not used by the driver or library. It is provided for the convenience of applications; for example the PdvShow and other camera configuration dialogs get and display the camera class, model and info strings to help the user to choose a spcific configuration.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
String representing the camera class.

See also:
pdv_get_cameratype, **camera_class** directive in the Camera Configuration Guide

Definition at line 1670 of file libpdv.c.

char pdv_get_camera_info (PdvDev * pdv_p)*

Gets the string set by the **camera_info** configuration file directive.

see pdv_get_cameratype for more information on camera strings.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
String representing the camera info.

See also:
pdv_set_camera_info, **camera_info** directive in the Camera Configuration Guide

Definition at line 1706 of file libpdv.c.

char pdv_get_camera_model (PdvDev * pdv_p)*

Gets the model of the camera, as set by initcam from the camera_config file **camera_model** directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
String representing the camera model.

See also:
pdv_set_camera_model, **camera_model** directive in the Camera Configuration Guide

Definition at line 1688 of file libpdv.c.

char pdv_get_cameratype (PdvDev * pdv_p)*

Gets the type of the camera, as set by *initcam* from the camera configuration file's camera description directives.

This is a concatenation of **camera_class**, **camera_model**, and **camera_info**, directives.

Note:
the camera class, model and info are for application/GUI information only, and are not used in any other way by the driver or library. They are provided for the convenience of applications such as PdvShow which uses them to help the user choose a specific camera configuration in the camera setup dialog.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
String representing the camera type.

See also:
pdv_get_camera_class, pdv_get_camera_model, pdv_get_camera_info, **camera_class**, **camera_model**, **camera_info** directives in the Camera Configuration Guide

Definition at line 1645 of file libpdv.c.

*int pdv_get_depth (PdvDev * pdv_p)*

Gets the depth of the image (number of bits per pixel), as set in the configuration file for the camera in use.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Number of bits per pixel in the image.

See also:
pdv_set_depth, pdv_get_extdepth, **depth** directive in the Camera Configuration Guide.

Definition at line 1340 of file libpdv.c.

*int pdv_get_dmasize (PdvDev * pdv_p)*

Returns the actual amount of image data for DMA.

Normally DMA is the same as the size of the sensor output (width x height x depth in bytes), so for example a 1K x 1k 8 bits per pixel camera would be 1024x1024x1 = 1048576 bytes, and a 1K x 1k 10 bits per pixel camera would be 1024x1024x2 = 2097152. However it can be different in a number of cases:

If DMA header data is enabled (for IRIGB timestamp input for example), dmasize will be imagesize plus the size of the header

If the sensor is a bayer or other interpolated image with one of the interleave options enabled (via the method_interlace: BGGR_WORD directive in the config file for example), imagesize will be at least 3x dmasize.

If the data is packed (e.g. 10-bit 8-tap mode), dmasize will be the exact size of the data coming in in bits, but imagesize will be the unpacked data size

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
DMA size in bytes – that is, the actual number of bytes acquired plus any added DMA if header data WITHIN the data is specified – see pdv_get_header_position, pdv_extra_headersize

See also:
pdv_image_size

Definition at line 787 of file libpdv.c.

*int pdv_get_exposure (PdvDev * pdv_p)*

Gets the exposure time on the digital imaging device.

Applies only when using board-controlled shutter timing (with a few cameras) for which shutter timing methods have been programmed into the library. The valid range is camera-dependent. See **method_camera_shutter_timing** configuration directive for more information.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Exposure time, in milliseconds.

See also:
pdv_set_exposure, **method_camera_shutter_timing** directive in the Camera Configuration Guide

Definition at line 3437 of file libpdv.c.

*int pdv_get_extdepth (PdvDev * pdv_p)*

Gets the extended depth of the camera.

The extended depth is the number of valid bits per pixel that the camera outputs, as set by *initcam* from the configuration file **edtdepth** directive. Note that if **depth** is set differently than **extdepth**, the actual number of bits per pixel passed through by the EDT framegrabber board will be different. For example, if **extdepth** is 10 but **depth** is 8, the board will only pass one byte per pixel, even though the camera is outputting two bytes per pixel.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open.

Returns:
The extended depth (an integer).

See also:
pdv_get_depth, **extdepth** directive in the Camera Configuration Guide.

Definition at line 1368 of file libpdv.c.

*int pdv_get_firstpixel_counter (PdvDev * pdv_p)*

Query state of the hardware first pixel counter register enable bit.

See /ref pdv_set_firstpixel_counter for details on this feature.

Only available on PCIe8 DVa C-Link, Visionlink, and going forward.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
state of the enable bit for this feature: 1=enabled, 0=disabled

Definition at line 3623 of file libpdv.c.

*int pdv_get_frame_height (PdvDev * pdv_p)*

Gets the camera image height.

The camera image height is in pixels, as set by the configuration file directive **height**, and is unaffected by changes made by setting the region of interest. Typically the value is the same as that returned by pdv_get_height unless the frame_height directive is specified in the config file and is different than height. This may occur in some cases where special handling of image data by an application is used such as multiple frames per image.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
the camera image height in pixels

See also:
pdv_set_roi, pdv_debug, pdv_get_imagesize

Definition at line 1203 of file libpdv.c.

*int pdv_get_frame_period (PdvDev * pdv_p)*

Get the frame period.

Returns the frame period, for boards that support the frame delay / frame period functionality. Frame_period is typically initialized via the **frame_period** configuration file directive (which pretty much always goes along with the **method_frame_timing** directive). **frame_period** is an integer value that determines either the number of microseconds between the start of one frame and the next, or the continuous frame trigger interval, depending on the state of the **frame_timing**. A more complete description of frame interval and frame timing can be found in pdv_set_frame_period.

Parameters:
pdv_p device handle returned by pdv_open

Returns:
period the frame period (microsecond units)

See also:
pdv_set_frame_period, **frame_period** directive in the Camera Configuration Guide

Definition at line 9866 of file libpdv.c.

*int pdv_get_gain (PdvDev * pdv_p)*

Gets the gain on the device.

Applies only to cameras for which extended control capabilities have been written into the library, such as the Kodak Megaplus i series.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Gain value. The valid range is -128 to 128. The actual range is camera-dependent.

Definition at line 3497 of file libpdv.c.

*int pdv_get_header_dma (PdvDev * pdv_p)*

Returns the current setting for flag which determines whether the header (or footer) size is to be added to the DMA size.

This is true if the camera/device returns header information at the beginning or end of its transfer.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 true or 0 false.

Definition at line 5924 of file libpdv.c.

*int pdv_get_header_offset (PdvDev * pdv_p)*

Returns the byte offset of the header in the buffer.

The byte offset is determined by the header position value. If header_position is PDV_HEADER_BEFORE, the offset is 0; if header_position is PDV_HEADER_AFTER (i.e. not really a header but a footer), the offset is the image size. If header_position is PDV_HEADER_WITHIN, the header offset can be set using the **header_offset** directive in the camera_configuration file, or by calling pdv_set_header_offset.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
A byte offset from the beginning of the buffer.

See also:
pdv_get_header_position, pdv_set_header_offset

Definition at line 5893 of file libpdv.c.

*HdrPosition pdv_get_header_position (PdvDev * pdv_p)*

Returns the header or fotter position value.

The header position value can by on of the following HdrPosition enumerated values:

HeaderNone

HeaderBefore

HeaderBegin

HeaderMiddle

HeaderEnd

HeaderAfter

HeaderSeparate

These values can be set in the configuration file with the **method_header_position** directive. The values in the configuration file should be the same as the definitions above without the leading **pdv_**.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Current header position.

See also:
pdv_get_header_offset, **header_offset** directive in the `Camera Configuration Guide`

Definition at line 5871 of file libpdv.c.

*int pdv_get_header_size (PdvDev * pdv_p)*

Returns the currently defined header or footer size.

This is usually set in the configuration file with the directive **header_size**. It can also be set by calling pdv_set_header_size.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Current header size.

See also:
pdv_set_header_size, **header_size** directive in the `Camera Configuration Guide`

Definition at line 5842 of file libpdv.c.

*int pdv_get_header_within (PdvDev * pdv_p)*

Tells if there is a header and it is within the data, and not extra data that gets added to the image DMA.

Returns 1 if header_position is any of the enumerated values HeaderBegin, HeaderMiddle, or HeaderEnd. Otherwise it returns 0.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 true or 0 false.

Definition at line 5939 of file libpdv.c.

*int pdv_get_height (PdvDev * pdv_p)*

Gets the height of the image (number of lines), based on the camera in use.

If the heigth has been changed by setting a region of interest, the new values are returned; use pdv_get_cam_height to get the unchanged height.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Height in pixels of images returned from an aquire.

See also:
pdv_get_cam_height, **height** directive in the Camera Configuration Guide.

Definition at line 1147 of file libpdv.c.

*int pdv_get_imagesize (PdvDev * pdv_p)*

Returns the size of the image, absent any padding or header data.

Since padding and header data are usually absent, the value returned from this is usually the same as that returned by pdv_image_size.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
image size, in bytes.

See also:
pdv_image_size

Definition at line 893 of file libpdv.c.

*int pdv_get_invert (PdvDev * pdv_p)*

Get the state of the hardware invert register enable bit.

See /ref pdv_invert for details on this feature.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
state of the enable bit for this feature: 1=enabled, 0=disabled

Definition at line 3577 of file libpdv.c.

*int pdv_get_max_gain (PdvDev * pdv_p)*

Gets the maximum allowable gain value for this camera, as set by *initcam* from the camera configuration file **gain_max** directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Maximum gain value.

See also:
gain directive in the Camera Configuration Guide

Definition at line 8686 of file libpdv.c.

*int pdv_get_max_offset (PdvDev * pdv_p)*

Gets the maximum allowable offset (black level) value for this camera, as set by *initcam* from the camera configuration file **offset_max** directive.

Parameters:
pdv_p device struct returned from pdv_open

Returns:
maximum offset value

See also:
offset directive in the Camera Configuration Guide

Definition at line 8720 of file libpdv.c.

*int pdv_get_max_shutter (PdvDev * pdv_p)*

Gets the maximum allowable exposure value for this camera, as set by *initcam* from the camera_config file **shutter_speed_max** directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Maximum exposure value.

Definition at line 8652 of file libpdv.c.

*int pdv_get_min_gain (PdvDev * pdv_p)*

Gets the minimum allowable gain value for this camera, as set by *initcam* from the camera configuration file **gain_min** directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Minimum gain value.

See also:
gain directive in the Camera Configuration Guide

Definition at line 8669 of file libpdv.c.

*int pdv_get_min_offset (PdvDev * pdv_p)*

Gets the minimum allowable offset (black level) value for this camera, as set by *initcam* from the camera configuration file **offset_min** directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Minimum offset value.

See also:
offset directive in the Camera Configuration Guide

Definition at line 8703 of file libpdv.c.

*int pdv_get_min_shutter (PdvDev * pdv_p)*

Gets the minimum allowable exposure value for this camera, as set by *initcam* from the camera_config file **shutter_speed_min** directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Minimum exposure value.

See also:
shutter_speed_min directive in the Camera Configuration Guide

Definition at line 8636 of file libpdv.c.

*int pdv_get_pitch (PdvDev * pdv_p)*

Gets the number of bytes per line (pitch).

Functionally equivalent to pdv_get_width.

Parameters:
pdv_p device struct returned by pdv_open

Returns:
width in pixels of images returned from an aquire.

See also:
pdv_get_width

Definition at line 724 of file libpdv.c.

*int pdv_get_shutter_method (PdvDev * pdv_p, u_int * mcl)*

Return shutter (expose) timing method and mode control (CC) state.

See pdv_set_shutter_method for an explanation of the return value (shutter method) and mcl parameter;

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

mcl mode control (CC line) state

Returns:

the shutter (expose) timing method

See also:

pdv_set_shutter_method

Definition at line 6190 of file libpdv.c.

*int pdv_get_width (PdvDev * pdv_p)*

Gets the width of the image (number of pixels per line), based on the camera in use.

If the width has been changed by setting a region of interest, the modified values are returned; use pdv_get_cam_width to get the unchanged width.

Parameters:

pdv_p device struct returned by pdv_open

Returns:

Width in pixels of images returned from an aquire.

Definition at line 704 of file libpdv.c.

*int pdv_image_size (PdvDev * pdv_p)*

Returns the size of the image buffer in bytes, based on its width, height, and depth.

Enabling a region of interest changes this value. The size returned includes allowance for buffer headers. To obtain the actual size of the image data without any optional header or other padding, see pdv_get_dmasize.

Parameters:

pdv_p device struct returned from pdv_open

Returns:

Total number of bytes in the image, including buffer header overhead.

See also:
pdv_set_roi

Definition at line 6875 of file libpdv.c.

*void pdv_invert (PdvDev * pdv_p, int val)*

Tell the EDT framergrabber hardware to invert each pixel before transferring it to the host computer's memory.

This is a hardware operation that is implemented in the board's firmware and has no impact on performance.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
val 1=invert, 0=normal

Returns:
void

Definition at line 3549 of file libpdv.c.

*void pdv_invert_fval_interrupt (PdvDev * pdv_p)*

Set the Frame Valid interrupt to occur on the rising instead of falling edge of frame valid.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 9785 of file libpdv.c.

*int pdv_picture_timeout (PdvDev * pdv_p, int value)*

Sets the length of time to wait for data on acquisition before timing out.

This function is only here for backwards compatibility. You should use pdv_set_timeout() instead.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
value the number of milliseconds to wait for timeout, or 0 to block waiting for data

Returns:
0 if successful, nonzero on failure.

Definition at line 1012 of file libpdv.c.

*int pdv_read_response (PdvDev * pdv_p, char * buf)*

Read serial response, wait for timeout (or serial_term if specified), max is 2048 (arbitrary).

This subroutine has limited usefulness. While it is convenient in that it combines the wait/read sequence, optimized command/response is usually better accomplished with separate pdv_serial_command / pdv_serial_wait / pdv_serial_read sequences

Returns:
number of characters read

See also:
pdv_serial_read, pdv_serial_wait

Definition at line 3469 of file libpdv.c.

*int pdv_set_binning (PdvDev * pdv_p, int xval, int yval)*

Set binning on the camera to the specified values, and recalculate the values that will be returned by pdv_get_width, pdv_get_height, and pdv_get_imagesize.

Only applicable to cameras for which binning logic has been implemented in the library – specifically DVC cameras that use the *BIN xval yval*, Atmel cameras that use *B*= val (where val= 0, 1 or 2), or in conjunction with the **serial_binning** camera configuration directive for any camera that uses an ASCII *CMD VALUE* pair to set binning.

This subroutine was an attempt to provide a way to set binning in a generic way, handling a few specific cameras via special code and others using an assumed serial format. As it turned out, the "assumed" format is not all that standard, therefore this subroutine is of limited usefulness.

If your camera is one that takes a single ASCII command / argument to set a binning mode, then this subroutine may still be handy since it can be a single-call method for setting the camera and the board in a given binning mode.

To use this method, simply set the **serial_binning** camera configuration directive to the command that sets binning. Then when called, this subroutine will send the command and reset the board's camera size.

If your camera does not fit any of the above formats (or if you would rather not depend on this flakey logic), simply use pdv_serial_command or pdv_serial_binary_command to send the command to put the camera into binned mode, then call pdv_setsize to reset the board to the new frame size.

If the PDV library does not know how to set binning on the camera in use, a -1 will be returned and the width/height/imagesize will remain unchanged.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

xval x binning value. Usually 1, 2, 4 or 8. Default is 1.

yval y binning value. Usually 1, 2, 4 or 8. Default is 1.

Returns:

0 on success, -1 on failure.

See also:

serial_binning directive in the Camera Configuration Guide

Definition at line 3346 of file libpdv.c.

*int pdv_set_binning_dvc (PdvDev * pdv_p, int xval, int yval)*

DVC 1312 binning.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

xval horizontal binning value

yval vertical binning value

Definition at line 9553 of file libpdv.c.

*int pdv_set_blacklevel (PdvDev * pdv_p, int value)*

Sets the black level (offset) on the input device.

Applies only to cameras for which extended control capabilities have been added to the library (see the source code), or that have a serial command protocol that has been configured using the **serial_offset** configuration directive. Unless you know that one of the above has been implemented for your camera, it is usually safest to just send the specific serial commands via pdv_serial_command or pdv_serial_write.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

value Black level value. The valid range is camera-dependent.

Returns:

0 on success, -1 on failure.

See also:

pdv_get_offset, **serial_offset** configuration file directive.

Definition at line 3168 of file libpdv.c.

*int pdv_set_cam_height (PdvDev * pdv_p, int value)*

Sets placeholder for original full camera frame height, unaffected by ROI changes and usually only called by pdv_initcam.

Not to be confused with pdv_set_height; this subroutine sets the pdv_p->dd_p->cam_height value, which only exists as a place to record the camera's (presumably) full height, normally set by the config file 'height' directive and unaffected by any subsequent region of interest or pdv_setsize changes. This subroutine is just here to give applications a way to change that value, though it normally only gets called by pdv_initcam. Doesn't change the buffer sizes or region of interest – for that, use pdv_set_roi or pdv_setsize.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

value height of the camera's sensor in pixels

Returns:

0 on success, -1 on failure.

Definition at line 1317 of file libpdv.c.

*int pdv_set_cam_width (PdvDev * pdv_p, int value)*

Sets placeholder for original full camera frame width, unaffected by ROI changes and usually only called by pdv_initcam.

Not to be confused with pdv_set_width; this subroutine sets the pdv_p->dd_p->cam_width value, which only exists as a place to record the camera's (presumably) full width, normally set by the config file 'width' directive and unaffected by any subsequent region of interest or pdv_setsize changes. Generally only useful to provide a hint to applications a way to change that value, though it normally only gets called by pdv_initcam. Doesn't change the buffer sizes or region of interest – for that, use pdv_set_roi or pdv_setsize.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

value width of the camera's sensor in pixels

Returns:

0 on success, -1 on failure.

Definition at line 868 of file libpdv.c.

*int pdv_set_cameratype (PdvDev * pdv_p, char * model)*

Sets the camera's type (model) string in the dependent structure.

typically the camera model is set via initcam using the **camera_model** configuration file directive. This subroutine is provided in case there is a need for an application program to modify the string.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

model camera model (31 characters max).

Returns:

0 on success, -1 on failure.

See also:

pdv_get_cameratype, **cameratype** directive in the Camera Configuration Guide

Definition at line 1616 of file libpdv.c.

*int pdv_set_depth (PdvDev * pdv_p, int value)*

Deprecated – instead use the combined pdv_set_depth_extdepth_dpath.

The bit depth is the number of valid bits per pixel that the board will transfer across the bus. Normally depth is initialized during *initcam* via the configuration file **depth** directive, and the only time this subroutine should be needed is if the depth changes, via a post-initialization command to the camera for example.

Note that if **depth** is set differently than **extdepth**, the actual number of bits per pixel passed through by the EDT framegrabber board will be different from that received from the camera. For example, if **extdepth** is 10 (matching a camera output of 10 bits) but **depth** is 8, the board will only pass one byte per pixel, even though the camera is outputting two bytes per pixel. There are also special cases including 24-bit depth / 8-bit extdepth (Bayer), and 10-bit depth / 80-bit extdepth (8-tap, 10-bit packed).

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

value the new depth value

Returns:

The extended depth (an integer).

See also:

pdv_set_depth_extdepth, pdv_set_depth_extdepth_dpath, pdv_get_depth, pdv_get_extdepth, **extdepth** directive in the Camera Configuration Guide

Definition at line 1528 of file libpdv.c.

*int pdv_set_depth_extdepth (PdvDev * pdv_p, int depth, int extdepth)*

Deprecated – instead use the combined pdv_set_depth_extdepth_dpath.

Sets the bit depth and extended depth. Depth is the number of valid bits per pixel that the board will transfer across the bus. Extended depth (extdepth) is usally the same but not always, for example if we want to pass only the upper 8 bits of data from a 12 bit camera, depth will be 8 and extdepth will 12. Bayer color cameras are another special case – for example a 24-bit RGB camera should have depth set to 24 and extdepth to 8.

Normally depth and extended depth are initialized during *initcam* via the configuration file **depth** and **extdepth** directives. Therefore, the only time this subroutine should be needed is if the depth changes, for example via a post-initialization command to the camera.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

depth the new depth value

extdepth the new extended depth value

Returns:

0 on success, -1 on failure

See also:

pdv_get_depth, pdv_get_extdepth, **depth**, **extdepth** directives in the Camera Configuration Guide

Definition at line 1400 of file libpdv.c.

*int pdv_set_depth_extdepth_dpath (PdvDev * pdv_p, int depth, int extdepth, u_int dpath)*

Sets the bit depth, extended depth, and camera link data path.

Depth is the number of valid bits per pixel that the board will transfer across the bus. Extended depth (extdepth) is usally the same but not always, for example if we want to pass only the upper 8 bits of data from a 12 bit camera, depth will be 8 and extdepth will 12. Bayer color cameras are another special case – for example a 24-bit RGB camera should have depth set to 24 and extdepth to 8.

This subroutine also allows you to set the camera link data path register for the specific number of taps and bits per pixel. Specific value (hex) is as follows:

Left (MS) nibble: number of taps minus 1

Right (LS) nibble: number of bits per pixel minus 1

For example for a 2-tap, 8-bit camera, dpath should be 0x17. The correct data path value can usually be inferred automatically from the depth. If you specify

a dpath value of 0, pdv_set_depth_extdepth_dpath will automatically set the register to the most likely value.

Normally depth, extended depth and dpath are initialized during *initcam* via the configuration file **depth** and **extdepth** and **CL_DATA_PATH_NORM** directives. Therefore, the only time this subroutine should be needed is if the depth changes, for example via a post-initialization command to the camera.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

depth the new depth value

extdepth the new extended depth value

dpath the new camera link data path value

Returns:

0 on success, -1 on failure

See also:

pdv_cl_set_base_channel, spdv_get_depth, pdv_get_extdepth, **depth**, **extdepth**, **CL_DATA_PATH_NORM** directives in the Camera Configuration Guide

Definition at line 1442 of file libpdv.c.

*int pdv_set_exposure (PdvDev * **pdv_p**, int **value**)*

Sets the exposure time, using the method defined by the directives in the camera configuration file, if set.

pdv_set_exposure will set the exposure (or not) on the camera depending on how the related directives are set in the camera configuration file. Specifically, the **method_camera_shutter_timing** directive (or pdv_set_shutter_method) defines whether timing is to be controlled via camera serial commands, or by the board via Camera Control (CC) lines.

If **method_camera_shutter_timing** is **AIA_MCL** or **AIA_MCL_100US** and something other than 0 is in the left nibble of **MODE_CNTL_NORM**, the board will use its internal shutter timer and send out an expose pulse on the specified CC line with a TRUE period of the number in milliseconds (AIA_MCL) or tenths of milliseconds (AIA_MCL_100US) specified by the **value** parameter. The valid range in either case is 0-25500.

If **method_camera_shutter_timing** is **AIA_SERIAL** (the default), and then this subroutine sends the appropriate serial commands based on the **method_serial_format** directive, which defines which serial format is to be used. The default format is SERIAL_ASCII, in which case the subroutine will set the exposure by sending the command specified by the **serial_exposure** directive, if

present. If **method_serial_format** is **SERIAL_ASCII** but there is no **serial_-exposure** directive, this subroutine is a no-op.

In the case of **method_serial_format: SERIAL_ASCII** or any other serial mode, the range is camera dependent. Other methods are available that are specific to specific cameras – see the Camera Configuration guide for details.

Note:
Using this subroutine for other than **AIA_MCL** or **AIA_100US** camera shutter timing modes (that is, any method that uses serial) is no longer recommended. Back in the AIA (pre-Camera Link) days, there was a manageable set of serial methods, so it made sense to have one subroutine that could control exposure time for all the available methods. But the sheer number of different schemes has outgrown this library's ability to keep up, so for any camera command sets other than those that use straight ASCII serial with an integer argument, it's more reliable to instead send any camera-specific serial commands using pdv_serial_command, pdv_serial_binary_command, or pdv_serial_write.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

value Exposure time. For AIA_MCL or AIA_MCL_100US, the valid range is 0-25500. For other methods, valid range and increments are camera-dependent.

Returns:
0 if successful, -1 if unsuccessful.

See also:
pdv_set_shutter_method, pdv_get_shutter_method, pdv_set_exposure_mcl Camera Configuration directives **MODE_CNTL_NORM**, **serial_-exposure** & **method_camera_shutter_timing**

Definition at line 1824 of file libpdv.c.

*int pdv_set_exposure_duncan_ch (PdvDev * pdv_p, int value, int ch)*

Set exposure for Redlake (formerly Duncantech) DT and MS series cameras.

ref. DuncanTech User Manual Doc # 9000-0001-05

Note:
Convenience routine, for Duncantech (Redlake) DT/MS series cameras only. Intended as a starting point for programmers wishing to use EDT serial commands with Duncantech cameras. These subroutines can be used as a template for controlling camera parameters beyond simple exposure and gain.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

value expsosure value

ch camera channel

See also:
pdv_send_duncan_frame, pdv_read_duncan_frame

Definition at line 2764 of file libpdv.c.

*int pdv_set_exposure_mcl (PdvDev * pdv_p, int value)*

Set the exposure when in pulse-width mode (also known as level trigger mode).

Sets data Path register decade bits as appropriate for value input. Called by pdv_set_exposure if dd_p->camera_shutter_timing is set to **AIA_MCL** or **AIA_MCL_100US** (typically set by config file directive **method_camera_shutter_timing: AIA_MCL**; (**MODE_CNTL_NORM: 10** should typically also be set). If AIA_MCL, units are milliseconds. If AIA_MCL_100US, units are in microseconds. Sets the actual exposure time to value + 1.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

value Exposure time, range 0-65535

Returns:
0 if successful, -1 if unsuccessful.

See also:
pdv_set_exposure, **MODE_CNTL_NORM** & **method_camera_shutter_timing** directive in the Camera Configuration Guide

Definition at line 2021 of file libpdv.c.

*int pdv_set_extdepth (PdvDev * pdv_p, int value)*

Deprecated – instead use the combined pdv_set_depth_extdepth_dpath.

Sets the bit depth coming from the camera. Normally only called by pdv_initcam; user applications should avoid calling this subroutine directly.

Extdepth must match the number of valid bits per pixel coming from the camera. Normally this is initialized during *initcam* via the configuration file **extdepth** directive. The only time this subroutine should be needed is if the camera's depth changes, via a post-initialization command sent to the camera for example.

Note that if **depth** is set differently than **extdepth**, the actual number of bits per pixel passed through by the EDT framegrabber board will be different. For example, if **extdepth** is 10 but **depth** is 8, the board will only pass one byte per pixel, even though the camera is outputting two bytes per pixel.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

value the extended depth, in bits per pixel

Returns:

The extended depth (an integer).

See also:

pdv_get_extdepth, pdv_set_depth, **extdepth** directive in the Camera Configuration Guide

Definition at line 1590 of file libpdv.c.

*void pdv_set_firstpixel_counter (PdvDev * pdv_p, int ena)*

Enable hardware overwrite of first two bytes of the frame with a counter.

Counter increments by one for every frame received by the framegrabber. Disabling this also resets the counter to zero, unless framesync mode is also enabled (see pdv_enable_framesync).

Only available on PCIe8 DVa C-Link, Visionlink, and going forward.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

val 1=enable, 0=disable

Returns:

void

Definition at line 3600 of file libpdv.c.

*int pdv_set_frame_period (PdvDev * pdv_p, int period, int method)*

Set the frame period counter and enable/disable frame timing.

Enables either continuous frame pulses at a specified interval, or extending the frame valid signal by the specified amount, to in- effect extend the amount of time after a frame comes in from the camera before the next trigger is issued. This can be used to hold off on issuing subsequent triggers for cameras that require an extra delay between triggers, or to set a specific trigger interval. Only applies when the camera is in triggered or pulse-width mode and the board is controlling the timing.

The camera config file directives **frame_period** and **method_frame_timing** (which pretty much always go together) are typically used to initialize these values at initcam time for cameras that need a fixed frame delay for reliable operation in a given mode (very rare). Frame timing functionality is disabled by default.

Note:
See the Triggering section in your EDT framegrabber's Users Guide, and also the `Camera Configuration Guide` for more on camera triggering methods.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

period frame period in microseconds-2, range 0-16777215

method one of:

0 – disable frame counter

PDV_FMRATE_ENABLE – continuous frame counter

PDV_FVAL_ADJUST – frame counter extends every frame valid by 'period' microseconds

Returns:
-1 on error, 0 on success

See also:
pdv_get_frame_period

Definition at line 9826 of file libpdv.c.

void *pdv_set_full_bayer_parameters (int* nSourceDepth*, double* scale*[3], double* gamma*, int* nBlackOffset*, int* bRedRowFirst*, int* bGreenPixelFirst*, int* quality*, int* bias*, int* gradientcolor*)*

Sets the full bayer parameters for images for PCI DV library decoding of bayer formatted color image data.

Bayer decoding by the library is typically enabled by setting the config file directive `method_interlace` to `BGGR` or `BGGR_WORD`; this subroutine can be used to manipulate the specific Bayer decoding parameters. Images captured with pdv_image, pdv_wait_images or othe PCI DV library acquisition routines (excepting `_raw` routines) will be preprocessed to RGB color before the image pointer is returned.

The `bRedRowFirst` and `bGreenPixelFirst` parameters are typically initialized by the `kbs_red_row_first` and `kbs_green_pixel_first` configuration file directives. Current values can be found in the `PdvDev` `dd_p->kbs_green_pixel_first` and `dd_p->kbs_dd_p->red_row_first` structure elements.

The most common operation for pdv_set_full_bayer_parameters is adjusting the white balance. To do so, the calling application should provide a method for acquiring an image of a white background, calcluate the average of all pixels in each of the R, G and B components, then set `scale`[0] (green) to 1.0, and adjust `scale`[1-2] (red/blue) such that red and blue will be scaled appropriately.

Click on the color wheel toolbar icon in PdvShow to see an example of such an implementation.

Note that the Bayer decoding functionality uses MMX instructions when run under the Windows environment, providing greater efficiency and more algorithm (quality) options. Only one algorithm is defined in the Linux/Unix implementation so the `quality` parameter will be ignored on those platforms.

Parameters:

nSourceDepth depth in bits of source (unfiltered) data

scale array of 3 values (R,G,B) for scaling (gain); default 1.0, 1.0, 1.0

gamma gamma value – default 1.0

nBlackOffset Black Offset (black level); 1 is default

bRedRowFirst 1 if red/green row is first on the sensor, 0 if blue/green is first

bGreenPixelFirst 1 if green pixel is first on sensor, 0 if red or blue

quality selects one of 3 Bayer decoding algorithms: 0=Bilinear, 1=Gradient, 2=Bias-corrected – MS Windows only. Note that in Linux/Unix, only Bilinear is implemented and this parameter is ignored

bias selects the bias for bias method Bayer algorithm; (MS Windows only)

gradientcolor selects the gradient for the gradient Bayer algorithm (MS Windows only)

See also:

method_interlace, **kbs_red_row_first**, **kbs_green_pixel_first** camera configuration directives – see the `Camera configuration guide`

Definition at line 237 of file pdv_bayer_filter.c.

*int pdv_set_gain (PdvDev * pdv_p, int value)*

Sets the gain on the input device.

Applies only to cameras for which extended control capabilities have been added to the library (see the source code), or that have a serial command protocol that has been configured using the **serial_gain** configuration directive. Unless you know that one of the above has been implemented for your camera, it is usually safest to just send the specific serial commands via pdv_serial_command or pdv_serial_write.

Example

```
pdv_set_gain(pdv_p, 0); // neutral gain
```

Returns:

0 on success, -1 on failure.

See also:
pdv_get_gain, **serial_gain** configuration file directive.

Definition at line 2954 of file libpdv.c.

int pdv_set_gain_duncan_ch (PdvDev * **pdv_p**, *int* **value**, *int* **ch**)

Set gain for Redlake (formerly Duncantech) DT and MS series cameras.

ref. DuncanTech User Manual Doc # 9000-0001-05

Note:
Convenience routine, for Duncantech (Redlake) DT/MS series cameras only. Intended as a starting point for programmers wishing to use EDT serial commands with Duncantech cameras. These subroutines can be used as a template for controlling camera parameters beyond simple exposure and gain.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

value gain value

ch camera channel

See also:
pdv_send_duncan_frame, pdv_read_duncan_frame

Definition at line 2797 of file libpdv.c.

void pdv_set_header_dma (PdvDev * **pdv_p**, *int* **header_dma**)

Sets the boolean value for whether the image header is included in the DMA from the camera.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

header_dma new value (0 or 1) for the header_dma attribute.

See also:
pdv_get_header_dma

Returns:
void

Definition at line 6027 of file libpdv.c.

*void pdv_set_header_offset (PdvDev * pdv_p, int header_offset)*

Sets the byte offset of the header data in the allocated buffer.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

header_offset new value for the header offset.

Returns:

void

Definition at line 6043 of file libpdv.c.

*void pdv_set_header_position (PdvDev * pdv_p, HdrPosition header_position)*

Sets the header (or footer) position.

Originally one of PDV_HEADER_BEFORE, PDV_HEADER_WITHIN, PDV_-HEADER_AFTER, later changed to the HdrPosition enumerated values:

HeaderNone,

HeaderBefore,

HeaderBegin,

HeaderMiddle,

HeaderEnd,

HeaderAfter,

HeaderSeparate

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

header_position the astarting point for the header position

See also:

pdv_get_header_offset, pdv_set_header_offset

Returns:

void

Definition at line 6008 of file libpdv.c.

*void pdv_set_header_size (PdvDev * pdv_p, int header_size)*

Sets the header (or footer) size, in bytes, for the device.

This can also be done by using the **header_size** directive in the camera configuration file.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

header_size new value for header size.

See also:

pdv_get_header_size, **header_size** directive in the Camera Configuration Guide

Returns:

void

Definition at line 5979 of file libpdv.c.

*pdv_set_header_type (PdvDev * pdv_p, int header_type, int irig_slave, int irig_offset, int irig_raw)*

Sets the header (or footer) type.

Enables header (or footer) functionality including position, size, dma, and associated registers for tagging data with magic number, count, and timestamp data.

Currently only one type, HDR_TYPE_IRIG2 is defined. For more about the IRIG functionality on the PCIe8 DV C-Link, see the Timestamping appendinx in the User's Guide.

This subroutine and the associated camera config directive **method_header_type** encapsulate setting the header logic for a specific method in a single operation. Header functionality can also be implemented by setting the header directives directly, via pdv_set_header_size, pdv_set_header_dma, pdv_set_header_offset, etc.

The subroutine will return a fail code if the EDT device is one that does not support this feature. Currently the PCIe8 DV C-link, PCIe4 DVa C-Link and PCIe4 DVa C-link boards support the IRIGB footer (any newer boards are expected to do so as well.) Note that only the device type, not the firmware rev, is checked, and PCIe8 firmware revs earlier than 4/22/2010 did not support HDR_TYPE_IRIG2. So programmers should make sure their board firmware is up-to-date with 4/22/2010 or later firmware via pciload. Applications can check edt_get_board_id

header type may be alternately set at init time via the configuration file directive **method_header_type: IRIG2**

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

header_type header type, as described above

irig_slave set to 1 if IRIGB time source is from a different device (or not present), 0 otherwise

irig_offset timecode offset, set to 2 typically (ignored if irig_slave is not set)

irig_raw enables irig timecode (ignored if irig_slave is not set)

Returns:

0 in success, -1 on failure

See also:

pdv_set_header_size, **method_header_type** directive in the Camera Configuration Guide, and the Timestamp appendix in the Users guide..

Definition at line 5782 of file libpdv.c.

int pdv_set_height (PdvDev * **pdv_p***, int* **value***)*

Sets height and reallocates buffers accordingly.

Since we rarely ever set height and not width, you should normally just use pdv_setsize to set both at once.

5/17/2012: added call to pdv_set_roi to specified height, avoids having to reset ROI separately when the height is changed

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

value the new height.

Definition at line 1285 of file libpdv.c.

int pdv_set_roi (PdvDev * **pdv_p***, int* **hskip***, int* **hactv***, int* **vskip***, int* **vactv***)*

Sets a rectangular region of interest, supporting cropping.

Sets the coordinates of a rectangular region of interest within the image. Checks the camera **width** and **height** directives in the configuration file and returns an error if the coordinates provided are out of range. Use this with pdv_enable_roi, which enables the region of interest.

Note that hactv + hskip should always be less than or equal to the actual output width of the camera, and vact + vskip should be less than or equal to the number of output lines.

An initial region of interest can be set from the config file with the hactv, hskip, vactv, and vskip directives.

Note:
Region of Interest may not work with some very old cameras which required special bitfiles. It will work with most DV, DVK, and all Camera Link boards (inlcluding DVFOX with RCX C-LINK).

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

hskip the X coordinate of the upper left corner of the region of interest.

hactv the width (number of pixels per line) of the region of interest.

vskip the Y coordinate of the upper left corner of the region of interest.

vactv the height (number of lines per frame) of the region of interest.

Example

```
//use the region of interest calls to cut off a 10 pixel wide
//border around the image.
       int cam_w = pdv_get_cam_width(pdv_p);
       int cam_h = pdv_get_cam_height(pdv_p);
       int hactv = cam_w - 20
       int vactv = cam_h - 20
       int hskip = 10;
       int vskip = 10;
       pdv_set_roi(pdv_p, hskip, hactv, vskip, vactv);
       pdv_enable_roi(pdv_p, 1);
```

Returns:
0 on success, -1 on failure.

See also:
pdv_enable_roi, **vskip**, **vactv**, **hskip**, **hactv** directives in the Camera Configuration Guide

Definition at line 7811 of file libpdv.c.

*int pdv_set_shutter_method (PdvDev * pdv_p, int method, unsigned int mcl)*

Set the device's exposure method and CC line state.

Typically the exposure method is set in the config file via the **method_camera_shutter_timing** and **MODE_CNTL_NORM** directives. This subroutine provides a programatic way to do the same thing, post-configuration.

The most common values for **method** (defined in pdv_dependent.h) are:

> **AIA_SERIAL:** Default. Expose timing is controlled via serial or other (camera-dependent) method and the board's hardware is not involved in timing the shutter.

AIA_MCL: CC pulse-width timing, millisecond granularity. Each image capture request (e.g. pdv_start_image) will cause the board to set the EXPOSE (CC) line or lines (as set via the **mcl** parameter's left nibble) TRUE for the current expose time in milliseconds, as set by pdv_set_exposure.

AIA_MCL_100US: CC pulse-width timing, 100 microsecond granularity. Each image capture request (e.g. pdv_start_image) will cause the board to set the EXPOSE (CC) line or lines (as set via the **mcl** parameter's left nibble) TRUE for the current expose time in 100 microsecond increments, as set by pdv_set_exposure.

Several other methods are defined, but most are specific to legacy AIA cameras / framegrabbers and are not applicable to Camera Link. For more information on all available methods see the Camera Configuration Guide.

The **mcl** parameter sets the state of the four camera control (CC) lines, as an 8-bit hexidecimal number. The right nibble sets the steady state of the CC lines, and the left nibble selects which of these lines, if any, the framegrabber hardware use to send out a trigger or expose pulse on each capture request. Most commonly, this value will be 0x00 when the camera generates images continuously or is triggered via an external source, or 0x10 if the board should send out a trigger pulse (1 millisecond, if **method** equals AIA_SERIAL) or timed pulse (as set via pdv_set_exposure if **method** equals AIA_MCL or AIA_MCL_100US) on the CC1 line on each image capture request. See the the Camera Configuration Guide for information on the less common values.

Note:
The AIA Camera Link specification doesn't define how the four CC lines should be used, if at all. However in our experience, virtually all Camera Link cameras that have CC-driven trigger or expose modes use CC1, which corresponds to an **mcl** value of 0x10. For more details see your camera's documentation, and the description of **0x07 Mode Control register** in the Firmware Guide for Camera Link.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
method method (see above)
mcl mode control (CC line) state (see above)

See also:
pdv_get_shutter_method

Returns:
0 on success, -1 on failure

Definition at line 6267 of file libpdv.c.

*int pdv_set_width (PdvDev * pdv_p, int value)*

Sets width and reallocates buffers accordingly.

Since we rarely ever set width and not height, you should normally just use pdv_setsize to set both.

5/17/2012: added call to pdv_set_roi to set specified width, avoids having to reset ROI separately when the width is changed

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

value the new width.

Definition at line 1261 of file libpdv.c.

*int pdv_setsize (PdvDev * pdv_p, int width, int height)*

Sets the width and height of the image.

Tells the driver what width and height (in pixels) to expect from the camera. This call is ordinarily unnecessary in an application program, because the width and height are set automatically when *initcam* runs. Exceptions can occur, however; for example, if the camera's output size can be changed while running, or if the application performs setup that supersedes initcam. This routine is provided for these special cases.

5/17/2012: added call to pdv_set_roi to specified width and height, eliminating the need to call it separately, given that ROI is usually enabled by default.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

width width of the image in pixels.

height height of the image in pixels.

Returns:

0 on success, -1 on failure.

Definition at line 833 of file libpdv.c.

*int pdv_shutter_method (PdvDev * pdv_p)*

Return shutter (expose) timing method.

This subroutine returns only the timing method, not the mode control (CC lines) state. Generally you'll want both so it's recommended to use the newer pdv_get_shutter_method() call. See the description for pdv_set_shutter_method() for explanation of the return values.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
the shutter (expose) timing method

See also:
pdv_set_shutter_method, pdv_get_shutter_method

Definition at line 6212 of file libpdv.c.

Initialization

Read configuration files and initialize the board and camera.

Typically the external utility program *initcam* handles these tasks (possibly invoked by an EDT application such as *pdvshow* or *camconfig*.) *initcam* calls these subroutines to do the work, and they are avaiable as well for programmers who wish to invoke them directly from a user application. See the initcam.c source code for an example of how how to use these subroutines to read configration files and initialize the board from within an application.

Functions

Dependent * pdv_alloc_dependent ()

Allocates a dependent structure, for use by pdv_readcfg and pdv_initcam, and checks for and reports error conditions as a result of the alloc.

---

int pdv_auto_set_timeout (PdvDev *pdv_p)

Sets a reasonable image timeout value based on image size and exposure time (if set) and pixel clock speed (also if set).

---

int pdv_initcam (EdtDev *pdv_p, Dependent *dd_p, int unit, Edtinfo *ei_p, const char *cfgfname, char *bitdir, int pdv_debug)

Initializes the framegrabber board and camera.

---

int pdv_readcfg (const char *cfgfile, Dependent *dd_p, Edtinfo *ei_p)

Reads a configuration file and fills in the dependent and edtinfo structures based on the information in the file.

Function Documentation

*Dependent** pdv_alloc_dependent ()*

Allocates a dependent structure, for use by pdv_readcfg and pdv_initcam, and checks for and reports error conditions as a result of the alloc.

The structure can be deallocated with free() later.

Returns:
pointer to a Dependent structure (defined in camera.h).

See also:
pdv_initcam, initcam.c and camera.h source files.

Definition at line 239 of file pdv_initcam.c.

*int pdv_auto_set_timeout (PdvDev * pdv_p)*

Sets a reasonable image timeout value based on image size and exposure time (if set) and pixel clock speed (also if set).

Note:
This subroutine is called by pdv_initcam so it generally isn't necessary to call it from a user application. Nevertheless it can be useful to know how initcam sets the default timeout value (and how to override it); hence this description.
pdv_initcam calls this subroutine after reading in the various camera parameters from the config file. Since most configs don't (presently) have a **pclock_speed** directive specified, it assumes a conservative 5 Mhz pixel clock speed, which can make for a long timeout value. As a result, for faster cameras in general, and large format ones specifically, if data loss occurs for whatever reason, the *pdv_wait* acquisition routines may block for an excessively long time if data loss occurs. To get around this, either add a **pclock_speed** directive to the config file (preferred), or set your own fixed timeout override with the **user_timeout** directive or pdv_set_timeout.

See also:
pdv_initcam, pdv_set_timeout, pdv_set_exposure, **pclock_speed** & **user_timeout** directive in the Camera Configuration Guide

Returns:
0 on success, -1 on failure.

*int pdv_initcam (EdtDev * pdv_p, Dependent * dd_p, int unit, Edtinfo * ei_p, const char * cfgfname, char * bitdir, int pdv_debug)*

Initializes the framegrabber board and camera.

This is the "guts" of the *inticam* program that gets executed to initialize when you choose a camera. The library subroutine is provided for programmers who wish to incorporate the initialization procedure into their own applications.

Note:
unlike other pdv library calls, pdv_inticam requires an edt device pointer returned from edt_open or edt_open_channel. After initializing, close the device with edt_close before reopening with pdv_open_channel or pdv_open for futher use.
pdv_initcam is designed to initialize EDT framegrabber (input) boards only. For simulator boards, (e.g. the PCIe8 DVa CLS) see the clsiminit.c example/utility application.

Parameters:
pdv_p pointer to edt device structure returned by edt_open

dd_p pointer to a previously allocated (via pdv_alloc_dependent) and initialized (through pdv_readcfg) dependent structure. The library uses this until it is either freed by edt_close, or no longer used by later calls to this function (which means that if you call pdv_initcam again, you should free() pdv_p->dd_p first to avoid memory leaks).

unit unit number of the device. The first unit is 0.

edtinfo miscellaneous variable information structure, defined in initcam.h, initialized via pdv_readcfg.

cfgfname path name of configuration file.

bitdir directory path name for .bit (FPGA) files. If NULL, pdv_initcam will search for bitfiles under ".", then "./camera_config/bitfiles".

pdv_debug should be set to 0 (but is ignored currently).

Returns:
0 on success, -1 on failure

Example

Note:
The following is simplified example code. Normally, we would check the return values and handle error conditions. See *initcam.c* for a complete example of reading the configuration file and configuring the pdv device driver and camera.

```
Dependent *dd_p;
Edtinfo ei_p;
EdtDev *edt_p;
int unit, channel;
char* unitstr = argv[1];

dep = pdv_alloc_dependent();
pdv_readcfg(cfgfname, dd_p, &edtinfo);
unit = edt_parse_unit_channel(unitstr, edt_devname, "pdv", &channel);
edt_p = edt_open_channel(edt_devname, unit, channel);
pdv_initcam(edt_p, dd_p, unit, &ei_p, cfgfname, bitdir, 0);
edt_close(edt_p);
free(dd_p);
```

See also:
pdv_readcfg, initcam.c source code

Definition at line 134 of file pdv_initcam.c.

int pdv_readcfg (const char * **cfgfile**, *Dependent* * **dd_p**, *Edtinfo* * **ei_p**)*

Reads a configuration file and fills in the dependent and edtinfo structures based on the information in the file.

These structures can then be passed in to pdv_initcam to initialize the board and camera.

Parameters:

cfgfile path name of configuration file to read

dd_p device and camera dependent information structure to fill in, defined in camera.h (user-allocated – see pdv_alloc_dependent) – persistent (stored in the driver)

ei_p structure holding non-persistent initialization strings and variables (information not in dd_p). Defined in initcam.h.

Returns:

0 on success, -1 on failure

See also:

pdv_initcam, initcam.c and initcam.h Utility application source code

Definition at line 322 of file readcfg.c.

Acquisition

Image acquisition subroutines.

The simplest way to acquire an image from an EDT digital imaging board is to use pdv_image (see simplest_take.c for an example).

Using pdv_start_image / pdv_wait_image splits image acquisition into *queue* and *retrieve* phases, allowing programmers to parallelize image acquisition and processing (see simple_take.c).

Using pdv_start_images with pdv_wait_images (or pdv_wait_image) adds prestart / queuing for further optimization. Other subroutines are provided for more specialized uses (see other ***simple_*∗*.c*** example programs).

Image acquisition subroutines such as pdv_wait_image return the data as a pointer to the image buffer. Images are not framed in any way, the buffer only contains the pixel data. Application programs should use query routines such as pdv_get_width, pdv_get_height and pdv_get_depth to find out the data line or frame size and number of bits per pixel.

The bitwise format of the pixel data will depend on the number of bits per pixel as defined by the camera and configuration file, as well as any data deinterleave or demosaicing method (e.g. bayer interpolation) that may be enabled via the config file's **method_interlace** directive (exception: **_raw** subroutines bypass data re-ordering). Pixel data for typical formats and re-ordering methods are as follows:

Camera Output	**Config Attributes (also see the Camera Configuration guide)**	**Buffer data**
Monochrome 8 bits	depth: 8 extdepth: 8 cl_data_path_norm: 07 (single ch.) or 17 (dual ch.)	1 byte/pixel
Monochrome 10-16 bits	depth: 10, 12, 14 or 16 extdepth: same as depth CL_DATA_PATH_-NORM: 09, 0b, 0d or 0f (single ch.), 19, 1b, 1d or 1f (dual ch.)	2 bytes/pixel, msb-justified
Bayer color 8 bits	depth: 24 extdepth: 8 CL_DATA_PATH_-NORM: 07 (single ch.), 17 (dual ch.) method_interlace: BGGR	3 bytes/pixel, B G R
Bayer color 10-16 bits	depth: 24 extdepth: 10, 12, 14 or 16 CL_DATA_PATH_-NORM: 09, 0b, 0d or 0f (single ch.), 19, 1b, 1d or 1f (dual ch.) method_interlace: BGGR_WORD	3 bytes/pixel, B G R
RGB color 24 bits	depth: 24 extdepth: 24 CL_CFG_NORM: 01	3 bytes/pixel, B G R
RGB color 30 bits	depth: 32 extdepth: 32 rgb30: 1 or 3 CL_CFG_NORM: 01 (note: PCI DV C-Link and PCIe4/8 DVa C-Link boards must be flashed with medium mode FPGA [see the users guide])	4 bytes/pixel, 8B 8G 8R 2B 2G 2R 2x

Functions

u_char ** pdv_buffer_addresses (PdvDev *pdv_p)

Returns the addresses of the buffers allocated by the last call to pdv_multibuf or pdv_set_buffers.

---

int pdv_cl_get_fv_counter (PdvDev *pdv_p)

Gets the number of frame valid transitions that have been seen by the board since the last time the board/channel was initialized or the last time pdv_cl_reset_fv_counter was called.

---

void pdv_cl_reset_fv_counter (PdvDev *pdv_p)

Resets the frame valid counter to zero.

---

void pdv_flush_channel_fifo (PdvDev *pdv_p)

OBSOLETE: just use pdv_flush_fifo(pdv_p) now.

---

void pdv_flush_fifo (PdvDev *pdv_p)

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

---

int pdv_force_single (PdvDev *pdv_p)

Returns the value of the **force_single** *flag.*

---

u_char * pdv_get_last_image (PdvDev *pdv_p)

Returns a pointer to the last image that was acquired (non-blocking).

---

u_char * pdv_get_last_raw (PdvDev *pdv_p)

get last raw image.

---

int pdv_get_lines_xferred (PdvDev *pdv_p)

Gets the number of lines transferred during the last acquire.

---

int pdv_get_timeout (PdvDev *pdv_p)

Gets the length of time to wait for data on acquisition before timing out.

---

int pdv_get_width_xferred (PdvDev *pdv_p)

Gets the number of pixels transferred during the last line transferred.

---

unsigned char * pdv_image (PdvDev *pdv_p)

Start image acquisition if not already started, then wait for and return the address of the next available image.

---

unsigned char * pdv_image_raw (PdvDev *pdv_p)

Start image acquisition if not already started, then wait for and return the address of the next available image (unprocessed).

---

int pdv_in_continuous (PdvDev *pdv_p)

Gets the status of the continuous flag.

---

int pdv_interlace_method (PdvDev *pdv_p)

Returns the interlace method, as set from the **method_interlace** *directive in the configuration file [from pdv_initcam].*

---

unsigned char * pdv_last_image_timed (PdvDev *pdv_p, u_int *timep)

Identical to pdv_wait_last_image_timed; included for backwards compatability only.

---

unsigned char * pdv_last_image_timed_raw (PdvDev *pdv_p, u_int *timep, int doRaw)

Identical to pdv_wait_last_image_timed_raw; included for backwards compatability only.

---

int pdv_multibuf (PdvDev *pdv_p, int numbufs)

Sets the number of multiple buffers to use in ring buffer continuous mode, and allocates them.

---

int pdv_overrun (PdvDev *pdv_p)

Determines whether data overran on the last aquire.

---

int pdv_read (PdvDev *pdv_p, unsigned char *buf, unsigned long size)

Reads image data from the EDT framegrabber board.

---

int pdv_set_buffers (PdvDev *pdv_p, int numbufs, unsigned char **bufarray)

Used to set up user-allocated buffers to be used in ring buffer mode, cannot be used on systems that have more than 3.5GB/memory (ie the subroutine has been depricated for all practical purposes, instead use pdv_multibuf).

---

void pdv_set_fval_done (PdvDev *pdv_p, int enable)

Enables frame valid done functionality on the board.

---

int pdv_set_timeout (PdvDev *pdv_p, int value)

Sets the length of time to wait for data on acquisition before timing out.

---

void pdv_setup_continuous (PdvDev *pdv_p)

Performs setup for continuous transfers.

void pdv_setup_continuous_channel (PdvDev *pdv_p)

Obsolete.

void pdv_setup_dma (PdvDev *pdv_p)

Sets up device for DMA.

void pdv_start_expose (PdvDev *pdv_p)

Start expose independent of grab - only works in continuous mode.

void pdv_start_hardware_continuous (PdvDev *pdv_p)

Starts hardware continuous mode.

void pdv_start_image (PdvDev *pdv_p)

Starts acquisition of a single image.

void pdv_start_images (PdvDev *pdv_p, int count)

Starts multiple image acquisition.

void pdv_stop_continuous (PdvDev *pdv_p)

Performs un-setup for continuous transfers.

void pdv_stop_hardware_continuous (PdvDev *pdv_p)

Stops hardware continous mode.

int pdv_timeout_cleanup (PdvDev *pdv_p)

Cleans up after a timeout, particularly when you've prestarted multiple buffers or if you've forced a timeout with edt_do_timeout.

int pdv_timeout_restart (PdvDev *pdv_p, int restart)

Cleans up after a timeout, particularly when you've prestarted multiple buffers or if you've forced a timeout with edt_do_timeout.

int pdv_timeouts (PdvDev *pdv_p)

Returns the number of times the device timed out (frame didn't transfer completely or at all) since the device was opened.

unsigned char * pdv_wait_image (PdvDev *pdv_p)

Wait for the image started by pdv_start_image, or for the next image started by pdv_start_images.

unsigned char * pdv_wait_image_raw (PdvDev *pdv_p)

Identical to pdv_wait_image, except image data is returned directly from DMA, bypassing any post-processing that may be in effect.

---

unsigned char * pdv_wait_image_timed (PdvDev *pdv_p, u_int *timep)

Identical to pdv_wait_image but also returns the time at which the DMA was complete on this image.

---

unsigned char * pdv_wait_image_timed_raw (PdvDev *pdv_p, u_int *timep, int doRaw)

Identical to pdv_wait_image_timed, except the new argument doRaw *specifies whether or not to perform the deinterleave.*

---

u_char * pdv_wait_images (PdvDev *pdv_p, int count)

Waits for the images started by pdv_start_images.

---

unsigned char * pdv_wait_images_raw (PdvDev *pdv_p, int count)

Identical to the pdv_wait_images, except that it skips any image deinterleave method defined by the **method_interlace** *config file directive.*

---

unsigned char * pdv_wait_images_timed (PdvDev *pdv_p, int count, u_int *timep)

Identical to pdv_wait_images but also returns the time at which the DMA was complete on the last image.

---

unsigned char * pdv_wait_images_timed_raw (PdvDev *pdv_p, int count, u_int *timep, int doRaw)

Identical to pdv_wait_images_timed, except the new argument doRaw *specifies whether or not to perform the deinterleave.*

---

unsigned char * pdv_wait_last_image (PdvDev *pdv_p, int *nSkipped)

Waits for the last image that has been acquired.

---

unsigned char * pdv_wait_last_image_raw (PdvDev *pdv_p, int *nSkipped, int doRaw)

Identical to the pdv_wait_last_image, except that it provides a way to determine whether to include or bypass any image deinterleave that is enabled.

---

unsigned char * pdv_wait_last_image_timed (PdvDev *pdv_p, u_int *timep)

Identical to pdv_wait_last_image, but also returns the time at which the DMA was complete on the last image.

---

unsigned char * pdv_wait_last_image_timed_raw (PdvDev *pdv_p, u_int *timep, int doRaw)

Identical to pdv_wait_last_image_raw but also returns the time at which the DMA was complete on the last image.

---

unsigned char * pdv_wait_next_image (PdvDev *pdv_p, int *n-Skipped)

Waits for the next image, skipping any previously started images.

---

unsigned char * pdv_wait_next_image_raw (PdvDev *pdv_p, int *n-Skipped, int doRaw)

Identical to the pdv_wait_next_image, except that it provides a way to include or bypass any image deinterleave method defined by the **method_interlace** *config file directive.*

Function Documentation

*u_char** pdv_buffer_addresses (PdvDev * pdv_p)*

Returns the addresses of the buffers allocated by the last call to pdv_multibuf or pdv_set_buffers.

See pdv_wait_images for a description and example of use.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
An array of pointers to image buffers. The size of the array is equal to the number of buffers allocated.

See also:
pdv_multibuf, pdv_set_buffers

Definition at line 6959 of file libpdv.c.

*int pdv_cl_get_fv_counter (PdvDev * pdv_p)*

Gets the number of frame valid transitions that have been seen by the board since the last time the board/channel was initialized or the last time pdv_cl_-reset_fv_counter was called.

The number returned is NOT the number of frames read in; the counter on the board counts all frame ticks seen whether images are being read in or not. As such this subroutine can be useful in determining whether a camera is connected (among other things), assuming that the camera is a freerun camera or has a continuous external trigger.

Note:
This subroutine only works on EDT Camera Link boards.

Returns:
number of frame valids seen

See also:
pdv_reset_fv_counter

Definition at line 10132 of file libpdv.c.

*void pdv_cl_reset_fv_counter (PdvDev * pdv_p)*

Resets the frame valid counter to zero.

Note:
This subroutine only works on EDT Camera Link boards.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

See also:
pdv_get_fv_counter

Returns:
void

Definition at line 10150 of file libpdv.c.

*void pdv_flush_fifo (PdvDev * pdv_p)*

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

This subroutine effectively resets the board, so calling it after every image will degrade performance and is not recommended. Additionally, resetting after a timeout, involves more than just flushing the FIFOs – therefore we recommend using pdv_timeout_restart to reset (which calls this, among other things).

Parameters:
pdv_p pointer to edt device structure returned by edt_open or edt_open_channel

Returns:
void

Definition at line 8425 of file libpdv.c.

*int pdv_force_single (PdvDev * pdv_p)*

Returns the value of the **force_single** flag.

This flag is 0 by default, and is set by the **force_single** directive in the config file (see Camera Configuration Guide). This flag is generally set in cases where the camera uses a trigger method that will violate the pipelining of multiple ring buffers. Most cameras are either continuous, or triggered from the frame grabber, or triggered externally through a trigger line, and won't have this flag set. But a very few cameras use a serial command or similar to trigger the camera, and possibly require a response to be read, in which case the parallel scheme won't work. It is for such cases that this variable is meant to be used. In these cases, the application should allocate only a single buffer (pdv_multibuf(pdv_p, 1)), and should never pre-start more than one buffer before waiting for it.

The take.c program has an example of use of this routine.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Value of the **force_single** flag.

See also:
force_single `camera configuration` directive, pdv_multibuf

Definition at line 3673 of file libpdv.c.

u_char pdv_get_last_image (PdvDev * pdv_p)*

Returns a pointer to the last image that was acquired (non-blocking).

It will return a pointer to the same buffer if called a second time with no new images acquired.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Address of the last image acquired.

See also:
pdv_wait_last_image, pdv_wait_last_image_raw, pdv_wait_image, edt_done_count

Definition at line 6915 of file libpdv.c.

u_char∗ pdv_get_last_raw (PdvDev ∗ pdv_p)

get last raw image.

Identical to the pdv_get_last_image, except that it skips any image deinterleave method defined by the **method_interlace** config file directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Address of the last image acquired.

See also:
pdv_get_last_image

Definition at line 6937 of file libpdv.c.

int pdv_get_lines_xferred (PdvDev ∗ pdv_p)

Gets the number of lines transferred during the last acquire.

Typically only used in line scan applications where the actual number of lines transferred into a given buffer is unknown at the time of the acquire (see also **fval_done** config file directive.) an interrupt (such as from an external sensor) that tells the device to stop acquiring before a full buffer has been read in. Note that if acquires are continuously being queued (as in `pdv_start_images(pdv_p, n)` where *n* is greater than 1), the number of lines tranferred may not reflect the last finished acquire.

Returns:
number of lines transferred on the last acquire

See also:
pdv_get_width_xferred

Definition at line 10074 of file libpdv.c.

int pdv_get_timeout (PdvDev ∗ pdv_p)

Gets the length of time to wait for data on acquisition before timing out.

A default time value for this is calculated based on the size of the image produced by the camera in use and set by pdv_open. If this value is 0, acquisition routines such as pdv_image and pdv_wait_image will wait forever for (block) the amount of data requested.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Timeout value, in milliseconds.

Definition at line 991 of file libpdv.c.

*int pdv_get_width_xferred (PdvDev * pdv_p)*

Gets the number of pixels transferred during the last line transferred.

Typically only used in line scan applications where the actual number of pixels transferred per line may not be known (see also **fval_done** config file directive.) an interrupt (such as from an external sensor) that tells the device to stop acquiring before a full buffer has been read in. Note that if lines are continuously being transferred (the normal case), the number of pixels tranferred may not reflect the last finished line.

Returns:
number of pixels transferred on the last line

See also:
pdv_get_lines_xferred

Definition at line 10104 of file libpdv.c.

unsigned char pdv_image (PdvDev * pdv_p)*

Start image acquisition if not already started, then wait for and return the address of the next available image.

This routine is the same as doing a pdv_start_image followed by pdv_wait_image. It is the simplest way to acquire an image, and in single shot applications may be all that is needed. For continuous sequential transfers with fast cameras, particularly when there is processing involved, (including displaying or saving), the separate start / wait calls will usually be necessary in order to avoid skipping images.

The format of the returned data depends on the number of bits per pixel and any post-capture reordering that is enabled via the config file. For detailed information on data formats, see the Acquisition section.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Address of the next available image buffer that has been acquired.

See also:
pdv_start_image, pdv_wait_image

Definition at line 4769 of file libpdv.c.

unsigned char pdv_image_raw (PdvDev * pdv_p)*

Start image acquisition if not already started, then wait for and return the address of the next available image (unprocessed).

This routine is the same as pdv_image but skips the deinterleave step (if enabled via the **method_interlace** config file directive).

Parameters:
pdv_p device struct returned from pdv_open

Returns:
Address of the next available image buffer that has been acquired

See also:
pdv_image

Definition at line 4789 of file libpdv.c.

*int pdv_in_continuous (PdvDev * pdv_p)*

Gets the status of the continuous flag.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 if continuous, 0 if not.

See also:
pdv_setup_continuous, pdv_stop_continuous

Definition at line 5434 of file libpdv.c.

*int pdv_interlace_method (PdvDev * pdv_p)*

Returns the interlace method, as set from the **method_interlace** directive in the configuration file [from pdv_initcam].

This method is used to determine how the image data will be rearranged (if at all) before being returned from pdv_wait_images or pdv_read.

For more on deinterleave methods, see the Camera Configuration Guide.

Note:
the `_raw` acquisition routines bypass the deinterleave logic.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:

One of the following interlace methods, defined in pdv_dependent.h

PDV_BGGR 8-bit Bayer encoded data

PDV_BGGR_DUAL 8-bit Bayer encoded data, from dual channel camera

PDV_BGGR_WORD 10-12 bit Bayer encoded data

PDV_BYTE_INTLV Data is byte interleaved (odd/even pixels are from odd/even lines, 8 bits per pixel).

PDV_WORD_INTLV Data is word interleaved (odd/even pixels are from odd/even lines, 16 bits per pixel).

DALSA_2CH_INTLV Byte data per 2 channel dalsa "A" model sensor format – see Dalsa D4/D7 camera manual

DALSA_4CH_INTLV Byte data with 4 channel Dalsa formatting – see Dalsa D4/D7 camera manual

EVEN_RIGHT_INTLV 8-bit data, pixels in pairs with 1st pixel from left half, 2nd pixel from right half of screen

PDV_FIELD_INTLC Data is byte interleaved (odd/even pixels are from odd/even lines, 8

PDV_FIELD_INTLC Data is field interlaced (odd/even lines are from top/bottom half of image).

PDV_ILLUNIS_BGGR BBGR for Illunis cameras (?)

PDV_ILLUNIS_INTLV Byte interleave from Illunis cameras (?)

PDV_INVERT_RIGHT_INTLV Byte data, even pixels are right half, inverted

PDV_PIRANHA_4CH_INTLV Piranha 4 channel line scan format (see Dalsa Piranha camera manual)

PDV_SPECINST_4PORT_INTLV Spectral instruments format (see Spectral Instruments camera manual)

PDV_WORD_INTLV Deinterlaced, word format

PDV_WORD_INTLV_HILO Deinterlaced, 2-bytes per pixel, even first

PDV_WORD_INTLV_ODD Deinterlaced, 2-bytes per pixel, odd first

See also:

method_interlace directive in the `Camera Configuration Guide`

Definition at line 6359 of file libpdv.c.

*int pdv_multibuf (PdvDev * pdv_p, int numbufs)*

Sets the number of multiple buffers to use in ring buffer continuous mode, and allocates them.

This routine allocates the buffers itself, in kernel or low memory as required by the EDT device driver for optimal DMA.

pdv_multibuf need only be called once after a pdv_open or pdv_open_channel, and before any calls to acquisition subroutines such as pdv_start_images / pdv_wait_images. If image size changes, call pdv_multibuf again to re-allocate buffers with the new image size.

The number of buffers is limited only by the amount of host memory available, up to approximately 3.5GBytes (or less, depending on other OS use of the low 3.5 GB of memory). Each buffer has a certain amount of overhead, so setting a large number, even if the images are small, is not recommended. Four is the recommended number: at any time, one buffer is being read in, one buffer is being read out, one is being set up for DMA, and one is in reserve in case of overlap. Additional buffers may be necessary with very fast cameras; 32 will almost always smooth out any problems with really fast cameras, and if the system can't keep up with 64 buffers allocated, there may be other problems.

Note:
The ring buffer scheme is designed for one primary purpose: optimal acquisition speed. Programmers should resist the temptation to increase the number of buffers and use them for storage or for processing. Instead use memcpy or equivalent to copy each buffer out after the image has been acquired, and do any processing etc. on the copy.

Returns:
0 on success, -1 on failure.

See also:
pdv_buffer_addresses

Definition at line 6781 of file libpdv.c.

*int pdv_overrun (PdvDev * pdv_p)*

Determines whether data overran on the last aquire.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
Number of bytes of data remaining from last aquire. 0 indicates no overrun.

Definition at line 6421 of file libpdv.c.

*int pdv_read (PdvDev * pdv_p, unsigned char * buf, unsigned long size)*

Reads image data from the EDT framegrabber board.

This is the lowest-level method for aquiring an image. pdv_read is not supported on all platforms; and is included mainly for historical reasons; we recommend instead setting up ring buffers using pdv_multibuf and ring buffer subroutines such as pdv_start_image to do the acquire. pdv_read should never be used when ring buffering is in effect (after calling pdv_multibuf), or be mixed with ring buffer acquisition commands.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

buf data buffer.

size size, in bytes, of the data buffer; ordinarily, width * height * bytes per pixel

Example

```
int size = pdv_get_dmasize(pdv_p) ;

unsigned char *buf = malloc(size);
int bytes_returned;
bytes_returned = pdv_read(pdv_p, buf, size;
```

Returns:

The number of bytes read.

Definition at line 4643 of file libpdv.c.

*int pdv_set_buffers (PdvDev * pdv_p, int numbufs, unsigned char ** bufarray)*

Used to set up user-allocated buffers to be used in ring buffer mode, cannot be used on systems that have more than 3.5GB/memory (ie the subroutine has been depricated for all practical purposes, instead use pdv_multibuf).

Note:

Due to PCI and EDT 32-bit driver architecture limitations, we recommend avoiding this subroutine, as it will not work on most systems that have more than 3.5 MB of memory. Instead, use pdv_multibuf to set up ring buffers, and pdv_buffer_addresses to retrieve the list of buffer pointers generated by pdv_multibuf, and copy out to your local buffers if needed.

This function takes an argument that is an array of buffers allocated by the user. The memory pointed to by the array must be in the lower 3.5 GB. Buffers should be page-aligned. We recommend using pdv_alloc which does this in a system-independent way.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

numbufs number of buffers. Must be 1 or greater. Four is recommended for most applications.

bufarray If NULL, the library allocates 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 returned by pdv_image_size and number of buffers specified in this call and will be used as the ring buffers.

Example

```
int size = pdv_image_size(pdv_p);
unsigned char *bufarray[4];
for (i=0; i < 4; i++)
          bufarray[i] = pdv_alloc(size);
pdv_set_buffers(pdv_p, 4, bufarray);
```

Returns:

0 on success, -1 on failure.

See also:

pdv_multibuf, pdv_buffer_addresses

Definition at line 6835 of file libpdv.c.

void pdv_set_fval_done (PdvDev ∗ **pdv_p***, int* **enable***)*

Enables frame valid done functionality on the board.

In most area scan and many line scan applications, the number of lines in the image is known beforehand. EDT boards start reading pixels in when FRAME VALID signal goes TRUE, but as an optimization measure, they ignore the frame valid FALSE and instead return when the expected number of pixels have been read in.

However when the number of lines is not known beforehand (for example in a mail scanner with a sensor that detects the start/end of packages) it becomes necessary to enable image termination on the Frame Valid. This subroutine enables that functionality.

When using this, the number of lines (**height** directive in the configuration file) should be equal to or greater than the largest possible number of lines that will be read in and the ring buffers should be big enough to accomodate the largest possible image. Otherwise, frames will be split across separate buffers.

use pdv_get_lines_xferred after the acquisition returns to find out how many lines actually transferred.

Note:
If the **fval_done: 1** directive is present in the configuration file (preferred), this subroutine to be called with enable=1 during initialization and it will not be necessary to call it from an application.

Returns:
void

See also:
pdv_get_lines_xferred, **fval_done** directive in the Camera Configuration Guide

Definition at line 10031 of file libpdv.c.

*int pdv_set_timeout (PdvDev * pdv_p, int value)*

Sets the length of time to wait for data on acquisition before timing out.

The default timeout value is set at initcam time and recalculated / updated whenever pdv_set_exposure is called (see pdv_auto_set_timeout). Calling this routine with a value of 0 or greater overrides the automatic value. If called with a value of 0, acquisition routines such as pdv_image and pdv_wait_image will wait forever for (block) the amount of data requested. A value between 0 and 65535 sets the timeout to a fixed time (millisecond units). A value of -1 tells the driver to revert to the automatically calculated value.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

value the number of milliseconds to wait for timeout, or 0 to block waiting for data, or -1 to revert to automatic timeouts

Returns:
0 on success, -1 on failure.

See also:
pdv_auto_set_timeout, **user_timeout** directive in the Camera Configuration Guide

Definition at line 953 of file libpdv.c.

*void pdv_setup_continuous (PdvDev * pdv_p)*

Performs setup for continuous transfers.

Shouldn't need to be called by user apps since pdv_start_images, etc. call it already. But it is in some EDT example applications from before this was the case.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

See also:
pdv_stop_continuous

Returns:
void

Definition at line 8499 of file libpdv.c.

*void pdv_setup_continuous_channel (PdvDev * pdv_p)*

Obsolete.

See pdv_setup_continuous.

Definition at line 8479 of file libpdv.c.

*void pdv_setup_dma (PdvDev * pdv_p)*

Sets up device for DMA.

Generally only for internal use.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 610 of file libpdv.c.

*void pdv_start_expose (PdvDev * pdv_p)*

Start expose independent of grab - only works in continuous mode.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 9769 of file libpdv.c.

*void pdv_start_hardware_continuous (PdvDev * pdv_p)*

Starts hardware continuous mode.

When hardware continuous mode is enabled, the hardware waits until the first acquisition request, and starts reading data when it sees the camera's first

FRAME VALID signal going TRUE. Subsequent frames are read in without regard to the state of FRAME VALID, and LINE VALID (and DATA VALID) are depended upon to gate the data.

This functionality is necessary in some cases where the interframe gap is too small for the OS/device driver to be able to respond for images, typically with very high frame rate cameras. The downside to this is that if data is ever dropped as a result of bandwidth saturation or an unplugged cable for example, frame synch will be forever lost, and cannot be regained without either operator intervention or some intelligent image recognition software. Therefore this mode should only be used when it is certain that it is needed.

Note:
Hardware continuous mode can be enabled at init time via the directive **continuous: 1** camera configuration directive.

Parameters:
pdv_p device struct returned from pdv_open

See also:
pdv_stop_hardware_continuous

Returns:
void

Definition at line 6992 of file libpdv.c.

*void pdv_start_image (PdvDev * pdv_p)*

Starts acquisition of a single image.

Returns without waiting for acquisition to complete. Used with pdv_wait_image, which waits for the image to complete and returns a pointer to it. pdv_start_image(pdv_p) is equivalent to pdv_start_images(pdv_p, 1).

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 4807 of file libpdv.c.

*void pdv_start_images (PdvDev * pdv_p, int count)*

Starts multiple image acquisition.

Queues multiple image acquisitions. Recommended to be used with ring buffering (see pdv_multibuf). Returns without waiting for acquisition to complete. Use pdv_wait_image, pdv_wait_images, or pdv_buffer_addresses to get the address(es) of the acquired image(s).

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

count number of images to start. A value of 0 starts freerun. To stop freerun, call pdv_start_images again with a *count* of 1.

Returns:
void

Definition at line 4829 of file libpdv.c.

*void pdv_stop_continuous (PdvDev * pdv_p)*

Performs un-setup for continuous transfers.

Shouldn't need to be called by user apps since other subroutines (e.g. pdv_timeout_cleanup) now call it as needed. But it is still in some EDT example applications from before this was the case.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

See also:
pdv_setup_continuous

Returns:
void

Definition at line 8547 of file libpdv.c.

*void pdv_stop_hardware_continuous (PdvDev * pdv_p)*

Stops hardware continous mode.

See pdv_start_hardware_continuous for further description.

Parameters:
pdv_p device struct returned from pdv_open

See also:
pdv_start_hardware_continuous

Returns:
void

Definition at line 7018 of file libpdv.c.

*int pdv_timeout_cleanup (PdvDev * pdv_p)*

Cleans up after a timeout, particularly when you've prestarted multiple buffers or if you've forced a timeout with edt_do_timeout.

Superseded by pdv_timeout_restart with newer boards such as the VisionLink and PCIe series.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

Returns:

# of buffers left undone; normally will be used as argument to pdv_start_images() if calling routine wants to restart pipeline as if nothing happened (see take.c for example of use)

See also:

pdv_timeout_restart

Definition at line 1112 of file libpdv.c.

*int pdv_timeout_restart (PdvDev * pdv_p, int restart)*

Cleans up after a timeout, particularly when you've prestarted multiple buffers or if you've forced a timeout with edt_do_timeout.

The example programs take.c and simple_take.c have examples of use; note that these examples call pdv_timeout_restart twice, which may be overkill for some applications/cameras. If the system is configured properly (and all cables/ cameras have robust connections), timeouts should not be a factor. Even so, a robust app will handle timeouts gracefully so it is a good idea to experiment with various timeout recovery methods to make sure you have something that works for your situation.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

restart whether to immediately restart acquiring.

Returns:

# of buffers left undone; normally will be used as argument to pdv_start_images if calling routine wants to restart pipeline as if nothing happened (see take.c and simple_take.c for examples of use).

See also:

pdv_timeouts

Definition at line 8583 of file libpdv.c.

*int pdv_timeouts (PdvDev * pdv_p)*

Returns the number of times the device timed out (frame didn't transfer completely or at all) since the device was opened.

Timeouts occur when some or all of the image failed to transfer. Reasons for this range from an unplugged cable to misconfiguration to system bandwidth saturation. If broken images, slow frame rates or blank images are encountered, it will usually be associated with an image timeout.

Frequent timeouts can be a result of the board being in a non-optimal bus slot or other system-related issues. This is especially true with legacy PCI devices such as the PCI DV C-Link. For more information on optimizing your configuration (and system requirements in general) see EDT's `System Requirements web page`.

A robust application will check to see whether the timeout counter has increased after every new acquire, and take appropriate action. Since timeouts are often associated with data overruns or underruns, they frequently indicate an out-of-synch condtion. So for continuous captures, applications should perform a reset and restart following detection of a timeout, by calling pdv_timeout_restart.

Various factors can prevent timeouts from being reported when data is dropped. With some versions of board firmware, if a small amount of data is lost on a line, the board's region-of-interest (ROI) logic will fill in the missing data using blanking between lines, preventing a timeout from occurring (but still resulting in an out-of-synch frame.) This situation can usually be rectified by updating the board firmware, since the new versions (e.g. PCIe board FW versions 14 and later) have the blanking feature disabled by default.

Hardware continuous mode (enabled via pdv_start_hardware_continuous or the fv_once or hardware_continuous config file directives) can be problmatic for timeouts. Since these modes cause the board to ignore all FVAL (frame start) signals beyond the first one in a continuous sequence, losses of relatively small amounts of data won't trigger a timeout, resulting in a persistently out-of-synch condition. The framesync footer logic shown in the **simple_irig2.c** example application was designed as a workaround for this, and more recently (e.g. driver/library versions 5.3.9.4 and later) the framesync logic was incorporated into pdv_timeouts, providing a convenient and transparent way to ensure detection of and recovery from an out-of-synch condition without the need to change any code. See pdv_enable_framesync and the the **method_framesync** directive in the `Camera Configuration Guide` for more on this.

See also:
pdv_timeout_restart, pdv_enable_framesync, **user_timeout** and **method_framesync** directives in the `Camera Configuration Guide`.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Example

```
int t, last_timeouts = 0;

t = pdv_timeouts(pdv_p);
if (t > last_timeouts) {
   printf("got timeout\n");
   // add recovery code here -- see simple_take.c for example
   last_timeouts = t;
}
```

Returns:
The number of timeouts since the device was opened.

See also:
pdv_set_timeout, pdv_get_timeout, pdv_auto_set_timeout

Definition at line 1088 of file libpdv.c.

unsigned char pdv_wait_image (PdvDev * pdv_p)*

Wait for the image started by pdv_start_image, or for the next image started by pdv_start_images.

Returns immediately if the image started by the last call to pdv_start_image is already complete.

Use pdv_start_image to start image acquisition, and pdv_wait_image to wait for it to complete. pdv_wait_image returns the address of the image. You can start a second image while processing the first if you've used pdv_multibuf to allocate two or more separate image buffers.

Note:
`pdv_wait_` subroutines wait for all of the image data (as determined by the configured width, height and depth) to be read in before returning. If data loss occurs during the transfer or there is no incoming camera data, the subroutines return (with partial or no data in the buffer) after the image timeout period has expired - see pdv_timeouts, pdv_set_timeout, pdv_get_timeout, and pdv_auto_set_timeout.

The format of the returned data depends on the number of bits per pixel and any post-capture reordering that is enabled via the config file. For detailed information on data formats, see the Acquisition section.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Example

```
//see also simple_take.c and simplest_take.c example program.
pdv_multibuf(pdv_p, 4);
pdv_start_image(pdv_p);
while(1) {
        unsigned char *image;

        image = pdv_wait_image(pdv_p); //returns the latest image
        pdv_start_image(pdv_p);        //start acquisition of next image

        //process and/or display image previously acquired here
        printf("got image\n");
}
```

Returns:
Address of the image.

See also:
pdv_start_image, pdv_wait_images, pd pdv_wait_image_raw, pdv_wait_-image_timed

Definition at line 4913 of file libpdv.c.

unsigned char pdv_wait_image_raw (PdvDev * pdv_p)*

Identical to pdv_wait_image, except image data is returned directly from DMA, bypassing any post-processing that may be in effect.

Post-processing is enabled by the method_interlace configuration file directive. When no method_interlace directive is present in the camera configuration file, this subroutine is equivalent to pdv_wait_image.

For information about camera configuration directives, see the Camera Configuration Guide.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Example

```
pdv_multibuf(pdv_p, 4);
pdv_start_image(pdv_p);
while(1) {
        unsigned char *image;

        image = pdv_wait_image_raw(pdv_p); //returns the latest image
        pdv_start_image(pdv_p);        //start acquisition of next image

        //process and/or display image previously acquired here
        printf("got raw image\n");
}
```

Returns:
Address of the image.

See also:
pdv_wait_image

Definition at line 4952 of file libpdv.c.

unsigned char pdv_wait_image_timed (PdvDev * pdv_p, u_int * timep)*

Identical to pdv_wait_image but also returns the time at which the DMA was complete on this image.

The argument *timep* should point to an array of unsigned integers which will be filled in with the seconds and microseconds of the time the image was finished being acquired.

Timestamp comes from the system clock (gettimeofday) at the time the image transfer from the camera to the host memory (DMA) is finished. Latency between the end of DMA to when the timestamp is made will be on the order of a few microseconds. There are other delays in the chain – the camera may have a lag between the end of frame valid and the end of sending out the data, and system interrupt response time may also be a factor. For more precise timestamping using an external time input, see the PCIe8 DV C-Link Application Note: Using the Timestamp function for IrigB input.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
timep a pointer to an array of at least two unsigned integers.

Example

```
u_int timestamp[2];

pdv_multibuf(pdv_p, 4);
pdv_start_image(pdv_p);
while(1) {
        unsigned char *image;

        // get the latest image
        image = pdv_wait_image_timed(pdv_p, timestamp);
        pdv_start_image(pdv_p);  //start acquisition of next image

        //process and/or display image previously acquired here
        printf("got image, at time %u\n", timestamp);
}
```

Returns:
Address of the image.

See also:
pdv_wait_image, pdv_start_image, pdv_wait_image_raw, pdv_wait_image_-timed_raw

Definition at line 5000 of file libpdv.c.

unsigned char pdv_wait_image_timed_raw (PdvDev * pdv_p, u_-int * timep, int doRaw)*

Identical to pdv_wait_image_timed, except the new argument *doRaw* specifies whether or not to perform the deinterleave.

If the doRaw option is 0, the deinterleave conversion will be performed; if the doRaw option is 1, the deinterleave conversion will not be performed.

Timestamp comes from the system clock (gettimeofday) at the time the image transfer from the camera to the host memory (DMA) is finished. Latency between the end of DMA to when the timestamp is made will be on the order of a few microseconds. There are other delays in the chain – the camera may have a lag between the end of frame valid and the end of sending out the data, and system interrupt response time may also be a factor. For more precise timestamping using an external time input, see the PCIe8 DV C-Link Application Note: Using the Timestamp function for IrigB input.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

timep a pointer to an array of at least two unsigned integers.

doRaw specifies raw (if 1) or interleaved (if 0) image data.

Example

```
u_int timestamp[2];

pdv_multibuf(pdv_p, 4);
pdv_start_image(pdv_p);
while(1) {
        unsigned char *image;

        // get the latest image
        image = pdv_wait_image_timed_raw(pdv_p, timestamp, 1);
        pdv_start_image(pdv_p);  //start acquisition of next image

        //process and/or display image previously acquired here
        printf("got raw image, at time %u\n", timestamp);
}
```

Returns:

Address of the image.

See also:
pdv_wait_image, pdv_wait_image_raw, pdv_start_image, pdv_wait_image_timed

Definition at line 5046 of file libpdv.c.

u_char * *pdv_wait_images (PdvDev* * **pdv_p**, *int* **count**)*

Waits for the images started by pdv_start_images.

Returns immediately if all of the images started by the last call to pdv_start_images are complete.

Use pdv_start_images to start image acquisition of a specificed number of images and pdv_wait_images to wait for some or all of them to complete. pdv_wait_images returns the address of the last image. If you've used pdv_multibuf to allocate two or more separate image buffers, you can start up to the number of buffers specified by pdv_multibuf, wait for some or all of them to complete, then use pdv_buffer_addresses to get the addresses of the images.

Note:
`pdv_wait_` subroutines wait for all of the image data (as determined by the configured width, height and depth) to be read in before returning. If data loss occurs during the transfer or there is no incoming camera data, the subroutines return (with partial or no data in the buffer) after the image timeout period has expired - see pdv_timeouts, pdv_set_timeout, pdv_get_timeout, and pdv_auto_set_timeout.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

count number of images to wait for before returning. If *count* is greater than the number of buffers set by pdv_multibuf, only the last *count* images will be available when pdv_wait_images returns.

Example

```
// see also simple_take.c example program
unsigned char **bufs;
pdv_multibuf(pdv_p, 4);
pdv_start_images(pdv_p, 4);
pdv_wait_images(pdv_p, 4);
bufs = pdv_buffer_addresses(pdv_p);
for (i=0; i<4; i++)
       process_image(bufs[i]);   // your processing routine
}
```

Returns:
The address of the last image.

See also:
pdv_wait_images_raw

Definition at line 5648 of file libpdv.c.

unsigned char pdv_wait_images_raw (PdvDev * pdv_p, int count)*

Identical to the pdv_wait_images, except that it skips any image deinterleave method defined by the **method_interlace** config file directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

count number of images to wait for before returning. If *count* is greater than the number of buffers set by pdv_multibuf, only the last *count* images will be available when this function returns.

Example

```
// see also simple_take.c example program
unsigned char **bufs;
pdv_multibuf(pdv_p, 4);
pdv_start_images(pdv_p, 4);
pdv_wait_images_raw(pdv_p, 4);
bufs = pdv_buffer_addresses(pdv_p);
for (i=0; i<4; i++)
       process_image(bufs[i]);   // your processing routine
}
```

Returns:
Address of the last image.

Definition at line 5544 of file libpdv.c.

unsigned char pdv_wait_images_timed (PdvDev * pdv_p, int count, u_int * timep)*

Identical to pdv_wait_images but also returns the time at which the DMA was complete on the last image.

The argument *timep* should point to an array of at least two unsigned integers which will be filled in with the seconds and microseconds of the time the last image was finished being acquired.

Timestamp comes from the system clock (gettimeofday) at the time the image transfer from the camera to the host memory (DMA) is finished. Latency between the end of DMA to when the timestamp is made will be on the order of a few microseconds. There are other delays in the chain – the camera may have a lag between the end of frame valid and the end of sending out the data, and system interrupt response time may also be

a factor. For more precise timestamping using an external time input, see the PCIe8 DV C-Link Application Note: Using the Timestamp function for IrigB input.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

count number of images to wait for before returning. If *count* is greater than the number of buffers set by pdv_multibuf, only the last *count* images will be available when this function returns.

timep a pointer to an array of at least two unsigned integers.

Returns:

The address of the last image.

See also:

pdv_start_image, pdv_wait_images, pdv_wait_images_timed_raw

Definition at line 5150 of file libpdv.c.

unsigned char pdv_wait_images_timed_raw (PdvDev * pdv_p, int count, u_int * timep, int doRaw)*

Identical to pdv_wait_images_timed, except the new argument *doRaw* specifies whether or not to perform the deinterleave.

If the doRaw option is 0, the deinterleave conversion will be performed; if the doRaw option is 1, the deinterleave conversion will not be performed.

Timestamp comes from the system clock (gettimeofday) at the time the image transfer from the camera to the host memory (DMA) is finished. Latency between the end of DMA to when the timestamp is made will be on the order of a few microseconds. There are other delays in the chain – the camera may have a lag between the end of frame valid and the end of sending out the data, and system interrupt response time may also be a factor. For more precise timestamping using an external time input, see the PCIe8 DV C-Link Application Note: Using the Timestamp function for IrigB input.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

count number of images to wait for before returning. If *count* is greater than the number of buffers set by pdv_multibuf, only the last *count* images will be available when this function returns.

timep a pointer to an array of at least two unsigned integers.

doRaw specifies raw (if 1) or interleaved (if 0) image data.

Example

```
unsigned char **bufs;
unsigned int timestamp[2];
int doRaw = 1; // true
int number_of_images = 4;

pdv_multibuf(pdv_p, num_images);
pdv_start_images(pdv_p, num_images);
pdv_wait_images_timed_raw(pdv_p, num_images, timestamp, doRaw);
bufs = pdv_buffer_addresses(pdv_p);
printf("got all images. last one at time: %u\n", timestamp);
for (i=0; i<4; i++)
       process_image(bufs[i]);   // your processing routine
}
```

Returns:
The address of the last image.

See also:
pdv_start_image, pdv_wait_images, pdv_wait_images_raw, pdv_wait_images_timed

Definition at line 5107 of file libpdv.c.

unsigned char pdv_wait_last_image (PdvDev * pdv_p, int * nSkipped)*

Waits for the last image that has been acquired.

This is useful if the display cannot keep up with acquisition and it is not necessary to store all images. If this routine is called for a second time before another image has been acquired, it will block waiting for the next image to complete.

The format of the returned data depends on the number of bits per pixel and any post-capture reordering that is enabled via the config file. For detailed information on data formats, see the Acquisition section.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

nSkipped pointer to an integer which will be filled in with number of images skipped, if any.

Returns:
Address of the image.

Example

```
int skipped_frames;
u_char *imagebuf;
imagebuf=pdv_wait_last_image(pdv_p &skipped_frames);
```

See also:
pdv_start_images, pdv_wait_image, pdv_wait_image_raw

Definition at line 5343 of file libpdv.c.

*unsigned char** pdv_wait_last_image_raw (PdvDev * pdv_p, int * nSkipped, int doRaw)

Identical to the pdv_wait_last_image, except that it provides a way to determine whether to include or bypass any image deinterleave that is enabled.

If data reordering is not enabled, the data buffer will be the same whether doRaw is 0 or 1. For more on data reordering, see the **method_interlace** directive in the Camera Configuration Guide.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

nSkipped pointer to an integer which will be filled in with number of images skipped, if any.

doRaw specifies raw (if 1) or interleaved (if 0) image data.

Returns:
Address of the image.

See also:
pdv_start_images, pdv_wait_image, pdv_wait_image_raw

Definition at line 5285 of file libpdv.c.

*unsigned char** pdv_wait_last_image_timed (PdvDev * pdv_p, u_int * timep)

Identical to pdv_wait_last_image, but also returns the time at which the DMA was complete on the last image.

The argument *timep* should point to an array of at least two unsigned integers which will be filled in with the seconds and microseconds of the time the last image was finished being acquired.

Timestamp comes from the system clock (gettimeofday) at the time the image transfer from the camera to the host memory (DMA) is finished. Latency between the end of DMA to when the timestamp is made will be on the order of a few microseconds. There are other delays in the chain – the camera may have a lag between the end of frame valid and the end of sending out the data, and system interrupt response time may also be a factor. For more precise timestamping using an external time input, see the PCIe8 DV C-Link Application Note: Using the Timestamp function for IrigB input.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

timep a pointer to an array of at least two unsigned integers.

Returns:

Address of the image.

See also:

pdv_start_images, pdv_wait_image, pdv_wait_image_raw

Definition at line 5248 of file libpdv.c.

unsigned char pdv_wait_last_image_timed_raw (PdvDev * pdv_p, u_int * timep, int doRaw)*

Identical to pdv_wait_last_image_raw but also returns the time at which the DMA was complete on the last image.

The argument *timep* should point to an array of at least two unsigned integers which will be filled in with the seconds and microseconds of the time the last image was finished being acquired.

Timestamp comes from the system clock (gettimeofday) at the time the image transfer from the camera to the host memory (DMA) is finished. Latency between the end of DMA to when the timestamp is made will be on the order of a few microseconds. There are other delays in the chain – the camera may have a lag between the end of frame valid and the end of sending out the data, and system interrupt response time may also be a factor. For more precise timestamping using an external time input, see the `PCIe8 DV C-Link Application Note: Using the Timestamp function for IrigB input`.

If reordering is not enabled, the data buffer will be the same whether doRaw is 0 or 1. For more on data reordering, see the **method_interlace** directive in the `Camera Configuration Guide`.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

timep a pointer to an array of at least two unsigned integers.

doRaw specifies raw (if 1) or interleaved (if 0).

Returns:

The address of the last image.

See also:

pdv_start_images, pdv_wait_images, pdv_wait_last_image_raw

Definition at line 5197 of file libpdv.c.

*unsigned char** *pdv_wait_next_image (PdvDev * pdv_p, int * nSkipped)*

Waits for the next image, skipping any previously started images.

The format of the returned data depends on the number of bits per pixel and any post-capture reordering that is enabled via the config file. For detailed information on data formats, see the Acquisition section.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

nSkipped pointer to an integer which will be filled in with number of images skipped, if any.

Returns:

Address of the image.

See also:

pdv_start_images, pdv_wait_image, pdv_wait_next_image_raw

Definition at line 5417 of file libpdv.c.

*unsigned char** *pdv_wait_next_image_raw (PdvDev * pdv_p, int * nSkipped, int doRaw)*

Identical to the pdv_wait_next_image, except that it provides a way to include or bypass any image deinterleave method defined by the **method_interlace** config file directive.

If data reordering is not enabled, the data buffer will be the same whether doRaw is 0 or 1. For more on data reordering, see the **method_interlace** directive in the Camera Configuration Guide.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

nSkipped pointer to an integer which will be filled in with number of images skipped, if any.

doRaw specifies raw (if 1) or interleaved (if 0) image data.

Returns:

Address of the image.

See also:

pdv_start_images, pdv_wait_image, pdv_wait_next_image

Definition at line 5369 of file libpdv.c.

Communications/Control

Serial communications and camera control subroutines.

Subroutines in this section of the library fall into three general categories: 1) low level serial communications and control, 2) framing commands for cameras that have sophisticated command framing protocols, and 3) high level convenience routines for specific operations on selected cameras.

These subroutines are used to communicate with cameras that have a serial command set. Since there is (to date) no standard command set, programmers who wish to embed camera control commands within applications will need to write code that is specific to the camera(s) in use.

Serial control typically consists a command/response sequence, and looks like the following:

```
pdv_serial_command(pdv_p, command_string); // ASCII; for binary use pdv_serial_binary_command
n = pdv_serial_wait(pdv_p, timeout, nchars);
pdv_serial_read(pdv_p, response_string, n);
```

The above is the most general purpose method, but it can be slow since pdv_serial_wait will only return after the timeout period expires, in order to ensure that all of the response characters have come in. If the last character of a response is known and can be assured to always be unique within that response, then the use of a *serial wait* character can be used. When set, it causes pdv_serial_wait to return immediately when the character is seen, without waiting for the full timeout period to expire:

```
pdv_set_waitchar(pdv_p, '\n');    // only needs to be set once
pdv_serial_command(pdv_p, command_string);
n = pdv_serial_wait(pdv_p, timeout, nchars); // still use max timeout in case of failure
pdv_serial_read(pdv_p, response_string, n);
```

Note:

When this library was originally developed, there were a relatively small number of cameras and camera command sets to deal with. Consequently, subroutines written to directly control specific camera parameters such as pdv_set_exposure, pdv_set_gain and pdv_set_blacklevel were coded to handle those tasks for the majority of cameras that had such functionality. With the proliferation of cameras and command sets over the years, these "convenience routines" have become less useful. They remain helpful for cameras whose command sets conform to the relatively narrow format defined by the **serial_exposure**, **serial_gain** and **serial_offset** config directives, but even then such control is limited, so for full control of cameras it is usually necessary for programmers to code such control with the lower-level subroutines pdv_serial_command, pdv_serial_binary_command, pdv_serial_wait and pdv_serial_read (or specialized framing commands such as pdv_send_basler_frame). One important exception is if the camera and board are to be set up for *pulse-width*,

aka *level trigger* acquisition control (where the length of the board's shutter timer is used to control the length of the EXPOSE pulse and consequently the camera's integration time). When using that mode (ebabled via the **method_-camera_shutter_timing** configuration directive), pdv_set_exposure *should* be used, since it also controls the board's expose timer.

Functions

int pdv_get_baud (PdvDev *pdv_p)

Get the baud rate, typically initialized by the **serial_baud** *directive in the config file (default 9600).*

---

int pdv_get_serial_block_size ()

Returns the block size for serial writes.

---

int pdv_get_waitchar (PdvDev *pdv_p, u_char *waitc)

Get serial wait character, or byte.

---

int pdv_query_serial (PdvDev *pdv_p, char *cmd, char **resp)

Send a serial command, get the response in a multiline string, one line per string pointer.

---

int pdv_read_basler_frame (PdvDev *pdv_p, u_char *frame, int len)

Read a Basler binary frame command.

---

int pdv_read_duncan_frame (PdvDev *pdv_p, u_char *frame)

Read response (binary serial) from a Duncantech MS and DT series camera – checks for STX and size, then waits for size+1 more bytes.

---

void pdv_reset_serial (PdvDev *pdv_p)

Resets the serial interface.

---

int pdv_send_basler_command (PdvDev *pdv_p, int cmd, int rwflag, int len, int data)

Send a basler binary command – do the framing and BCC.

---

int pdv_send_basler_frame (PdvDev *pdv_p, u_char *cmd, int len)

Send a Basler formatted serial frame.

---

void pdv_send_break (PdvDev *pdv_p)

Send serial break (only Camera Link, and aiag and related FPGA files).

---

int pdv_send_duncan_frame (PdvDev *pdv_p, u_char *cmdbuf, int size)

Send a Duncantech MS / DT series camera frame – adds the framing and checksum, then sends the command.

---

int pdv_send_msg (PdvDev *ed, int chan, const char *buf, int size)

wrapper for edt_send_msg, but added pause between bytes if indicated by pause_for_serial (done initially for imperx cam)

---

int pdv_serial_binary_command (PdvDev *pdv_p, const char *cmd, int len)

Sends binary serial command(s) to the camera.

---

int pdv_serial_binary_command_flagged (PdvDev *pdv_p, const char *cmd, int len, u_int flag)

Sends a binary serial command.

---

int pdv_serial_check_enabled (PdvDev *pdv_p)
int pdv_serial_command (PdvDev *pdv_p, const char *cmd)

Sends an ASCII serial command to the camera, with ASCII camera command formatting.

---

int pdv_serial_command_flagged (PdvDev *pdv_p, const char *cmd, u_int flag)

Bottom level serial_command that takes a flag for different options.

---

int pdv_serial_command_hex (PdvDev *pdv_p, const char *str, int length)

Send hex byte command (formatted ascii "0xNN") as binary.

---

int pdv_serial_get_numbytes (PdvDev *pdv_p)

Returns the number of bytes of unread data in the serial response buffer.

---

char * pdv_serial_prefix (PdvDev *pdv_p)

Get the serial prefix.

---

int pdv_serial_read (PdvDev *pdv_p, char *buf, int count)

Performs a serial read over the serial control lines.

---

int pdv_serial_read_blocking (PdvDev *pdv_p, char *buf, int size)

Performs a serial read over the serial control lines, blocks until all requested serial is read.

---

int pdv_serial_read_disable (PdvDev *pdv_p)
int pdv_serial_read_enable (PdvDev *pdv_p)
int pdv_serial_read_nullterm (PdvDev *pdv_p, char *buf, int size, int nullterm)

Preforms a serial read over the RS-422 or RS-232 lines if EDT has provided a special cable to accommodate RS-422 or RS-232 serial control.

---

char * pdv_serial_term (PdvDev *pdv_p)

Get the serial terminator.

---

void pdv_serial_txrx (PdvDev *pdv_p, char *txbuf, int txcount, char *rxbuf, int rxcount, int timeout, u_char *wchar)

Serial send AND recieve – send a command and wait for the response.

---

int pdv_serial_wait (PdvDev *pdv_p, int msecs, int count)

Waits for response from the camera as a result of a pdv_serial_write or pdv_serial_command.

---

int pdv_serial_wait_next (EdtDev *pdv_p, int msecs, int count)

Wait for next serial to come in – ignore any previous if 0, just wait for the next thing, however many it is.

---

int pdv_serial_write (PdvDev *pdv_p, const char *buf, int size)

Performs a serial write over the serial lines.

---

int pdv_serial_write_available (PdvDev *pdv_p)

pdv_serial_write_avail Get the number of bytes available in the driver's serial write buffer.

---

int pdv_serial_write_single_block (PdvDev *pdv_p, const char *buf, int size)

Writes a serial command buffer to a serial aia (Kodak type) device.

---

int pdv_set_baud (PdvDev *pdv_p, int baud)

Sets the baud rate on the serial lines; applies only to cameras with serial control.

---

void pdv_set_serial_block_size (int newsize)

Sets the block size for serial writes if the default of 512 is not adequate.

---

void pdv_set_serial_delimiters (PdvDev *pdv_p, char *prefix, char *term)

Get the serial prefix.

---

int pdv_set_serial_parity (PdvDev *pdv_p, char parity)

Sets parity to even, odd, or none.

int pdv_set_waitchar (PdvDev *pdv_p, int enable, u_char wchar)

Set serial wait character.

Function Documentation

*int pdv_get_baud (PdvDev * pdv_p)*

Get the baud rate, typically initialized by the **serial_baud** directive in the config file (default 9600).

Returns:
baud rate in bits/sec, or 0 on error

See also:
serial_baud directive in the Camera Configuration Guide

Definition at line 7185 of file libpdv.c.

int pdv_get_serial_block_size (void)

Returns the block size for serial writes.

Returns:
the serial block size

See also:
pdv_get_serial_block_size

Definition at line 3921 of file libpdv.c.

*int pdv_get_waitchar (PdvDev * pdv_p, u_char * waitc)*

Get serial wait character, or byte.

This value, if set, is what pdv_serial_wait will return immediately after it comes in instead of waiting for the serial timeout period to expire.

Parameters:
pdv_p same as it ever was

waitc character (byte) to wait for

Returns:
1 if waitchar enabled, 0 if disabled

See also:
pdv_set_waitchar and **serial_waitchar** directive in the Camera Configuration Guide

Definition at line 6566 of file libpdv.c.

*int pdv_query_serial (PdvDev * pdv_p, char * cmd, char ** resp)*

Send a serial command, get the response in a multiline string, one line per string pointer.

Returns:
the number of strings found. Max return string length is 2048

Definition at line 9161 of file libpdv.c.

*int pdv_read_basler_frame (PdvDev * pdv_p, u_char * frame, int len)*

Read a Basler binary frame command.

Check the framing and BCC – ref. BASLER A202K Camera Manual Doc. ID number DA044003

RETURNS number of characters read back, or 0 if none or failure

Definition at line 4443 of file libpdv.c.

*int pdv_read_duncan_frame (PdvDev * pdv_p, u_char * frame)*

Read response (binary serial) from a Duncantech MS and DT series camera – checks for STX and size, then waits for size+1 more bytes.

Ref. DuncanTech User Manual Doc # 9000-0001-05.

Convenience routine for Duncantech (Redlake) DT/MS series cameras only.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
frame buffer containing the frame read back from the camera

See also:
pdv_send_duncan_frame

Definition at line 4539 of file libpdv.c.

*void pdv_reset_serial (PdvDev * pdv_p)*

Resets the serial interface.

This is mostly used during initialization (*initcam*) to make sure any outstanding reads and writes from previous interrupted applications are cleaned up and to put the serial state machine in a known idle state. Applications typically do not need to call this subroutine.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 7238 of file libpdv.c.

*int pdv_send_basler_command (PdvDev * pdv_p, int cmd, int rwflag, int len, int data)*

Send a basler binary command – do the framing and BCC.

ref. BASLER A202K Camera Manual Doc. ID number DA044003.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
cmd basler command
rwflag read/write flag – 1 if read, 0 if write
len data length
data the data (if any)

Returns:
0 on success, -1 on failure

Definition at line 2676 of file libpdv.c.

*int pdv_send_basler_frame (PdvDev * pdv_p, u_char * cmd, int len)*

Send a Basler formatted serial frame.

Adds the framing and BCC, ref. BASLER A202K Camera Manual Doc. ID number DA044003

RETURNS 0 on success, -1 on failure

Definition at line 4417 of file libpdv.c.

*void pdv_send_break (PdvDev * pdv_p)*

Send serial break (only Camera Link, and aiag and related FPGA files).

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 8791 of file libpdv.c.

*int pdv_send_duncan_frame (PdvDev * pdv_p, u_char * cmdbuf, int size)*

Send a Duncantech MS / DT series camera frame – adds the framing and checksum, then sends the command.

Ref. DuncanTech User Manual Doc # 9000-0001-05.

cmdbuf: command buf: typically includes command, 2 size bytes, and size-1 message butes size: number of message bytes plus command byte

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

cmdbuf buffer containing the command, minus framing information

size number of bytes in the cmdbuf

Returns:

0 on success, -1 on failure

See also:

pdv_read_duncan_frame

Definition at line 4493 of file libpdv.c.

*int pdv_send_msg (PdvDev * ed, int chan, const char * buf, int size)*

wrapper for edt_send_msg, but added pause between bytes if indicated by pause_for_serial (done initially for imperx cam)

Returns:

0 on success, -1 on failure. If an error occurs, call pdv_perror to get the system error message.

Definition at line 3807 of file libpdv.c.

*int pdv_serial_binary_command (PdvDev * pdv_p, const char * cmd, int len)*

Sends binary serial command(s) to the camera.

Applicable only to cameras that use RS-232 or RS-422 binary serial for camera-computer communications. Similar to pdv_serial_command, but for binary instead of ASCII commands, it uses a count instead of a terminating NULL to indicate the end of the data. Also, it doesn't add on any terminating CR or LF characters.

Consult your camera manufacturer user's guide for information on serial command format requirements.

For a detailed example of serial communications, see the serial_cmd.c example program.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

cmd buffer containing serial command(s)

len number of bytes to send

Returns:

0 on success, -1 on failure.

See also:

pdv_serial_command, pdv_serial_read, pdv_serial_wait

Definition at line 4404 of file libpdv.c.

*int pdv_serial_binary_command_flagged (PdvDev * **pdv_p**, const char * **cmd**, int **len**, u_int **flag**)*

Sends a binary serial command.

convenience wrapper for pdv_serial_write() – takes the command string and prepends the 'c' to it if FOI, then calls pdv_serial_write(). Because of the FOI issue, applications should ALWAYS use this or one of the other pdv serial command calls (pdv_serial_binary_command, pdv_serial_command_flagged, etc.) instead of calling pdv_serial_write directly

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

cmd command – must be a valid serial command for the camera in use, as defined in camera manufacturer's user's manual

len number of bytes of cmd to write

flag flag bits – so far only SCFLAG_NORESP is defined – tells the driver not to wait for a response before returning

Returns:

0 on success, -1 on failure

Definition at line 4586 of file libpdv.c.

*int pdv_serial_command (PdvDev * **pdv_p**, const char * **cmd**)*

Sends an ASCII serial command to the camera, with ASCII camera command formatting.

Applies only to cameras that use a serial control method for camera-computer communications.

Appends the required serial terminator onto the string before sending. The default serial terminator is the '\r' (carriage return) character, which is the most common serial terminator character for cameras with use ASCII serial command sets. If the **serial_term** directive is present in the config file in use, it will use the terminator specified by that instead. For example, if the camera requires a CR/LF (carriage return/line feed) to terminate instead of just a single carriage return, make sure the following command is in the config file in use:

```
serial_term: "\r\n"
```

Also available but much less common is the serial prefix, which can also be added to any command via the **serial_prefix** camera configuration directive. By default there is no serial prefix.

For a detailed example of serial communications, see the serial_cmd.c example program.

Consult your camera manufacturer's users guide for information on serial command format reqirements.

Example

```
pdv_serial_command(pdv_p, "DEF_ON"); // set defect correction on
```

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

cmd command – must be a valid serial command for the camera in use, as as defined in the camera manufacturer's user's manual

Returns:

0 on success, -1 on failure

See also:

part of this comment.
pdv_serial_term, pdv_serial_prefix, pdv_set_serial_delimiters, pdv_serial_write

Definition at line 4174 of file libpdv.c.

*int pdv_serial_command_flagged (PdvDev * pdv_p, const char * cmd, u_int flag)*

Bottom level serial_command that takes a flag for different options.

Primarily for internal use; applications should avoid calling directly and instead use pdv_serial_command.

The only flag is the SCFLAG_NORESP flag, which says whether to wait for response on FOI. Normal case is no, but internally (when called from pdv_set_exposure, for example) the flag is set to 1 so it doesn't slow down the data stream.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

cmd command to send

flag flag whether to wait for response on FOI

Returns:

0 on success, -1 on failure

Definition at line 4207 of file libpdv.c.

*int pdv_serial_command_hex (PdvDev * pdv_p, const char * str, int length)*

Send hex byte command (formatted ascii "0xNN") as binary.

Assumes the format has already been checked.

Not all that useful for user applications, mainly it's here for special use by pdv_initcam.

Attention:

length is unused – here only for future use if/when we want to send more than one byte at a time. For now only one byte at a time (and only used by initcam really...).

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

str ASCII command string containing "0x%s" hex formatted string

length reserved, for future use

Definition at line 7754 of file libpdv.c.

*int pdv_serial_get_numbytes (PdvDev * pdv_p)*

Returns the number of bytes of unread data in the serial response buffer.

Similar to pdv_serial_wait but doesn't wait for any timeout period, nor does it have any minimum count parameter.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

count Maximum number of bytes to wait for before returning.

Returns:

Number of bytes of unread data in the serial response buffer

Definition at line 6483 of file libpdv.c.

char * *pdv_serial_prefix (PdvDev* * pdv_p*)*

Get the serial prefix.

See pdv_serial_command for more about the serial prefix.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
a character string containing any serial prefix character(s)

See also:
pdv_serial_command

Definition at line 4291 of file libpdv.c.

int pdv_serial_read (PdvDev * pdv_p*, char* * buf*, int* count*)*

Performs a serial read over the serial control lines.

The serial data read will be stored in a user supplied buffer. That buffer will be NULL-terminated. Use pdv_serial_read_nullterm(pdv_p, FALSE) if you don't want that behavior.

Example

```
int count = 64;
// wait for 64 bytes, or timeout, whichever comes first.
int got = pdv_serial_wait(pdv_p, 0, count);
// read the data we waited for.
char buf[count+1];
pdv_serial_read(pdv_p, buf, got);
if (got < count) {
    printf("timeout occurred while waiting for serial data\n");
}
if (got != 0) {
    printf("data read over serial: %s\n", buf);
}
```

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
buf pointer to data buffer–must be preallocated to at least **count** + 1 bytes (**count** bytes of data plus a one byte NULL terminator).
count Number of bytes to be read.

Returns:
the number of bytes read into the buffer

See also:
pdv_serial_wait

Definition at line 3794 of file libpdv.c.

*int pdv_serial_read_blocking (PdvDev * pdv_p, char * buf, int size)*

Performs a serial read over the serial control lines, blocks until all requested serial is read.

Similar to pdv_serial_read but blocks until all requested serial bytes have been received. The serial data read will be stored in a user supplied buffer.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

buf pointer to data buffer–must be preallocated to at least **count** + 1 bytes (**count** bytes of data plus a one byte NULL terminator).

count Number of bytes to be read.

Returns:

the number of bytes read in

See also:

pdv_serial_wait, pdv_serial_read

Definition at line 4072 of file libpdv.c.

*int pdv_serial_read_nullterm (PdvDev * pdv_p, char * buf, int size, int nullterm)*

Preforms a serial read over the RS-422 or RS-232 lines if EDT has provided a special cable to accommodate RS-422 or RS-232 serial control.

The buffer passed in will be NULL-terminated if nullterm is true.

Parameters:

pdv_p device struct returned from pdv_open

buf pointer to data buffer–must be preallocated to at least **count** bytes

size number of bytes to be read, which must be at most one less than the size of the buf (so there is room for the NULL terminator).

nullterm true to null terminate the buffer read in, false to disable that.

Returns:

The number of bytes read into buf.

Definition at line 3713 of file libpdv.c.

*char** pdv_serial_term (PdvDev * pdv_p)

Get the serial terminator.

See pdv_serial_command for more about the serial terminator.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
a character string containing any serial terminator character(s)

See also:
pdv_serial_command

Definition at line 4277 of file libpdv.c.

void pdv_serial_txrx (PdvDev * pdv_p, *char* * txbuf, *int* txcount, *char* * rxbuf, *int* rxcount, *int* timeout, *u_char* * wchar)

Serial send AND recieve – send a command and wait for the response.

Takes both expected receive count and char on which to terminate the receive – if both are specified will return on first one – that is if there's a count of 4 but the 3rd char back is the one specified in wchar, then will return after 3.

Parameters:
pdv_p device handle returned by pdv_open

txbuf buffer to send out

txcount number of characters to send out

rxbuf buffer to hold response

rxcount number of characters expected back

timeout number of milliseconds to wait for expected response

wchar pointer to terminating char (NULL if none)

Returns:
void

Definition at line 9897 of file libpdv.c.

int pdv_serial_wait (PdvDev * pdv_p, *int* msecs, *int* count)

Waits for response from the camera as a result of a pdv_serial_write or pdv_serial_command.

After calling this function, use pdv_serial_read to get the data. For a detailed example of serial communications, see the serial_cmd.c example program.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

msecs number of milliseconds to wait before timing out. If this parameter is 0, the defualt timeout value is used, as specified by the **serial_timeout** directive in the current configuration file. If no default timout value was specified, the default is 1000 milliseconds (1 second).

count Maximum number of bytes to wait for before returning.

Returns:

Number of bytes of serial data returned from the camera.

See also:

pdv_serial_read for simple example.

Definition at line 6452 of file libpdv.c.

*int pdv_serial_wait_next (EdtDev * pdv_p, int msecs, int count)*

Wait for next serial to come in – ignore any previous if 0, just wait for the next thing, however many it is.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

msecs number of milliseconds to wait before timing out

count number maximum number to wait for

Returns:

number of characters seen, can be passed to pdv_serial_read

See also:

pdv_serial_wait, pdv_serial_read

Definition at line 6510 of file libpdv.c.

*int pdv_serial_write (PdvDev * pdv_p, const char * buf, int size)*

Performs a serial write over the serial lines.

This command applies only to cameras that use a serial control method.

This function is mainly for sending binary data over the serial lines to the camera. It can be used for ASCII commands, but pdv_serial_command is generally easier.

For a detailed example of serial communications, see the serial_cmd.c example program.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

buf buffer containing serial command(s)

size number of bytes to send

Returns:

0 on success, -1 on failure.

See also:

pdv_serial_command

Definition at line 3987 of file libpdv.c.

*int pdv_serial_write_available (PdvDev * pdv_p)*

pdv_serial_write_avail Get the number of bytes available in the driver's serial write buffer.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

Returns:

the number of bytes available in the driver's write buffer

Definition at line 3889 of file libpdv.c.

*int pdv_serial_write_single_block (PdvDev * pdv_p, const char * buf, int size)*

Writes a serial command buffer to a serial aia (Kodak type) device.

Note: applications should pretty much ALWAYS use pdv_serial_command or pdv_serial_binary_command instead of calling pdv_serial_write directly since, when a FOI is detected, those two calls prepend the required that is needed to pass the command on to the camera.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

buf string to send to the device

size number of bytes to write

Returns:

0 on success, -1 on failure (and errno is set). If an error occurs, call pdv_perror to get the system error message.

Definition at line 3846 of file libpdv.c.

*int pdv_set_baud (PdvDev * pdv_p, int baud)*

Sets the baud rate on the serial lines; applies only to cameras with serial control.

Valid values are 9600, 19200, 38500, 57500, and 115200.

Note:
The baud rate is ordinarily initialized using the value of the **serial_baud** directive in the configuration file, and defaults to 9600 if the directive is not present. Under most circumstances, applications do not need to set the baud rate explicitly.

Parameters:
baud the desired baud rate.

pdv_p pointer to pdv device structure returned by pdv_open

Returns:
0 on success, -1 on error

Definition at line 7090 of file libpdv.c.

void pdv_set_serial_block_size (int newsize)

Sets the block size for serial writes if the default of 512 is not adequate.

Parameters:
newsize the new serial block size

Returns:
void

Definition at line 3909 of file libpdv.c.

*void pdv_set_serial_delimiters (PdvDev * pdv_p, char * prefix, char * term)*

Get the serial prefix.

The serial prefix (if any) is typically set through the config file, which is that is the preferred way to set up any serial delimiters; calling this subroutine directly should be avoided.

See pdv_serial_command for more about the serial delimiters.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

prefix - see pdv_serial_command

term - see pdv_serial_command

See also:
pdv_serial_command

Returns:
void

Definition at line 4312 of file libpdv.c.

int pdv_set_serial_parity (PdvDev * **pdv_p**, *char* **parity**)

Sets parity to even, odd, or none.

Parameters:
parity the desired partity. Should be 'e', 'o', or 'n' for even, odd, or none (respectively).

pdv_p pointer to pdv device structure returned by pdv_open

Returns:
0 on success, -1 on error

Definition at line 7042 of file libpdv.c.

int pdv_set_waitchar (PdvDev * **pdv_p**, *int* **enable**, *u_char* **wchar**)

Set serial wait character.

Normally pdv_serial_wait will wait until the serial_timeout period expires before returning (unless the max number of characters is seen). This is the most general purpose and robust method since there's no other way of knowing all different camera response formats. However if the camera formats are known, and specifically a if each response can be expected to be 1 line terminated by the same character (such as a newline) every time, then setting the serial_waitchar to that character can greatly shorten the time it takes for a pdv_serial_wait call to return.

This character can also be initialized in the `camera configuration` directive **serial_waitchar**.

Returns:
0 in success, -1 on failure

Definition at line 6537 of file libpdv.c.

Utility

Various utility subroutines.

Most PDV utility routines have a `dvu_` prefix. `dvu_` subroutines are not necessarily specific to the EDT digital imaging hardware. For example, dvu_write_rasfile could concievably be used to write a raster file from any source, not just one captured by an EDT framegrabber. As such, `dvu_` subroutines do not operate on an PdvDev device handle in their parameter lists.

There are a few utility subroutines that don't take a PdvDev device handle but do have a `pdv_` prefix, and may or may not have some PDV specificity.

The remaining `pdv_` subroutines that do take a PdvDev device handle are tagged as utility subroutines because they do not fit any other category.

Defines

#define BI_BITFIELDS 3L
#define BI_RGB 0L
#define BI_RLE4 2L
#define BI_RLE8 1L
#define BYTE unsigned char
#define DVUFATAL PDVLIB_MSG_FATAL
#define DWORD unsigned int
#define LONG int
#define RAS_MAGIC 0x59a66a95
#define RMT_EQUAL_RGB 1
#define RMT_NONE 0
#define RT_STANDARD 1
#define WIDTHBYTES(bits) (((bits) + 31) / 32 * 4)
#define WORD unsigned short

Functions

int dvu_exp_histeq (u_char *src, u_char *dst, int size, int depth, int cutoff)

Perform a histogram equalization on an image, with cutoff (experimental).

---

void dvu_free_tables ()
int dvu_free_window (dvu_window *w)
int dvu_histeq (u_char *src, u_char *dst, int size, int depth)

Perform a histogram equalization on an image.

---

dvu_window * dvu_init_window (u_char *data, int sx, int sy, int dx, int dy, int xdim, int ydim, int depth)
int dvu_load_lookup (char *filename, int depth)
void dvu_long_to_charbuf (unsigned int val, u_char *buf)
int dvu_lookup (u_char *src, u_char *dst, int size, int depth)
void dvu_perror (char *str)
dvu_window * dvu_read_window (char *fname)
dvu_window * dvu_reset_window (dvu_window *s, u_char *data, int sx, int sy, int dx, int dy)
int dvu_save_lookup (char *filename, int depth)
int dvu_winscale (dvu_window *wi, dvu_window *bi, int minbyte, int maxbyte, int doinit)
int dvu_word2byte (u_short *wbuf, u_char *bbuf, int count, int depth)
int dvu_word2byte_with_stride (u_short *wbuf, u_char *bbuf, int wstride, int bstride, int xsize, int ysize, int depth)
int dvu_wordscale (u_short *words, u_char *bytes, int count, int minbyte, int maxbyte, int doinit)
int dvu_write_bmp (char *fname, u_char *buffer, int width, int height)

Writes an 8-bit per pixel data buffer as a grayscale Windows bitmap file.

---

int dvu_write_bmp_24 (char *fname, u_char *buffer, int width, int height)

Writes a 24-bit per pixel RGB data buffer as a Windows bitmap file.

---

int dvu_write_image (char *fname, u_char *addr, int x_size, int y_size, int istride)

Utility routine that outputs a 1-band, 8-bit image to a Sun raster format file (regardless of platform), with stride.

---

int dvu_write_image24 (char *fname, u_char *addr, int x_size, int y_size, int istride)

Writes a 24-bit per pixel RGB data buffer as a Sun Raster format file, with stride.

---

int dvu_write_rasfile (char *fname, u_char *addr, int x_size, int y_size)

Utility routine that outputs a 1-band, 8-bit image to a Sun raster format file (regardless of platform).

---

int dvu_write_rasfile16 (char *fname, u_char *addr, int x_size, int y_size, int depth_bits)

converts 1 band, 10-16 bit image to a sun raster format file and writes to a file.

---

int dvu_write_rasfile24 (char *fname, u_char *addr, int x_size, int y_size)

Writes a 24-bit per pixel RGB data buffer as a Sun Raster format file.

---

int dvu_write_raw (int imagesize, u_char *imagebuf, char *fname)

Writes a 24-bit per pixel RGB data buffer as a raw data file (no formatting).

---

int dvu_write_window (char *fname, dvu_window *w)
int pdv_access (char *fname, int perm)

Determines file access independent of operating system.

---

uchar_t * pdv_alloc (int size)

Convenience routine to allocate memory in a system-independent way.

---

int pdv_bytes_per_line (int width, int depth)

Returns bytes per line based on width and bit depth, including depth < 8.

---

int pdv_cl_camera_connected (PdvDev *pdv_p)

Checks whether a camera is connected and turned on.

---

void pdv_free (uchar_t *ptr)

Convenience routine to free the memory allocated with pdv_alloc.

---

int pdv_is_atmel (PdvDev *pdv_p)

Infers that this device is connected to is an Atmel camera, based on the camera_-class directive.

---

int pdv_is_cameralink (PdvDev *pdv_p)

Infers that this device is connected to is a Camera Link camera (as opposed to RS-422 or LVDS parallel), based on settings from the loaded camera config file.

---

int pdv_is_dvc (PdvDev *pdv_p)

Infers that this device is connected to is a DVC camera, from settings from the loaded camera config file.

---

int pdv_is_hamamatsu (PdvDev *pdv_p)

Infers that this device is connected to is a Hamamatsu camera based on the camera class string.

---

int pdv_is_kodak_i (PdvDev *pdv_p)

Infer if it's a Redlake (formerly Roper, formerly Kodak) 'i' camera from the serial settings.

---

int pdv_is_simulator (PdvDev *pdv_p)

Infers that this device is a simulator – either a PCI DV CLS board, or a PCIe DV C-Link with simulator FPGA loaded.

void pdv_perror (char *err)

Formats and prints a system error.

int pdv_update_values_from_camera (PdvDev *pdv_p)

Deprecated – Queries certain specific cameras via serial, and sets library variables for gain, black level, exposure time and binning to values based on the results of the query.

int ten2one (u_short *wbuf, u_char *bbuf, int count)

Variables

int Pdv_debug

Function Documentation

int dvu_exp_histeq (u_char * **src**, *u_char* * **dst**, *int* **size**, *int* **depth**, *int* **cutoff**)

Perform a histogram equalization on an image, with cutoff (experimental).

Parameters:

src source buffer

dst destination buffer

size size in pixels

depth depth in bits

cutoff histogram cutoff

Definition at line 964 of file libdvu.c.

int dvu_histeq (u_char * **src**, *u_char* * **dst**, *int* **size**, *int* **depth**)

Perform a histogram equalization on an image.

Parameters:

src source buffer

dst destination buffer

size size in pixels

depth depth in bits

Definition at line 800 of file libdvu.c.

*int dvu_write_bmp (char * fname, u_char * buffer, int width, int height)*

Writes an 8-bit per pixel data buffer as a grayscale Windows bitmap file.

Example

```
int err=dvu_write_bmp("file.bmp", buf_p,
          pdv_get_width(pdv_p),
          pdv_get_height(pdv_p)
        );
```

Parameters:

fname filename

buffer data buffer, one byte per pixel

width number of pixels (bytes) per line

height number of lines in the image

Returns:

0 on success, -1 on failure.

See also:

dvu_write_bmp_24

Definition at line 1269 of file libdvu.c.

*int dvu_write_bmp_24 (char * fname, u_char * buffer, int width, int height)*

Writes a 24-bit per pixel RGB data buffer as a Windows bitmap file.

Example

```
int err=dvu_write_bmp("file.bmp", buf_p, pdv_get_width(pdv_p),
pdv_get_height(pdv_p);
```

Parameters:

fname filename

buffer data buffer, one byte per pixel

width number of pixels (bytes) per line

height number of lines in the image

Returns:

0 on success, -1 on failure.

See also:

dvu_write_bmp

Definition at line 1417 of file libdvu.c.

*int dvu_write_image (char * fname, u_char * addr, int x_size, int y_size, int istride)*

Utility routine that outputs a 1-band, 8-bit image to a Sun raster format file (regardless of platform), with stride.

This function can be used to output a partial image. For example, to output the top left 100x100 pixels of the image, specify *x_size* of 100 and *y_size* * 100, and an *istride* of the actual width of the image.

Note:
Use dvu_write_rasfile to output a full image in the most efficient way.

Parameters:

fname the name of the output file

addr the address of the image data (8 bits per pixel)

x_size width in pixels of image

y_size height in pixels of image

istride number of pixels (bytes) to skip from one the beginning of one line to the beginning of the next (this should just be the image width, unless you want something weird like a diagonally skewed image).

Returns:
0 on success, -1 on failure

Definition at line 494 of file libdvu.c.

*int dvu_write_image24 (char * fname, u_char * addr, int x_size, int y_size, int istride)*

Writes a 24-bit per pixel RGB data buffer as a Sun Raster format file, with stride.

Note:
To output a full image in the most efficient way, use dvu_write_rasfile24.

Parameters:

fname the name of output file

addr data buffer, three bytes per pixel (RGB)

x_size number of pixels per line

y_size number of lines in the image

istride number of pixels to skip between sucessive lines

Example

```
// skip every other line
int err=dvu_write_image24("file.ras", buf_p,
           pdv_get_width(pdv_p),
           pdv_get_height(pdv_p) / 2,
           pdv_get_width(pdv_p)
        );
```

Returns:
0 on success, -1 on failure

Definition at line 619 of file libdvu.c.

int dvu_write_rasfile (char * **fname***, u_char* * **addr***, int* **x_size***, int* **y_size***)*

Utility routine that outputs a 1-band, 8-bit image to a Sun raster format file (regardless of platform).

Parameters:
fname the name of the output file

addr the address of the image data (8 bits per pixel)

x_size width in pixels of image

y_size height in pixels of image

Returns:
0 on success, -1 on failure

Definition at line 197 of file libdvu.c.

int dvu_write_rasfile16 (char * **fname***, u_char* * **addr***, int* **x_size***, int* **y_size***, int* **depth_bits***)*

converts 1 band, 10-16 bit image to a sun raster format file and writes to a file.

Parameters:
fname the name of the output file

addr the address of the image data (8 bits per pixel)

x_size width in pixels of image

y_size height in pixels of image

depth_bits number of bits per pixel

Returns:
0 on success, -1 on failure

Definition at line 266 of file libdvu.c.

int dvu_write_rasfile24 (char * **fname***, u_char* * **addr***, int* **x_size***, int* **y_size***)*

Writes a 24-bit per pixel RGB data buffer as a Sun Raster format file.

Parameters:

fname file name

addr data buffer, three bytes per pixel (RGB)

x_size number of pixels per line

y_size number of lines in the image

Returns:

0 on success, -1 on failure

Definition at line 553 of file libdvu.c.

int dvu_write_raw (int **imagesize***, u_char* * **imagebuf***, char* * **fname***)*

Writes a 24-bit per pixel RGB data buffer as a raw data file (no formatting).

Example

```
int err=dvu_write_raw(pdv_get_imagesize(pdv_p), buf_p, "file.raw");
```

Parameters:

imagesize number of bytes in the image.

imagebuf pointer to image data buffer.

fname output file name.

Returns:

0 on success, -1 on failure.

Definition at line 1535 of file libdvu.c.

int pdv_access (char * **fname***, int* **perm***)*

Determines file access independent of operating system.

This a convenience routine that maps to `access()` on Unix/Linux systems, and `_access` on Windows systems.

Parameters:

fname path name of the file to check access of.

perm permission flag(s) to test for. See `access()` (Unix/Linux) or `_access` (Windows) for valid arguments.

Example

```
if (pdv_access("file.ras", F_OK))
    print("Warning: about to overwrite file %s/n", "file.ras");
```

Returns:
0 on success, -1 on failure.

Definition at line 8123 of file libpdv.c.

uchar_t pdv_alloc (int* size*)*

Convenience routine to allocate memory in a system-independent way.

The buffer returned is page aligned. Page alignment is required for some EDT image routines and always preferred. This function uses VirtualAlloc on Windows NT/2000/XP systems, or valloc on Linux/Unix systems.

Example

```
unsigned char *buf = pdv_alloc(pdv_image_size(pdv_p));
```

Parameters:
size the number of bytes of memory to allocate

Returns:
The address of the allocated memory, or NULL on error. If NULL, use pdv_perror to print the error.

See also:
pdv_free

Definition at line 7265 of file libpdv.c.

int pdv_bytes_per_line (int width*, int* depth*)*

Returns bytes per line based on width and bit depth, including depth < 8.

Parameters:
width pixels per line
depth bits per pixel

Returns:
bytes per line

Definition at line 659 of file libpdv.c.

*int pdv_cl_camera_connected (PdvDev * pdv_p)*

Checks whether a camera is connected and turned on.

Looks for an active (changing) pixel clock from the camera, and returns 1 if detected.

Note:
This subroutine only works on EDT Camera Link boards, and only those that have base mode firmware (pdvcamlk or pdvcamlk2 11/02/2006 (rev 34) or later.

Returns:
1 if active pixel clock is detected, 0 if not detected OR not supported (not camera link). Will also return 0 if firmware does not support this feature (see above).

Definition at line 10174 of file libpdv.c.

*void pdv_free (uchar_t * ptr)*

Convenience routine to free the memory allocated with pdv_alloc.

Parameters:
ptr Address of memory buffer to free.

Returns:
void

Definition at line 7279 of file libpdv.c.

*int pdv_is_atmel (PdvDev * pdv_p)*

Infers that this device is connected to is an Atmel camera, based on the camera_class directive.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 if device pdv_p has been setup for Atmel camera, 0 otherwise.

See also:
pdv_get_camera_class

Definition at line 8966 of file libpdv.c.

*int pdv_is_cameralink (PdvDev * pdv_p)*

Infers that this device is connected to is a Camera Link camera (as opposed to RS-422 or LVDS parallel), based on settings from the loaded camera config file.

Generally useful only for applications that may use both Camera Link and the (older) AIA cameras, and that need to differentiate between the two. Specifically for framegrabbers, will return false (0) for simulators.

Parameters:
pdv_p device handle returned by pdv_open

Returns:
1 if Camera Link framegrabber, 0 otherwise

Definition at line 9943 of file libpdv.c.

*int pdv_is_dvc (PdvDev * pdv_p)*

Infers that this device is connected to is a DVC camera, from settings from the loaded camera config file.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 if device pdv_p has been setup for DVC camera, else 0.

Definition at line 9534 of file libpdv.c.

*int pdv_is_hamamatsu (PdvDev * pdv_p)*

Infers that this device is connected to is a Hamamatsu camera based on the camera class string.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 if device pdv_p has been setup for Hamamatsu camera, else 0.

Definition at line 8984 of file libpdv.c.

*int pdv_is_kodak_i (PdvDev * pdv_p)*

Infer if it's a Redlake (formerly Roper, formerly Kodak) 'i' camera from the serial settings.

Since serial commands have changed quite a bit over the years, this subroutine should not be depended on and is only included for backwards compatability.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
1 if pdv_p appears setup for Redlake, 0 otherwise.

Definition at line 8942 of file libpdv.c.

*int pdv_is_simulator (PdvDev * pdv_p)*

Infers that this device is a simulator – either a PCI DV CLS board, or a PCIe DV C-Link with simulator FPGA loaded.

Parameters:
pdv_p device handle returned by pdv_open

Returns:
1 if a simulator, 0 otherwise

Definition at line 9973 of file libpdv.c.

*void pdv_perror (char * err)*

Formats and prints a system error.

Convenience function to format and print a system error. In Linux implementations, the routine just turns around and makes a perror system call, with the errstr argument. NT implementations format and print the last error using GetLastErrorString.

Returns:
void

Definition at line 7209 of file libpdv.c.

*int pdv_update_values_from_camera (PdvDev * pdv_p)*

Deprecated – Queries certain specific cameras via serial, and sets library variables for gain, black level, exposure time and binning to values based on the results of the query.

Included for backwards compatability only. The cameras supported are all older (pre-2000) cameras made by Kodak Megaplus 'i', Hamamatsu, DVC, and Atmel.

This subroutine will be removed in a future release and should not be used.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

Returns:

0 on success, -1 on error (including if the camera is not one that is supported by this subroutine)

Definition at line 9006 of file libpdv.c.

Debug

Get and set flags that determine debug output from the library.

For more information, see the EDT Message Handler Library.

Functions

int pdv_debug (int flag)
Sets the debug level of the PDV library.

---

int pdv_debug_level ()
Gets the debug level, as set by pdv_debug or outside environment variables.

Function Documentation

int pdv_debug (int flag*)*

Sets the debug level of the PDV library.

This results in debug output being written to the screen by PDV library calls. The same thing can be accomplished by setting the PDVDEBUG environment variable to 1. See also the program setdebug.c for information on using the device driver debug flags.

To control the output of messages from the DV library, see the EDT Message Handler Library.

Parameters:
flag flags debug output on (nonzero) or off (zero).

Returns:
previous debug level

Definition at line 6384 of file libpdv.c.

int pdv_debug_level (void)

Gets the debug level, as set by pdv_debug or outside environment variables.

For values, see the EDT Message Handler Library.

Returns:
debug level

Definition at line 6403 of file libpdv.c.

EDT Camera Link Simulator Library

The Camera Link Simulator (CLS) Library provides programming access to the EDT Camera Link Simulator boards, including the PCI DV CLS and PCIe8 DVa CLS.

The source code for the library is in clsim_lib.c and clsim_lib.h.

The following applications are also provided:

pciload: queries EDT boards and provides utilities for verifying and updating board firmware

clsiminit (clsiminit.c): initializes the CLS simulator

simple_clsend (simple_clsend.c): example code for sending an image or images via the simulator

send_tiffs (send_tiffs.c): another example application for sending an image or images via the simulator

clink_tester (clink_tester.c): unit testing between an EDT framegrabber and an EDT simulator

Defines

#define PDV_CLS_DEFAULT_HGAP 300
#define PDV_CLS_DEFAULT_HGAP 300
#define PDV_CLS_DEFAULT_VGAP 400
#define PDV_CLS_DEFAULT_VGAP 400

Functions

int pdv_cls_dep_sanity_check (PdvDev *pdv_p)

Checks for inconsistencies in the configuration (stub).

---

void pdv_cls_dump_geometry (PdvDev *pdv_p)

Prints board geometry only to stdout.

---

void pdv_cls_dump_state (PdvDev *pdv_p)

Prints the board state to stdout.

---

double pdv_cls_frame_time (PdvDev *pdv_p)

Computes and returns the frame time in milliseconds.

---

int pdv_cls_get_hgap (PdvDev *pdv_p)

Computes the horizontal gap value based on the difference between active clocks (hblank) and the total clocks.

---

int pdv_cls_get_vgap (PdvDev *pdv_p)

Computes the vertical gap value based on the difference between active lines(vblank) and the total lines.

---

void pdv_cls_init_serial (PdvDev *pdv_p)

Re-intializes and enables the serial communication.

---

void pdv_cls_set_clock (EdtDev *edt_p, double freq)

Set the clock frequency (MHz).

---

void pdv_cls_set_datacnt (PdvDev *pdv_p, int state)

Enables / disables internal image data generation.

---

int pdv_cls_set_dep (PdvDev *pdv_p)

Initializes simulator values based on PdvDependent structure in `pdv_p`.

---

void pdv_cls_set_depth (PdvDev *pdv_p, int value)
void pdv_cls_set_dvalid (PdvDev *pdv_p, u_char skip, u_char mode)

Set the values for Data Valid (DVAL), timing.

---

void pdv_cls_set_fill (PdvDev *pdv_p, u_char left, u_char right)

Sets the left and right fill values when READVAL is set.

---

void pdv_cls_set_firstfc (PdvDev *pdv_p, int state)

Enables / disables frame count in the first word of each frame.

---

void pdv_cls_set_height (PdvDev *pdv_p, int rasterlines, int vblank)

Set the height of outgoing frames, as well as the number of lines (vgap) between lines.

---

void pdv_cls_set_intlven (PdvDev *pdv_p, int state)

Enables or disables four-tap interleaving.

---

void pdv_cls_set_led (PdvDev *pdv_p, int state)

Controls state of the board's green LED.

---

void pdv_cls_set_line_timing (PdvDev *pdv_p, int width, int taps, int Hfvstart, int Hfvend, int Hlvstart, int Hlvend, int Hrvstart, int Hrvend)

Set the values for frame valid (FVAL), line valid (LVAL), and read valid (RVAL) timing.

---

void pdv_cls_set_linescan (PdvDev *pdv_p, int state)

When set, once the start-of-frame conditions are met, the simulator runs forever, emulating a linescan camera (as if the total vertical active and total vertical count maximum were set to infinity.

---

void pdv_cls_set_lvcont (PdvDev *pdv_p, int state)

Enables / disables line valid timing during vertical blanking.

---

void pdv_cls_set_readvalid (PdvDev *pdv_p, u_short HrvStart, u_short HrvEnd)

Sets the horizontal start and end positions of the ReadValid signal.

---

void pdv_cls_set_rven (PdvDev *pdv_p, int state)

Enables or disables ReadValid Enable (RVEN).

---

void pdv_cls_set_size (PdvDev *pdv_p, int taps, int depth, int width, int height, int hblank, int totalwidth, int vblank, int totalheight)

Set the width and height of the simulator frame.

---

void pdv_cls_set_smallok (PdvDev *pdv_p, int state)

Sets simulator FIFO for small (less than 16KB) images.

---

void pdv_cls_set_trigframe (PdvDev *pdv_p, int state)

Set to enable frame-valid triggering.

---

void pdv_cls_set_trigline (PdvDev *pdv_p, int state)

Set to enable line-valid triggering.

---

void pdv_cls_set_trigpol (PdvDev *pdv_p, int state)

Sets the trigger polariry.

---

void pdv_cls_set_trigsrc (PdvDev *pdv_p, int state)

Selects which input pins to look at for external trigger.

---

void pdv_cls_set_uartloop (PdvDev *pdv_p, int state)

Enables or disables UART looping (echo) of serial data.

---

void pdv_cls_set_width (PdvDev *pdv_p, int width, int hblank)

Set the width of outgoing lines, as well as the number of clocks (hgap) between lines.

void pdv_cls_set_width_lval_rval (PdvDev *pdv_p, int width, int hblank, int hlvstart, int hlvend, int hrvstart, int hrvend)

Set the width of outgoing lines, as well as the number of clocks (hgap) between lines and start and end of line valid and read valid.

void pdv_cls_setup_interleave (PdvDev *pdv_p, short tap0start, short tap0delta, short tap1start, short tap1delta, short tap2start, short tap2delta, short tap3start, short tap3delta)

Sets the start address and delta for each tap.

void pdv_cls_sim_start (PdvDev *pdv_p)

Clears the CFG register including the FIFO_RESET bit (bit 3, 0x08) which clears the fifo and starts the simulator.

void pdv_cls_sim_stop (PdvDev *pdv_p)

Sets the CFG register FIFO_RESET bit (bit 3, 0x08) which stops the simulator.

Function Documentation

*int pdv_cls_dep_sanity_check (PdvDev * pdv_p)*

Checks for inconsistencies in the configuration (stub).

Currently this is a stub. In the future it will return a nonzero error code if a problem is found.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
0 if ok, otherwise error code

See also:
pdv_cls_set_dep

Definition at line 1108 of file clsim_lib.c.

*void pdv_cls_dump_geometry (PdvDev * pdv_p)*

Prints board geometry only to stdout.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 1250 of file clsim_lib.c.

*void pdv_cls_dump_state (PdvDev * pdv_p)*

Prints the board state to stdout.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 1148 of file clsim_lib.c.

*double pdv_cls_frame_time (PdvDev * pdv_p)*

Computes and returns the frame time in milliseconds.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
the computed frame time

Definition at line 1121 of file clsim_lib.c.

*int pdv_cls_get_hgap (PdvDev * pdv_p)*

Computes the horizontal gap value based on the difference between active clocks (hblank) and the total clocks.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
horizontal gap value

Definition at line 153 of file clsim_lib.c.

*int pdv_cls_get_vgap (PdvDev * pdv_p)*

Computes the vertical gap value based on the difference between active lines(vblank) and the total lines.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
vertical gap value

Definition at line 185 of file clsim_lib.c.

*void pdv_cls_init_serial (PdvDev * pdv_p)*

Re-intializes and enables the serial communication.

Rarely used since the serial gets initialized at device open.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Returns:
void

Definition at line 208 of file clsim_lib.c.

*void pdv_cls_set_clock (EdtDev * edt_p, double freq)*

Set the clock frequency (MHz).

On PCI boards, this sets the MPC9230 PLL on PCI CD-CLSIM to 3.5 times the requested pixclk freq. On PCIe DVa boards, sets the SI570 PLL to 1.25x the requeted freq. On PCIe DV boards, sets the SI570 PLL to 1x the requested freq. Valid range is 19.9-85.1. A warning is produced for frequencies outside this range

Parameters:
freq pixel clock frequency (MHz)

Returns:
void

Definition at line 713 of file clsim_lib.c.

*void pdv_cls_set_datacnt (PdvDev * pdv_p, int state)*

Enables / disables internal image data generation.

When enabled, image data comes from the counters instead of the DMA stream.

The simulated 32-bit data generated has a 16-bit count in the LSbs; the 16 MSbs are an inverted version of the LSBs. The count is cleared to zero at the start of each frame. Thus the first 32-bit word of each frame is 0xffff0000, the second is fffe0001, and so on. The CLS treats this data as little-endian, so the foruth 8-bit pixel fo the frame has a value of 0x01. When set, also setting SMALLOK (pdv_cls_set_smallok) stops the simulator at the start of the next frame, to enable getting a single frame of counter data.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

state 1 outputs internally-generated data. When disabled, outputs data from the host via DMA.

See also:
pdv_cls_set_smallok

Returns:
void

Definition at line 406 of file clsim_lib.c.

*int pdv_cls_set_dep (PdvDev * pdv_p)*

Initializes simulator values based on PdvDependent structure in `pdv_p`.

The structure is normally filled in by clsiminit. Assumes bitfile already loaded.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

Definition at line 969 of file clsim_lib.c.

*void pdv_cls_set_dvalid (PdvDev * pdv_p, u_char skip, u_char mode)*

Set the values for Data Valid (DVAL), timing.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

skip how many clocks to skip (ALERT CHECK THIS)

mode mode on or off (ALERT CHECK THIS)

Returns:
void

Definition at line 887 of file clsim_lib.c.

*void pdv_cls_set_fill (PdvDev * pdv_p, u_char left, u_char right)*

Sets the left and right fill values when READVAL is set.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

left the 8 bit left fill value (FillA in CLSIM docs)

right the 8 bit right fill value (FillB in CLSIM docs)

See also:
pdv_cls_set_rven

Returns:
void

Definition at line 900 of file clsim_lib.c.

*void pdv_cls_set_firstfc (PdvDev * pdv_p, int state)*

Enables / disables frame count in the first word of each frame.

When set, the first word of the frame is the frame count: a 16-bit flag of 0x3333 in the MSbs and a 16-bit framecount in the LSbs. It replaces the first 32-bit word of DMA or internally generated data, after any interleaving. When clear, the forst word is the DMA data or generated data per pdv_cls_set_firstfc.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

state 1 enables the first word frame count, 0 disables it.

Returns:
void

Definition at line 380 of file clsim_lib.c.

*void pdv_cls_set_height (PdvDev * pdv_p, int height, int vblank)*

Set the height of outgoing frames, as well as the number of lines (vgap) between lines.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

height number of pixels per line

vgap number of clocks between lines (vertical gap)

Returns:
void

Definition at line 650 of file clsim_lib.c.

*void pdv_cls_set_intlven (PdvDev * pdv_p, int state)*

Enables or disables four-tap interleaving.

When set, enables four-tap interleaving – the four-tap reordering of 8-bit pixel values.

See the CLS Users Guide, Appendix A for a complete description of how data is interleaved. For example, 0x60-61 Tap 0 Start through 0xE-6F Tap 3 Delta. Image data destined for the framegrabber is first passed through an interleaving mechanism to duplicate the data ordering that some cameras exhibit. WHen interleaving is enabled, rasters are restricted to a maximum of 4096 eight-bit pixels of active image data (DMA plus fill).

When clear (default), interleaving is disabled.

To use interleave, first set up the interleave scheme using pdv_cls_setup_interleave.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

enable true to turn on interleave, false to disable it.

See also:

pdv_cls_setup_interleave

Returns:

void

Definition at line 360 of file clsim_lib.c.

*void pdv_cls_set_led (PdvDev * pdv_p, int state)*

Controls state of the board's green LED.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

power_state true (non-zero) to turn on LED, false to turn it off.

Returns:

void

Definition at line 418 of file clsim_lib.c.

*void pdv_cls_set_line_timing (PdvDev * pdv_p, int width, int taps, int Hfvstart, int Hfvend, int Hlvstart, int Hlvend, int Hrvstart, int Hrvend)*

Set the values for frame valid (FVAL), line valid (LVAL), and read valid (RVAL) timing.

In each case, if the end value is 0, the number of clocks required for width is added to the start value (default 0). So if start and end are 0, defaults are start = 0 and end = width/taps.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

taps number of clocks per line

width active pixels per line

Hfvstart

Hfvend

Hlvstart

Hlvend

Hrvstart

Hrvend

Returns:

void

Definition at line 103 of file clsim_lib.c.

void pdv_cls_set_linescan (PdvDev * **pdv_p**, *int* **state**)

When set, once the start-of-frame conditions are met, the simulator runs forever, emulating a linescan camera (as if the total vertical active and total vertical count maximum were set to infinity.

)

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

state 1 enables linescan, 0 disables linescan

Returns:

void

Definition at line 254 of file clsim_lib.c.

void pdv_cls_set_lvcont (PdvDev * **pdv_p**, *int* **state**)

Enables / disables line valid timing during vertical blanking.

When set, line valid is asserted continuously with it's normal timing, even during the vertical blanking interval between frames. When celar, line valid remains low during vertical blanking. Default is unset (0).

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

state 1 enables continuous line valid , 0 disables it

Returns:

void

Definition at line 273 of file clsim_lib.c.

void pdv_cls_set_readvalid (PdvDev * pdv_p, *u_short* HrvStart, *u_short* HrvEnd)

Sets the horizontal start and end positions of the ReadValid signal.

Note that these values have no effect unless RVEN is true.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

HrvStart start postion

HrvEnd end position

See also:

pdv_cls_set_rven.

Definition at line 914 of file clsim_lib.c.

void pdv_cls_set_rven (PdvDev * pdv_p, *int* state)

Enables or disables ReadValid Enable (RVEN).

Read valid is special functionality (not in the Camera Link specification) that allows for outputting an image that's wider than the image data provided. The data outside the image data margins is filled with dummy data values.

When RVEN is set, then the start and end margins of each raster are filled with the values from the FillA and FillB registers respectively, the positions of the margins are determined by HrvStart and HrvEnd. When RVEN is cleared, the entire raster is filled with valid data. HrvStart and HrvEnd can be set with pdv_cls_set_readvalid().

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

enable true to enable ReadValid so data in margins comes from Fill values.

Returns:

void

Definition at line 296 of file clsim_lib.c.

void pdv_cls_set_size (PdvDev * pdv_p, *int* taps, *int* depth, *int* width, *int* height, *int* hblank, *int* totalwidth, *int* vblank, *int* totalheight)

Set the width and height of the simulator frame.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

taps number of clocks per line

depth in bits of data

width width of active data

height number of lines of active data

hblank horizontal blanking between lines

totalwidth total width including horizontal blanking if hblank is zero

vblank horizontal blanking between lines

totalwidth total number of lines including vertical blanking if vblank is zero

There are two ways to set the total width and height including blanking: If hblank is non-zero, the total line width is width + hblank otherwise it is the value passed in in totalwidth. Likewise, if vblank is non-zero, the number of lines between frame valids is height + vblank, otherwise it's the value passed in in totalheight

Returns:
void

Definition at line 36 of file clsim_lib.c.

*void pdv_cls_set_smallok (PdvDev * pdv_p, int state)*

Sets simulator FIFO for small (less than 16KB) images.

When set, simulator starts DMA when 1KB of data is in the FIFO, allowuing the simulator to handle images smaller than 16 KB. When clear, simulator waits until 16 KB of data is in the FIFO before starting DMA. Default for this state is 0 (clear).

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

state 1 enables the state, 0 disables it

Returns:
void

Definition at line 333 of file clsim_lib.c.

*void pdv_cls_set_trigframe (PdvDev * pdv_p, int state)*

Set to enable frame-valid triggering.

Simulator waits at the start of each frame until a trigger is detected.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

state 1 enables, 0 disables

See also:
pdv_cls_set_trigsrc, pdv_cls_set_trigpol, pdv_cls_set_trigline

Returns:
void

Definition at line 471 of file clsim_lib.c.

*void pdv_cls_set_trigline (PdvDev * pdv_p, int state)*

Set to enable line-valid triggering.

Simulator waits at the start of each raster until a trigger is detected. A Dalsa linescan camera starts the next raster when it detects a rising edge on the CC1 line.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
state 1 enables, 0 disables

See also:
pdv_cls_set_trigsrc, pdv_cls_set_trigpol, pdv_cls_set_trigframe

Returns:
void

Definition at line 489 of file clsim_lib.c.

*void pdv_cls_set_trigpol (PdvDev * pdv_p, int polarity)*

Sets the trigger polariry.

A value of 1 sets the trigger polarity to positive TRUE (the default). A value of 0 sets it to negative TRUE.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open
state trigger polarity

See also:
pdv_cls_set_trigsrc, pdv_cls_set_trigframe, pdv_cls_set_trigline

Returns:
void

Definition at line 454 of file clsim_lib.c.

*void pdv_cls_set_trigsrc (PdvDev * pdv_p, int select)*

Selects which input pins to look at for external trigger.

When set, selects camera control line 2 as trigger source. When clear, selects camera control line 1.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

select 1 enables input trigger on CC2, when clear uses CC1

See also:

pdv_cls_set_trigpol, pdv_cls_set_trigframe, pdv_cls_set_trigline

Returns:

void

Definition at line 436 of file clsim_lib.c.

*void pdv_cls_set_uartloop (PdvDev * pdv_p, int state)*

Enables or disables UART looping (echo) of serial data.

When set, serial data emitted by the framegrabber is echoed back unchanged, allowing testing of the framegrabber's serial port.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

state 1 enables uart looping, 0 disables it.

Returns:

void

Definition at line 314 of file clsim_lib.c.

*void pdv_cls_set_width (PdvDev * pdv_p, int width, int hblank)*

Set the width of outgoing lines, as well as the number of clocks (hgap) between lines.

Make sure depth / taps are set correctly first by calling pdv_set_depth (or use pdv_cls_set_size instead of this routine), otherwise the registers won't be set correctly. Also note that this overwrites the horizontal line valid start values with new values based on the width & blanking, and sets readvalid to the full width. Follow this with a call to /ref pdv_cls_set_line_timing if you want to set specific values for those.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

width number of pixels per line

hgap number of clocks between lines (horizontal gap)

See also:
pdv_cls_set_height, pdv_cls_set_line_timing

Returns:
void

Definition at line 539 of file clsim_lib.c.

void pdv_cls_set_width_lval_rval (*PdvDev* * pdv_p, *int* width, *int* hblank, *int* hlvstart, *int* hlvend, *int* hrvstart, *int* hrvend)

Set the width of outgoing lines, as well as the number of clocks (hgap) between lines and start and end of line valid and read valid.

Same as pdv_cls_set_width but includes lval and readval start and end. Make sure depth / taps are set correctly first by calling pdv_set_depth (or use pdv_cls_set_size instead of this routine), otherwise the registers won't be set correctly.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

width number of pixels per line

hgap number of clocks between lines (horizontal gap)

hlvstart number of clocks between lines (horizontal gap)

hlvend number of clocks between lines (horizontal gap)

See also:
pdv_cls_set_width

Returns:
void

Definition at line 598 of file clsim_lib.c.

void pdv_cls_setup_interleave (*PdvDev* * pdv_p, *short* tap0start, *short* tap0delta, *short* tap1start, *short* tap1delta, *short* tap2start, *short* tap2delta, *short* tap3start, *short* tap3delta)

Sets the start address and delta for each tap.

The start address is the 12-bit address of an 8-bit pixel within the 4096 pixel raster. The delta is the amount added to the pixel address with each pixel clock.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

tap0start the start address for tap 0

tap0delta the delta for tap 0

tap1start the start address for tap 1

tap1delta the delta for tap 1

tap2start the start address for tap 2

tap2delta the delta for tap 2

tap3start the start address for tap 3

tap3delta the delta for tap 3

Returns:

void

Definition at line 940 of file clsim_lib.c.

*void pdv_cls_sim_start (PdvDev * pdv_p)*

Clears the CFG register including the FIFO_RESET bit (bit 3, 0x08) which clears the fifo and starts the simulator.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

Returns:

void

Definition at line 504 of file clsim_lib.c.

*void pdv_cls_sim_stop (PdvDev * pdv_p)*

Sets the CFG register FIFO_RESET bit (bit 3, 0x08) which stops the simulator.

Parameters:

pdv_p pointer to pdv device structure returned by pdv_open

Returns:

void

Definition at line 518 of file clsim_lib.c.

EDT Message Handler Library

Provides generalized error- and message-handling for the edt and pdv libraries.

These routines provide a way for application programs to intercept and handle edtlib and pdvlib error, warning, and debug messages, but you can also use them for application messages.

By default, output goes to the console (stdout), but you can substitute user-defined functions – for example, a function that pops up a window to display text. You can set different message levels for different output, and multiple message handles can exist within an application, with different message handlers associated with them.

Predefined message flags are described in the "Defines" section of this document. Those starting with `EDTAPP_MSG_` are for general application use, those starting with `EDTLIB_MSG_` are for libedt messages, and those beginning with `PDVLIB_MSG_` are for libpdv messages. Application programmers can define other flags in the 0x1000 to 0x1000000 range.

Message levels are defined by flag bits, and each bit can be set or cleared individually. So, for example, to have a message-handler called only for fatal and warning application messages, 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 ordinarily 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*. For compatibility with possible future changes, do not access structure elements directly; instead always use the error subroutines.

Data Structures

struct _edt_msg_handler

Structure used by the Message Handler Library to control the output of messages.

Defines

#define edt_msg_add_default_level(addlevel) edt_msg_set_level(edt_msg_default_handle(), edt_msg_default_level() | addlevel)
#define EDT_MSG_ALWAYS 0x80000000
#define EDT_MSG_FATAL EDTAPP_MSG_FATAL | EDTLIB_MSG_FATAL | PDVLIB_MSG_FATAL
#define EDT_MSG_INFO_1 EDTAPP_MSG_INFO_1 | EDTLIB_MSG_INFO_1 | PDVLIB_MSG_INFO_1
#define EDT_MSG_INFO_2 EDTAPP_MSG_INFO_2 | EDTLIB_MSG_INFO_2 | PDVLIB_MSG_INFO_2
#define EDT_MSG_WARNING EDTAPP_MSG_WARNING | EDTLIB_MSG_WARNING | PDVLIB_MSG_WARNING
#define EDTAPP_MSG_FATAL 0x1

Fatal-error messages in applications.

#define EDTAPP_MSG_INFO_1 0x4

First level info messages in applications.

#define EDTAPP_MSG_INFO_2 0x8

Second level info messages in applications.

#define EDTAPP_MSG_WARNING 0x2

Warning messages in applications.

#define EDTLIB_MSG_FATAL 0x10

Fatal-error messages in libedt.

#define EDTLIB_MSG_INFO_1 0x40

Informative messages in libedt.

#define EDTLIB_MSG_INFO_2 0x80

Debugging messages in libedt.

#define EDTLIB_MSG_WARNING 0x20

Warning messages in libedt.

#define PDVLIB_MSG_FATAL 0x100

Fatal-error messages in libpdv.

#define PDVLIB_MSG_INFO_1 0x400

Informative messages in libpdv.

#define PDVLIB_MSG_INFO_2 0x800

Debugging messages in libpdv.

#define PDVLIB_MSG_WARNING 0x200

Warning messages in libpdv.

Typedefs

typedef int(∗) EdtMsgFunction (void ∗target, int level, const char ∗message)

An EdtMsgFunction is a function which outputs a message if that message's level is high enough.

---

typedef _edt_msg_handler EdtMsgHandler

Structure used by the Message Handler Library to control the output of messages.

Functions

int edt_get_verbosity (void)

int edt_msg (int level, const char ∗format,...)

Submits a message to the default message handler, which will conditionally (based on the flag bits) send the message to the default message handler function.

---

void edt_msg_add_level (EdtMsgHandler ∗msg_p, int level)

Sets the message level to the combination of the specified level with the message handler's previous level.

---

void edt_msg_close (EdtMsgHandler ∗msg_p)

Closes and frees up memory associated with a message handler.

EdtMsgHandler ∗ edt_msg_default_handle (void)

Gets the default message handler.

---

int edt_msg_default_level (void)

Gets the message level that messages must match in order to be handled by the default message handler.

---

int edt_msg_get_level (EdtMsgHandler ∗msg_p)

Gets the message level that messages must match in order to be handled by the message handler msg_p*.*

---

void edt_msg_init (EdtMsgHandler *msg_p)

Initializes a message handler with default values.

---

void edt_msg_init_files (EdtMsgHandler *msg_p, FILE *file, int level)

Initializes a message handler to use the specified file and level.

---

void edt_msg_init_names (EdtMsgHandler *msg_p, char *file, int level)

Initializes a message handler to use the named file and specified level.

---

char * edt_msg_last_error (void)

Gets the message last sent to the output by the edt message handling system.

---

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

Submits a message using the msg_p *message handler, which will conditionally (based on the flag bits) send the message to the handler's function.*

---

int edt_msg_output_perror (EdtMsgHandler *msg_p, int level, const char *message)

Conditionally (based on the flag bits) outputs `message`*, followed by the last system error message, to* `msg_p`*.*

---

int edt_msg_output_printf_perror (EdtMsgHandler *msg_p, int level, const char *format,...)

Writes to the specified EdtMsgHandler a caller-specified message (in the printf-style format) followed by the last system error message.

---

int edt_msg_perror (int level, const char *msg)

Conditionally outputs a system perror using the default message handler.

---

int edt_msg_printf_perror (int level, const char *format,...)

Outputs a caller-specified message to the output, followed by the last system error message.

---

void edt_msg_set_file (EdtMsgHandler *msg_p, FILE *f)

Sets the output file pointer for the message handler.

---

void edt_msg_set_function (EdtMsgHandler *msg_p, EdtMsgFunction f)

Sets the function to call when a message event occurs.

---

void edt_msg_set_level (EdtMsgHandler *msg_p, int newlevel)

Sets the "message level" flag bits that determine whether to call the message handler for a given message.

---

void edt_msg_set_name (EdtMsgHandler *msg_p, const char *f)

Sets the output file to the named file.

---

void edt_msg_set_target (EdtMsgHandler *msg_p, void *t)

Sets the target in the message handler.

---

void edt_set_verbosity (int verbose)
int lvl_printf (int delta, char *format,...)

Typedef Documentation

*typedef int(*) EdtMsgFunction(void *target, int level, const char *message)*

An EdtMsgFunction is a function which outputs a message if that message's level is high enough.

Parameters:

target this stores extra info useful to the specific function defined. In the default message handler setup by edt_msg_init, the function used expects target to be a FILE pointer.

level The message level associated with with the message.

message The message which can be output by the function.

Definition at line 83 of file edt_error.h.

Function Documentation

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

Submits a message to the default message handler, which will conditionally (based on the flag bits) send the message to the default message handler function.

This function uses the default message handler, and is equivalent to calling edt_msg_output(edt_msg_default_handle(), ...). To submit a message for handling by other than the default message handle, use edt_msg_output.

Parameters:

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.

Example

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

Returns:

0 on success, -1 on failure.

Definition at line 279 of file edt_error.c.

void edt_msg_add_level (EdtMsgHandler * **msg_p**, *int* **level**)

Sets the message level to the combination of the specified level with the message handler's previous level.

Parameters:

msg_p pointer to message handler, initiailzed by edt_msg_init

level a message level flag, as defined in the overview.

Definition at line 421 of file edt_error.c.

void edt_msg_close (EdtMsgHandler * **msg_p**)

Closes and frees up memory associated with a message handler.

Use only on message handlers that have been explicitly initialized by edt_msg_init. Do not try to close the default message handler. If the message handler has been configured to use a file which the user opened, through functions such as edt_msg_init_files or edt_msg_set_file, then the user is responsible for closing that file after calling this function.

Parameters:

msg_p pointer to message handler to close, which was initiailzed by edt_msg_init

Returns:

0 on success, -1 on failure.

Definition at line 246 of file edt_error.c.

EdtMsgHandler* edt_msg_default_handle (void)

Gets the default message handler.

This is useful if you want to modify the default handler's behaviour, with functions such as edt_msg_set_level, edt_msg_set_function, edt_msg_set_file, edt_msg_set_name, or edt_msg_set_target.

Definition at line 716 of file edt_error.c.

int edt_msg_default_level (void)

Gets the message level that messages must match in order to be handled by the default message handler.

The level is a combination of flags OR'ed together as described in the overview.

The equivalent function for a user defined message handler is edt_msg_get_level.

Definition at line 733 of file edt_error.c.

int edt_msg_get_level (EdtMsgHandler * msg_p)

Gets the message level that messages must match in order to be handled by the message handler *msg_p*.

The level is a combination of flags OR'ed together as described in the overview.

Parameters:

msg_p pointer to message handler

Definition at line 407 of file edt_error.c.

void edt_msg_init (EdtMsgHandler * msg_p)

Initializes a message handler with default values.

The message file is initialized to `stderr`. The output subroutine pointer is set to `fprintf` (to write output to the console). The message level is set to EDT_MSG_WARNING | EDT_MSG_FATAL.

Parameters:

msg_p pointer to message handler structure to initialize

Example

```
EdtMsgHandler msgLogger;
edt_msg_init(&msgLogger);
```

See also:

edt_msg_output

Definition at line 174 of file edt_error.c.

void edt_msg_init_files (EdtMsgHandler * **msg_p**, *FILE* * **file**, *int* **level**)

Initializes a message handler to use the specified file and level.

Similar to edt_msg_init_names but rather than opening a named file, this takes a pointer to a FILE which has been opened by the caller.

Parameters:

msg_p pointer to message handler structure to initialize

file FILE pointer returned by e.g. fopen().

level the level that future messages must match against if they are to be handled by the *msg_p* handler.

Definition at line 221 of file edt_error.c.

void edt_msg_init_names (EdtMsgHandler * **msg_p**, *char* * **file**, *int* **level**)

Initializes a message handler to use the named file and specified level.

Parameters:

msg_p pointer to message handler structure to initialize

file the name of a file to open and write messages to.

level the level that future messages must match against if they are to be handled by the *msg_p* handler.

Definition at line 201 of file edt_error.c.

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

Submits a message using the *msg_p* message handler, which will conditionally (based on the flag bits) send the message to the handler's function.

To submit a message to the default message handler, use edt_msg.

Parameters:

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.

Example

```
int my_error_popup(void *target, int level, char *message) {
    GtkWindow * parentWindow = (GtkWindow *)target;
    GtkWidget * dialog = gtk_message_dialog_new(parentWindow, 0,
    GTK_MESSAGE_WARNING, GTK_BUTTONS_NONE, message);
}

if (edt_access(fname, 0) != 0)
     edt_msg_output(msgLogger, EDTAPP_MSG_WARNING, "file '%s' not
     found", fname);

GtkWindow *window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_show(window);

EdtMsgHandler msgLogger;
edt_msg_init(&msgLogger);
edt_msg_set_target(window);
edt_msg_set_function(msgLogger, (EdtMsgFunction *)my_error_popup);
edt_msg_set_level(msgLogger, EDT_MSG_FATAL | EDT_MSG_WARNING);
```

Returns:
0 on success, -1 on failure.

Definition at line 344 of file edt_error.c.

*int edt_msg_output_perror (EdtMsgHandler * msg_p, int level, const char * msg)*

Conditionally (based on the flag bits) outputs `message`, followed by the last system error message, to `msg_p`.

To output to the default message handler, use edt_msg_perror.

Parameters:
msg_p pointer to message handler, initiailzed by edt_msg_init

level message level for the current message, as described in the overview

msg message to concatenate to the system error message

See also:
edt_perror

Definition at line 606 of file edt_error.c.

*int edt_msg_output_printf_perror (EdtMsgHandler * msg_p, int level, const char * format, ...)*

Writes to the specified EdtMsgHandler a caller-specified message (in the printf-style format) followed by the last system error message.

If you want to just use the default handler (to just print to the console), use edt_msg_printf_perror instead.

Parameters:

msg_p pointer to message handler, initiailzed by edt_msg_init

level the EDT Message level. This function will only output the message if level is greater than or equal to that set by edt_msg_init, edt_msg_set_level, edt_msg_add_level, or edt_msg_add_default_level.

format a printf() style format string. Like printf(), it should be followed by arguments to match the format.

See also:

edt_msg_printf_perror for an example

Definition at line 699 of file edt_error.c.

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

Conditionally outputs a system perror using the default message handler.

This function is equivalent to calling edt_msg_output_perror(edt_msg_default_handle(), level, msg);

Parameters:

level message level for the current message, as described in the overview

msg message to concatenate to the system error message

Example

```
if ((fp = fopen ("file.txt", "r")) == NULL)
    edt_msg_perror(EDT_FATAL, "couldn't open file.txt for reading");
```

Returns:

0 on success, -1 on failure.

See also:

edt_perror, edt_msg_output_perror

Definition at line 554 of file edt_error.c.

int edt_msg_printf_perror (int level*, const char* * format*, ...)*

Outputs a caller-specified message to the output, followed by the last system error message.

This is useful when an error occurs, and you want your error message to be followed by the system's error message.

Example:

```
char *file_name = "/aFileThatDoesNotExist";
FILE *file_ptr = fopen(file_name, "r");
if (file_ptr == NULL) {
    edt_msg_printf_perror(
          EDTAPP_MSG_FATAL,
          "Couldn't open file '%s'",
          file_name);
    exit(1);
}
```

Which will print something like "Couldn't open file '/aFileThatDoesNotExist': No such file or directory"

Parameters:

level the EDT Message level. This function will only output the message if level is greater than or equal to that set by edt_msg_init, edt_msg_set_level, edt_msg_add_level, or edt_msg_add_default_level.

format a printf() style format string. Like printf(), it should be followed by arguments to match the format.

Returns:

0 on success, -1 on failure.

Definition at line 673 of file edt_error.c.

void edt_msg_set_file (EdtMsgHandler * **msg_p**, *FILE* * **fp**)

Sets the output file pointer for the message handler.

The user still owns the file, so they are responsible for closing it after this message handler is done with it, such as after this function is called again, or after edt_msg_close is called.

Parameters:

msg_p pointer to message handler

fp FILE pointer to an opened file, to which the messages should be output.

Example

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

... some time later ...

edt_msg_close(msg_p);
fclose(fp);
```

Definition at line 477 of file edt_error.c.

*void edt_msg_set_function (EdtMsgHandler * msg_p, EdtMsgFunction f)*

Sets the function to call when a message event occurs.

The default message function is `fprintf()` (which outputs to `stderr`); this routine allows programmers to substitute any type of message handler (pop-up callback, file write, etc).

For an example of how this could be used, see edt_msg.

Parameters:

msg_p pointer to message handler

f The funtion to call when a message event occurs.

See also:

edt_msg_set_level, edt_msg

Definition at line 441 of file edt_error.c.

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

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.

Parameters:

msg_p pointer to message handler

newlevel The new level to set in the message handler.

Example

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

Definition at line 389 of file edt_error.c.

*void edt_msg_set_name (EdtMsgHandler * msg_p, const char * name)*

Sets the output file to the named file.

Parameters:

msg_p pointer to message handler

name the name of a file to open. Future messages will be written to that file.

Definition at line 499 of file edt_error.c.

*void edt_msg_set_target (EdtMsgHandler * msg_p, void * target)*

Sets the target in the message handler.

The target would usually be an object that messages are sent to, such as a window, but exactly what it will be depends on what the message handler's function expects.

See also:
edt_msg_set_function, EdtMsgFunction

Definition at line 524 of file edt_error.c.

OCM/OC192 Library

The functions in lib_ocm.c are meant to simplify the sometimes confusing task of setting up and resetting DMA channels on the OCM and OC192 mezzanine boards (and future boards which also do framed SONET input).

The goal is to be able to say "I want data at this rate on this channel" without worrying about the details; if that isn't possible there will be an error return.

There are two board-specific sets of functions (edt_ocm_xxx, edt_oc192_xxx) and a general set of edt_ocx_xxx. If a particular task is identical for both mezzanine cards, there will only be the edt_ocx_xxx version. Otherwise, the edt_ocx_xxx function will either call other edt_ocx_xxx functions or will call the board-specific version.

The initialization functions are separated into the following sequence of stages. It is possible to return to functions in the sequence for full or partial reinitialization a channel. At each stage in the sequence there will be different diagnostic functions available.

The definition of the target channel state is carried in the EdtOCConfig structure passed as a pointer to most of the library functions. This includes the target line rate, framing parameters, and any non-default bitfiles desired. The baseboard and channel are associated with the EdtDev * pointer passed to all of the functions.

Stage 1: edt_ocx_base_init()

First, to start with the base board in an unknown state, call edt_ocx_base_init. This will make sure that at least a default baseboard and mezzanine bitfile(s) are loaded so the mezzanine board can be identified, and the PLLs between baseboard and mezzanine are in synch (checking both SYS_LOCK and LOCAL_SYS_LOCK).

This function will abort any dma on the other channel on the OCM card.

At this point the mezzanine card can be identified, and the SFP or XFP modules can be queried for their status.

Normally this function need only be called once after poweron, unless a different baseboard interface bitfile is requested.

Stage 2: edt_ocx_channel_set_rate()

The rate setting step makes sure that the correct mezzanine bitfile is loaded for the target line rate, and the correct clock source is selected. If an improper rate for the channel is requested, there will be a non-zero error return.

Stage 3: edt_ocx_channel_setup()

This sets the framing parameters, descrambling, enables memory, etc.

Stage 4: edt_ocx_channel_lock_frontend()

This starts the framer and resets the frontend PLLs. The channel fifo is flushed. Failure to see the SIG_DET bit or if the LOL bit is set will cause a non-zero error return.

At this point framing errors can be checked by calling edt_ocx_get_framing_errors().

Stage 5: edt_ocx_channel_start()

This assumes that ring-buffers have been configured. It starts the ring-buffer acquisition, then turns on the channel enable bit to start DMA. If framing is enabled, it will wait for frame and return an error if framing times out.

Steps 1 through 4 can be executed at once using the function edt_ocx_configure(), which will run each step and return an error code if any step fails for some reason.

Modules

OCM Mezzanine Access Functions

Setup and diagnostic functions specific to OCM mezzanine channels.

OC192 Mezzanine Access Functions

Setup and diagnostic functions specific to OC192 mezzanine channels.

OC192 LIU Access Functions

OC192 Mezzanine LIU Serial Access Functions The oc192_mdio functions are for reading and writing the LIU chip through its serial protocol.

OCM Mezzanine Access Functions

OC192 Mezzanine Access Functions

OC192 LIU Access Functions

IRIG-B Timecode Library

The functions in libedt_timing.c and libedt_timing.h provide services for the IRIG-B Timecode package running on an EDT I/O board.

The services include:

Functions to acquire and display the IRIG-B timecode from the embedded MSP430 timecode processor. Normally the timecode will be embedded in the DMA stream the EDT board firmware. Contact EDT for more information.

Functions to control the configuration of the embedded MSP430 timecode processor.

Functions to load updated firmware to the MSP430 timecode processor boot flash prom.

Modules

Configuration Functions

Configuration Functions.

Display Functions

Display Functions.

Firmware Update Functions

Firmware Update Functions.

Configuration Functions

Display Functions

Firmware Update Functions

SDH to E1 Firmware Demultiplex Library

The services include:

Board initialization loads base and mezzanine bitfiles and returns a handle for use with the following functions.

DMA channel setup with user function callback registration to dispose of demultiplexed E1 packets. Another function cancels this e1 processing data stream.

Functions to enable and disable demultiplexing on a selected STM1 data channel.

To be implemented:

Access to diagnostic status concerning G.707 configuration and data pathways, pointer processing, framing status, loss of light, and DMA access to upstream data prior to demultiplexing.

prbs checking for demultiplexed E1 packets as well as upstream DMA data sources.

EDT Time Library

EDT Time software functions include setting the board time to system time as UNIX time (seconds since January 1, 1970), retrieving the 64-bit time value, and adjusting for the errors between system time and EDT Time.

The clock on the EDT board can be adjusted to compensate for the drift between board time and system time, as well as adjusted to converge back to the desired system time without time values ever decreasing. Also, functions are provided to create a monitoring thread that periodically samples the error between EDT time and system time, then adjusts the board time accordingly.

EDT Time starts automatically as soon as the FPGA configuration file is loaded. The following table summarizs the most useful time functions:

Purpose	Function
To set the time to current system time	edt_sstm_set_to_sys
To retrieve the current time	edt_sstm_timestamp
To get the current error between EDT time and system time	edt_sstm_measure_drift
To measure the drift between EDT Time and system time	edt_sstm_sys_error
To calculate the current error and revert to system time gradually	edt_sstm_iterate_adjust
To create and start an adjustment thread	edt_sstm_launch_adjuster

Note:
It doesn't matter which channel an application opens, as there's only one clock per board.

Below is a simple example to set the board time, then launch an adjustment thread that samples every five minutes:

```
edt_p = edt_open(EDT_INTERFACE, unit);
edt_sstm_set_to_sys(edt_p);
adjuster = edt_sstm_launch_adjuster(edt_p,
                300, // check every 5 minutes
                20,  // # of adjustment_scalar of adjustment as error gets smaller
                10,  // each iteration should take 10 secs.
                200, // maximum 200 microsecond error allowed
                20,  // try to get within 20 microseconds
                0    // loop indefinitely
                );
// for this example just go to sleep
while (1)
    edt_msleep(300000);
```

The sample program provided, *edt_ss_time.c*, implements the above code. To run it, enter:

```
edt_ss_time -T -L 300 20 200
```

It also exercises the other EDT Time functions.

Functions

void edt_sstm_adjuster_start (EdtTimeController *tm)

Start an adjuster thread.

---

void edt_sstm_adjuster_stop (EdtTimeController *tm)

Stop an adjuster thread.

---

void edt_sstm_disable_adjust (EdtTimeController *tm)

Turn off rate adjustment.

---

void edt_sstm_enable_adjust (EdtTimeController *tm)

Turn on rate adjustment.

---

int edt_sstm_get_adj_sample_secs ()

Get the current value of adj_sample_seconds.

---

int edt_sstm_get_adj_samples ()

Get the current value of adj_samples.

---

int edt_sstm_get_adjust_enabled (EdtTimeController *tm)

Returns 0 or 1 depending on EDT_SSTM_ADJ_EN bit.

---

int edt_sstm_get_adjust_sign (EdtTimeController *tm)

Returns -1 or 1 depending on EDT_SSTM_ADJ_PLUS bit.

---

u_int edt_sstm_get_adjust_ticks (EdtTimeController *tm)

Returns the signed value of the adjustment.

---

u_int edt_sstm_get_counts (EdtTimeController *tm)

Returns the # of ticks = (1<<20)/1000000 microseconds.

---

u_int edt_sstm_get_seconds (EdtTimeController *tm)

Returns the current value of seconds.

---

void edt_sstm_get_time_parts (EdtTimeController *tm, u_int *seconds, u_int *usecs)

Gets both integer parts (secs/usecs) of time values.

---

u_int edt_sstm_get_usecs (EdtTimeController *tm)

Returns the current # of usecs as an unsigned int.

---

void edt_sstm_latch_time (EdtTimeController *tm)

Latches the current time into registers.

EdtTimeController * edt_sstm_launch_adjuster (EdtTimeController *tm)

Start a thread to check and correct time against system time.

---

double edt_sstm_measure_drift (EdtTimeController *tm)

Calculate basic error rate between SS clock and sys clock.

---

void edt_sstm_set (EdtTimeController *tm, unsigned int second)

Set the current seconds value to second + 1, clears usecs, synched to system time.

---

void edt_sstm_set_adj_from_drift (EdtTimeController *tm, double drift)

Uses a drift value in usecs/sec to set the adjustment value on tm.

---

void edt_sstm_set_adj_sign (EdtTimeController *tm, int positive)

Sets the sign bit for rate adjustment.

---

void edt_sstm_set_adj_ticks (EdtTimeController *tm, int ticks, int positive)

Sets the time adjustment to tick counts between an adjustment.

---

void edt_sstm_set_drift_sampling (int seconds, int samples)

Sets the parameters used to measure drift.

---

void edt_sstm_set_secs (EdtTimeController *tm, unsigned int second)

Set the current seconds value, clears usecs.

---

void edt_sstm_set_to_sys (EdtTimeController *tm)

Sets the time to the current system time, by waiting for zero crossing, then half a second, then calling edt_sstm_set.

---

void edt_sstm_set_to_sys_error (EdtTimeController *tm, int error)

Sets the time to the current system time + an error in milliseconds.

void edt_sstm_setup (EdtTimeController *tm, char *bitfile)

Set the EDT timer - load the desired bitfile if necessary.

void edt_sstm_strobe (EdtTimeController *tm, unsigned int bits)

Execute strobed command in bits for EDT timer.

double edt_sstm_sys_error (EdtTimeController *tm)

Return the mean error between EDT time and sys time as a double (in seconds).

int edt_sstm_ticks_from_drift (double drift)

Compute the adjustment ticks from drift value in ppm.

double edt_sstm_timestamp (EdtTimeController *tm)

Returns EDT time as double - seconds and microseconds.

Function Documentation

*void edt_sstm_adjuster_start (EdtTimeController * tm)*

Start an adjuster thread.

Starts a thread running with adjuster tm, by setting active to 1 and launching a new thread.

Parameters:

tm The adjuster structure originally created by edt_sstm_launch_adjuster.

Definition at line 1120 of file ss_time_lib.c.

*void edt_sstm_adjuster_stop (EdtTimeController * tm)*

Stop an adjuster thread.

Stops the thread running with adjuster tm, by setting active to 0 and waiting until done goes true.

Parameters:

tm The adjuster structure originally created by edt_sstm_launch_adjuster.

Definition at line 1100 of file ss_time_lib.c.

*void edt_sstm_disable_adjust (EdtTimeController * tm)*

Turn off rate adjustment.

Parameters:
tm The device handle for the SS/GS board.

Definition at line 593 of file ss_time_lib.c.

*void edt_sstm_enable_adjust (EdtTimeController * tm)*

Turn on rate adjustment.

Parameters:
tm The device handle for the SS/GS board.

Definition at line 605 of file ss_time_lib.c.

int edt_sstm_get_adj_sample_secs ()

Get the current value of adj_sample_seconds.

adj_sample_seconds is the total time sampled by the drift measure routine. Set using edt_sstm_set_drift_sampling.

Returns:
the current value of adj_samples;

Definition at line 804 of file ss_time_lib.c.

int edt_sstm_get_adj_samples ()

Get the current value of adj_samples.

adj_samples is the number of samples used to compute drift Set using edt_sstm_set_drift_sampling.

Returns:
the current value of adj_samples;

Definition at line 818 of file ss_time_lib.c.

*int edt_sstm_get_adjust_enabled (EdtTimeController * tm)*

Returns 0 or 1 depending on EDT_SSTM_ADJ_EN bit.

Parameters:
tm The device handle for the SS/GS board.

Returns:
0 if adjustment not enabled, 1 if it is.

Definition at line 367 of file ss_time_lib.c.

*int edt_sstm_get_adjust_sign (EdtTimeController * tm)*

Returns -1 or 1 depending on EDT_SSTM_ADJ_PLUS bit.

Parameters:
tm The device handle for the SS/GS board.

Returns:
-1 if positive adjustment not enabled, 1 if it is.

Definition at line 386 of file ss_time_lib.c.

*u_int edt_sstm_get_adjust_ticks (EdtTimeController * tm)*

Returns the signed value of the adjustment.

Parameters:
tm The device handle for the SS/GS board.

Definition at line 403 of file ss_time_lib.c.

*u_int edt_sstm_get_counts (EdtTimeController * tm)*

Returns the # of ticks = (1<<20)/1000000 microseconds.

Parameters:
tm The device handle for the SS/GS board.

Definition at line 429 of file ss_time_lib.c.

*u_int edt_sstm_get_seconds (EdtTimeController * tm)*

Returns the current value of seconds.

Parameters:
tm The device handle for the SS/GS board.

Definition at line 417 of file ss_time_lib.c.

*void edt_sstm_get_time_parts (EdtTimeController * tm, u_int * seconds, u_int * usecs)*

Gets both integer parts (secs/usecs) of time values.

This routine latches the current time, then returns the two 32 bit integers into the pointers passed in.

Parameters:
tm The device handle for the SS/GS board.
seconds Pointer to value returned for seconds.
usecs Pointer to value returned for microseconds.

Definition at line 467 of file ss_time_lib.c.

*u_int edt_sstm_get_usecs (EdtTimeController * tm)*

Returns the current # of usecs as an unsigned int.

Parameters:
tm The device handle for the SS/GS board.

Returns:
The value calculated by multiplying the counts register by (1<<20)/1000000).

Definition at line 446 of file ss_time_lib.c.

*void edt_sstm_latch_time (EdtTimeController * tm)*

Latches the current time into registers.

Parameters:
tm The device handle for the SS/GS board.

Definition at line 355 of file ss_time_lib.c.

EdtTimeController edt_sstm_launch_adjuster (EdtTimeController * tm)*

Start a thread to check and correct time against system time.

Parameters:
tm The EdtTimeController

Returns:
A pointer to the EdtTimeController structure.

Definition at line 1073 of file ss_time_lib.c.

*double edt_sstm_measure_drift (EdtTimeController * tm)*

Calculate basic error rate between SS clock and sys clock.

Take mean of *adj_samples* over *sample_seconds*

Parameters:
tm The device handle for the SS/GS board.

Returns:
Measured drift in usecs/sec.

Definition at line 875 of file ss_time_lib.c.

void edt_sstm_set (EdtTimeController * **tm**, *unsigned int* **second**)

Set the current seconds value to second + 1, clears usecs, synched to system time.

Calls edt_wait_for_zero before strobing value.

Parameters:

tm The device handle for the SS/GS board.

second The value for the seconds counter. Note that this ends up incremented by one, because of the wait for zero crossing in system time.

Definition at line 521 of file ss_time_lib.c.

void edt_sstm_set_adj_from_drift (EdtTimeController * **tm**, *double* **drift**)

Uses a drift value in usecs/sec to set the adjustment value on tm.

Parameters:

tm The device handle for the SS/GS board.

drift The drift in usecs/sec for which to compensate.

Definition at line 855 of file ss_time_lib.c.

void edt_sstm_set_adj_sign (EdtTimeController * **tm**, *int* **positive**)

Sets the sign bit for rate adjustment.

Parameters:

tm The device handle for the SS/GS board.

positive Set to 1 for positive adjustment, 0 for negative.

Definition at line 618 of file ss_time_lib.c.

void edt_sstm_set_adj_ticks (EdtTimeController * **tm**, *int* **ticks**, *int* **positive**)

Sets the time adjustment to tick counts between an adjustment.

Positive indicates whether change is positive or negative. You can't use the sign of ticks itself because -0 is very different from 0.

Parameters:

tm The device handle for the SS/GS board.

ticks The number of adjustment ticks to set.

positive Whether the change is positive or negative

Definition at line 652 of file ss_time_lib.c.

void edt_sstm_set_drift_sampling (int seconds*, int* samples*)*

Sets the parameters used to measure drift.

Parameters:

seconds How many seconds to measure in total

samples How many samples to take over the time set by seconds

Definition at line 789 of file ss_time_lib.c.

void edt_sstm_set_secs (EdtTimeController * tm*, unsigned int* second*)*

Set the current seconds value, clears usecs.

This isn't in synch with system time - to do so use edt_sstm_set instead, which waits for system zero crossing.

Parameters:

tm The device handle for the SS/GS board.

second The value for the seconds counter

Definition at line 504 of file ss_time_lib.c.

void edt_sstm_set_to_sys (EdtTimeController * tm*)*

Sets the time to the current system time, by waiting for zero crossing, then half a second, then calling edt_sstm_set.

Parameters:

tm The device handle for the SS/GS board.

Definition at line 538 of file ss_time_lib.c.

void edt_sstm_set_to_sys_error (EdtTimeController * tm*, int* error*)*

Sets the time to the current system time + an error in milliseconds.

Waits for zero crossing, then half a second, then calling edt_sstm_set. Attempts to add error milliseconds to time.

Parameters:

tm The device handle for the SS/GS board.

error Signed error in milliseconds.

Definition at line 557 of file ss_time_lib.c.

*void edt_sstm_setup (EdtTimeController * tm, char * bitfile)*

Set the EDT timer - load the desired bitfile if necessary.

Parameters:

tm The device handle for the SS/GS board.

bitfile Name of an optional bitfile. Null uses default "c3_demux.bit".

Definition at line 263 of file ss_time_lib.c.

*void edt_sstm_strobe (EdtTimeController * tm, unsigned int bits)*

Execute strobed command in bits for EDT timer.

Commands to the EDT timer are passed by strobing in to register 8f (tm->cmd). Possible values are:

```
EDT_SSTM_COPY     - This copies the value of register tm->set to timer
EDT_SSTM_LATCH    - This latches the current counter values into the EDT_SSTM_TIME registers
EDT_SSTM_COPY_ADJ - This copies the value of register tm->set to timer adjust register
```

The routine uses the top bit (7) as a "lock" to minimize contention

Bits 4, 5, and 6 are preserved.

Parameters:

tm The device handle for the SS/GS board.

bits Which bit to strobe.

Definition at line 292 of file ss_time_lib.c.

*double edt_sstm_sys_error (EdtTimeController * tm)*

Return the mean error between EDT time and sys time as a double (in seconds).

The error is measured by getting the EDT time before and after the system time, then averages the difference.

Parameters:

tm The device handle for the SS/GS board.

Returns:

The difference in seconds between EDT time and system time, precise to microseconds.

Definition at line 679 of file ss_time_lib.c.

int edt_sstm_ticks_from_drift (double drift*)*

Compute the adjustment ticks from drift value in ppm.

Parameters:
drift The drift value in ppm or usecs/sec to correct.

Returns:
the integer value to set the adjustment.

Definition at line 832 of file ss_time_lib.c.

double edt_sstm_timestamp (EdtTimeController * tm*)*

Returns EDT time as double - seconds and microseconds.

Parameters:
tm The device handle for the SS/GS board.

Returns:
The current time in seconds from the board, precise to microseconds.

Definition at line 484 of file ss_time_lib.c.

Prominfo

Edt_undoc

Defines

#define SERIAL_ENABLED_FLAGS (PDV_EN_TX | PDV_EN_RX | PDV_EN_RX_INT | PDV_EN_DEV_INT)

Functions

int pdv_set_gain_ch (PdvDev *pdv_p, int value, int chan)

This method is obsolete and should not be used.

---

void pdv_set_interlace (PdvDev *pdv_p, int interlace)

Set the interlace flag.

---

int pdv_set_mode (PdvDev *pdv_p, char *mode, int mcl)

This method is obsolete and should not be used.

---

int pdv_set_mode_atmel (PdvDev *pdv_p, char *mode)
int pdv_set_mode_hamamatsu (PdvDev *pdv_p, char *mode)
int pdv_set_mode_kodak (PdvDev *pdv_p, char *mode)

Obsolete.

---

int pdv_set_strobe_counters (PdvDev *pdv_p, int count, int delay, int period)

pdv_set_strobe_counters.

---

int pdv_set_strobe_dac (PdvDev *pdv_p, u_int value)

Sets the strobe DAC level.

---

int pdv_strobe (PdvDev *pdv_p, int count, int delay)

Fires the strobe.

---

int pdv_strobe_method (PdvDev *pdv_p)

check if the strobe is even valid for this FPGA, and which method is used.

---

int pdv_variable_size (PdvDev *pdv_p)

Obsolete.

Function Documentation

int pdv_set_gain_ch (PdvDev * **pdv_p**, *int* **value**, *int* **chan**)*

This method is obsolete and should not be used.

The current implementation creates a warning message and returns -1 signifying failure.

Returns:
-1 for failure, always.

Definition at line 3140 of file libpdv.c.

void pdv_set_interlace (PdvDev * **pdv_p**, *int* **interlace**)*

Set the interlace flag.

Flag is no longer used so it's obsolete. Currently de-interleaving is iset via the config file and the dd_p flag is switnerlace.

Returns:
void

Definition at line 6291 of file libpdv.c.

int pdv_set_mode (PdvDev * **pdv_p**, *char* * **mode**, *int* **mcl**)*

This method is obsolete and should not be used.

The current implementation creates a warning message and returns -1 signifying failure.

Returns:
-1 for failure, always.

Definition at line 9371 of file libpdv.c.

int pdv_set_strobe_counters (PdvDev * **pdv_p**, *int* **count**, *int* **delay**, *int* **period**)*

pdv_set_strobe_counters.

NEW method (method2) – so far only for c-link but will probably be folded back into pdv/pdvk eventually. Only works with new strobe xilinx. check pdv_strobe_method for PDV_LHS_METHOD2.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

count the number of strobe pulses. range 0-4095

delay the # of msecs before the first and after the last pulse actual interval before the first pulse will be delay+period range 0-255

period delay (msecs) between pulses. range 0-255

Returns 0 on success, -1 on failure

See also:
pdv_enable_strobe, pdv_set_strobe_dac, pdv_strobe_method

Definition at line 8192 of file libpdv.c.

*int pdv_set_strobe_dac (PdvDev * pdv_p, u_int value)*

Sets the strobe DAC level.

This is a specialized routine that only works with a camera that has a strobe cable connected to an EG&G strobe with EDT strobe trigger circuitry including DAC level, and custom aia_strobe.bit XILINX downloaded.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

value DAC voltage level. Valid values are 0-4095.

Returns:
0 on success, -1 on failure.

See also:
pdv_strobe, strobe.c example program

Definition at line 8346 of file libpdv.c.

*int pdv_strobe (PdvDev * pdv_p, int count, int delay)*

Fires the strobe.

This is a specialized routine that only works with a camera that has a strobe cable connected to an EG&G or Perkin_Elmer strobe with EDT strobe trigger circuitry including DAC level, and custom aia_strobe.bit XILINX downloaded.

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

count number of strobe pulses.

delay number of msecs between pulses, as well as before the first and after the last pulse. Actual delay between flashes is 100us * (delay + 1). High 4 bits of count is ignored so maximum count is 4095.

Example

```
// fire the strobe 10 times, with a
// 101 millisecond delay between pulses
pdv_strobe(pdv_p, 10, 100);
```

Returns:
0 on success, -1 on failure

Definition at line 8153 of file libpdv.c.

*int pdv_strobe_method (PdvDev * pdv_p)*

check if the strobe is even valid for this FPGA, and which method is used.

Returns:
0 (not implemenented) PDV_LHS_METHOD1 (original method) PDV_LHS_-METHOD2 (new method)

Definition at line 8273 of file libpdv.c.

*int pdv_variable_size (PdvDev * pdv_p)*

Obsolete.

Is variable_size set ("variable_size: 1" in the config file)?

Parameters:
pdv_p pointer to pdv device structure returned by pdv_open

See also:
variable_size `camera configuration` directive

Definition at line 3686 of file libpdv.c.

Data Structure Documentation

_bitfile_list Struct Reference

Definition at line 70 of file edt_bitload.h.

Data Fields

EdtBitfileHeader * bitfiles
int nbfiles

_dma_data_block Struct Reference

Definition at line 839 of file libedt.h.

Data Fields

u_int buffernum
u_int length
u_int offset
u_char * pointer

_edt_msg_handler Struct Reference

`#include <edt_error.h>`

Structure used by the Message Handler Library to control the output of messages.

Definition at line 91 of file edt_error.h.

Data Fields

FILE * file

The file the default handler function sends output to (`stderr`).

EdtMsgFunction func
int level
unsigned char own_file

Flag set by edt_msg_set_name to indicate that we are responsible for closing the file.

---

void * target

_EdtBitfileDescriptor Struct Reference

Definition at line 891 of file libedt.h.

Data Fields

edt_bitpath bitfile_name
edt_bitpath mezz_name0
edt_bitpath mezz_name1
char mezz_optionstr0 [32]
char mezz_optionstr1 [32]
char optionstr [68]
EdtOptionStringFields ostr
int revision_register
int string_type

_EdtMezzDescriptor Struct Reference

Definition at line 679 of file libedt.h.

Data Fields

uint_t extended_data [MAX_EXTENDED_WORDS]
int extended_rev
int id
int n_extended_words

_EdtPostProc Struct Reference

Definition at line 27 of file pdv_interlace.h.

Data Fields

int dest_depth
int dest_type
void * dll_handle
char dll_name [256]
int frame_height
int func_type
int interlace
int nTaps
int offset
int order
post_process_f process
int process_mode
int shrink
int src_depth
int src_type
PdvInterleaveTap taps [MAX_INTLV_TAPS]

_optionstr_fields Struct Reference

Definition at line 879 of file libedt.h.

Data Fields

int available_DMA_channels
int board_type
int custom_DMA_channels
char date [12]
int DMA_channels
char filename [68]
char mezzanine_type [68]
int rev_number
int version_number

_PdvDependent Struct Reference

`#include <pdv_dependent.h>`

The PdvDependent structure holds PDV specific information inside the PdvDev structure.

In the PDV software package, the file *edtinc.h* defines the type `Dependent` to be `PdvDependent`.

For portability, we strongly recommend using the EDT Digital Imaging Library calls rather than accessing the structure elements directly.

Definition at line 117 of file pdv_dependent.h.

Data Fields

int acquire_mult
int aperture
int aperture_max
int aperture_min
int binx
int biny
int byteswap
int cam_height
int cam_width
int camera_binning
char camera_class [CAMCLASSLEN]
char camera_command_file [KBSFNAMELEN]
int camera_continuous
int camera_data_rate
int camera_download
char camera_download_file [KBSFNAMELEN]
char camera_info [MAXSER ∗2]
char camera_model [MAXSER]
int camera_shutter_speed
int camera_shutter_timing
int cameralink
int cameratest
char cameratype [CAMNAMELEN]
char cfgname [FNAMELEN]
int cl_cfg
int cl_cfg2
int cl_channels
int cl_data_path
int cl_hmax
ClSimControl cls
u_int cnt_continuous
u_char config_reg
int continuous

u_char datapath_reg
int dbl_trig
int default_aperture
int default_gain
int default_offset
int default_shutter_speed
int depth
int direction
int dis_shutter
int disable_mdout
int double_rate
int dual_channel
int enable_dalsa
int enddma
int extdepth
int fieldid_trig
int first_open
int fixedlen
int flushdma
char foi_init [OLDMAXINIT]
char foi_remote_rbtfile [FNAMELEN]
int foi_unit
int force_single
int frame_delay
int frame_height
int frame_period
int frame_timing
int framesync_mode
int fv_once
int fval_done
int gain
int gain_frontp
int gain_max
int gain_min
int gendata
int genericsim
int get_aperture
int get_gain
int get_offset
int hactv
int header_dma
int header_offset
int header_position

int header_size
int header_type
int height
int hskip
int htaps
int hwinterlace
int hwpad
char idstr [FNAMELEN]
int image_depth
int image_offset
int imagesize
int interlace
char interlace_module [FNAMELEN]
int interlace_offset
PdvInterleaveTap intlv_taps [MAX_INTLV_TAPS]
int inv_fvalid
int inv_ptrig
int inv_shutter
u_char irig_offset
u_char irig_raw
u_char irig_slave
int kbs_green_pixel_first
int kbs_red_row_first
int last_close
u_char * last_image
u_char * last_raw
int level
int line_delay
int linerate
int lock_shutter
int markbin
int markras
int markrasx
int markrasy
int mask
int maxdmasize
int mc4
int mode16
int mode_cntl_norm
int n_intlv_taps
int offset_frontp
int offset_max
int offset_min

int pause_for_serial
int pclock_speed
int photo_trig
int pingpong_varsize
int pulnix
int rascnt
char rbtfile [FNAMELEN]
int register_wrap
char RESERVED1 [MAXSER]
char RESERVED2 [MAXSER]
char RESERVED4 [MAXSER]
u_int RESERVEDUINT1
u_int RESERVEDUINT2
u_int RESERVEDUINT3
int rgb30
int roi_enabled
int sel_mc4
int serial_baud
char * serial_binit
char serial_binning [MAXSER]
char serial_exposure [MAXSER]
int serial_format
char serial_gain [MAXSER]
char serial_init [OLDMAXINIT]
int serial_init_delay
int serial_mode
char serial_offset [MAXSER]
char serial_prefix [MAXSER]
int serial_respcnt
char serial_response [MAXSER]
char serial_term [MAXSER]
int serial_timeout
char serial_trigger [MAXSER]
u_int serial_waitc
int set_aperture
int set_gain
int set_offset
int shift
int shortswap
int shutter_speed
int shutter_speed_frontp
int shutter_speed_max
int shutter_speed_min

u_int sim_ctl
int sim_enable
int sim_height
int sim_speed
int sim_width
int skip
int slop
int start_delay
int startdma
int started
int started_continuous
int startup_delay
int strobe_count
int strobe_enabled
int strobe_interval
int swinterlace
int timeout
int timeout_multiplier
int trig_pulse
int trigdiv
int user_timeout
int user_timeout_set
int util2
int vactv
int variable_size
int vskip
int vtaps
int width
u_char xilinx_flag [MAXXIL]
char xilinx_init [OLDMAXINIT]
int xilinx_opts
int xilinx_rev
u_char xilinx_value [MAXXIL]

_prom_addr Struct Reference

Definition at line 750 of file libedt.h.

Data Fields

u_int esn_addr
u_int extra_data_addr
u_int extra_size
u_int extra_size_addr
u_int extra_tag_addr
u_int id_addr
u_int maclist_addr
u_int optsn_addr
u_int osn_addr

_si5326_regs Struct Reference

Definition at line 23 of file edt_si5326.h.

Data Fields

int autosel_reg
int bwsel_reg
int bypass_reg
int ck1_actv_pin
int ck1_actv_reg
int ck1_bad_pin
int ck2_actv_reg
int ck2_bad_pin
int ck_actv_pol
int ck_bad_pol
int ck_prior1
int ck_prior2
int cksel_pin
int cksel_reg
int clat
int clatprogress
int clkin1rate
int clkin2rate
int dhold
int digholdvalid
int dsbl1_reg
int dsbl2_reg
int flat

int flat_valid
int fos1_flg
int fos1_int
int fos1_msk
int fos2_flg
int fos2_int
int fos2_msk
int fos_en
int fos_thr
int fosrefsel
int fxdly
int grade_ro
int hist_avg
int hist_del
int hlog_1
int hlog_2
int ical
int icmos
int incdec_pin
int independentskew1
int independentskew2
int int_pin
int int_pol
int lockt
int lol_flg
int lol_int
int lol_msk
int lol_pin
int lol_pol
int los1_flg
int los1_int
int los1_msk
int los2_flg
int los2_int
int los2_msk
int losx_flg
int losx_int
int losx_msk
int n1_hs
int n2_hs
int n2_ls
int n31
int n32

int nc1_ls
int nc2_ls
int nvm_rev
int partnum_ro
int pd_ck1
int pd_ck2
int revid_ro
int rst_reg
int sfout1_reg
int sfout2_reg
int sleep
int spim
int sq_ical
int valtime

_sim_control Struct Reference

Definition at line 29 of file pdv_dependent.h.

Data Fields

unsigned short dummy
u_char Exsyncdly
u_char FillA
u_char FillB
unsigned short hblank
unsigned short Hcntmax
unsigned short Hfvend
unsigned short Hfvstart
unsigned short Hlvend
unsigned short Hlvstart
unsigned short Hrvend
unsigned short Hrvstart
float pixel_clock
double si570_nominal
u_char taps
unsigned int vblank
unsigned int Vcntmax
u_char cfga
u_char cfgb
u_char cfgc

unsigned int datacnt:1
unsigned int dvmode:4
unsigned int dvskip:4
unsigned int firstfc:1
unsigned int intlven:1
unsigned int led:1
unsigned int linescan:1
unsigned int lvcont:1
unsigned int rven:1
unsigned int smallok:1
unsigned int trigframe:1
unsigned int trigline:1
unsigned int trigpol:1
unsigned int trigsrc:1
unsigned int uartloop:1

_tagDVCState Struct Reference

Definition at line 316 of file libpdv.h.

Data Fields

int binx
int biny
int blackoffset
int exposure
int gain
char mode [4]

_tap_descriptor Struct Reference

Definition at line 97 of file pdv_dependent.h.

Data Fields

int dx
int dy
int length
int startx
int starty
int stridex
int stridey

_timeregs Struct Reference

`#include <ss_time_lib.h>`

Structure for time access encapsulates register addresses and EdtTime-Controller poiunter.

Definition at line 22 of file ss_time_lib.h.

Data Fields

int active
u_int adj_value
double adjust_drift
int adjust_sample
int adjustment_scalar
int check_interval
u_int cmd
double converge
u_int counts
EdtDev * dev_p
int done
double drift
double err
int iter
int loop
int loops
int max_error
u_int secs
u_int set
int sign
thread_t thread
int ticks
int tolerance

buf_args Struct Reference

Definition at line 1733 of file libedt.h.

Data Fields

uint64_t addr
uint_t index
uint_t size
uint_t writeflag

cl_logic_summary Struct Reference

Definition at line 42 of file cl_logic_lib.h.

Data Fields

int bufsize
int current_frame
ClLogicStat current_width
ClLogicStat end_hblank
ClLogicStat frame_gap
ClLogicStat frameclocks
ClFrameSummary * frames
ClLogicStat hblank
ClLogicStat hblank_frame
ClLogicStat height
ClLogicStat line_stats [CL_LOGIC_MAXLINES]
int nframes
int nframesallocated
int nLines
int numbufs
double pixel_clock
ClLogicStat start_hblank
int testmask
int timeout
ClLogicStat totalframeclocks
ClLogicStat totallineclocks
ClLogicStat width

ClLogicStat Struct Reference

Definition at line 27 of file cl_logic_lib.h.

Data Fields

int high
int low
int mean
unsigned int n
uint64_t sum

cmdop Struct Reference

Definition at line 13 of file initedt.h.

Data Fields

u_int cmd_intval1
u_int cmd_intval2
char * cmd_name
cmdop * cmd_next
char * cmd_pathval
double cmd_realval
int cmd_type

Edt_bdinfo Struct Reference

Definition at line 809 of file libedt.h.

Data Fields

int bd_id
int id
int promcode
char type [8]

edt_bitfile_desc Struct Reference

Definition at line 32 of file edt_bitfile.h.

Data Fields

char * intfc_bitfile_comments [64]
int intfc_bitfile_count
char * intfc_bitfile_names [64]
char * pci_bitfile_comment
char * pci_bitfile_name

edt_board_desc Struct Reference

Definition at line 24 of file edt_bitfile.h.

Data Fields

char board_name [64]
char * intfc_bitfile
char pci_flash_name [64]
char pci_xilinx_type [64]
int unit_no

edt_buf Struct Reference

Definition at line 1708 of file libedt.h.

Data Fields

uint_t desc
uint_t flags
uint64_t value

edt_device Struct Reference

Definition at line 906 of file libedt.h.

Data Fields

u_int adt7461_reg
uint_t b_count
unsigned char * base_buffer

EdtBitfileDescriptor bfd
EdtDMADataBlock * blocks
u_int buffer_granularity
u_int channel_no
uint_t cursample
u_char * data_end
Dependent * dd_p
uint_t debug_level
uint_t devid
uint_t devtype
edt_directDMA_t * directDMA_p
u_char DMA_channels
u_int dmy_started
bufcnt_t donecount
char edt_devname [64]
EdtEventHandler event_funcs [EDT_MAX_KERNEL_EVENTS]
HANDLE fd
u_char freerun
u_int fullbufsize
int header_offset
u_int header_size
int hubidx
u_int is_serial_enabled
double last_buffer_time
char last_direction
u_char * last_sample_end
u_char last_wait_ret
uint_t loops
volatile caddr_t mapaddr
EdtMezzDescriptor mezz
uint_t minchunk
u_int mmap_buffers
double next_sample
uint_t nextwbuf
unsigned char * output_base
unsigned char ** output_buffers
void * Pdma_p
u_int pending_samples
u_int period
void * pInterleaver
uint_t port_no
u_int promcode
EdtRingBuffer rb_control [MAX_DMA_BUFFERS]

volatile u_char * reg_fifo_cnt
volatile u_char * reg_fifo_ctl
volatile u_int * reg_fifo_io
volatile u_char * reg_intfc_dat
volatile u_char * reg_intfc_off
int regBAR0_fd
int regBAR1_fd
int regUIFPGA_fd
uint_t ring_buffer_allocated_size
uint_t ring_buffer_bufsize
uint_t ring_buffer_numbufs
unsigned char * ring_buffers [MAX_DMA_BUFFERS]
uint_t ring_buffers_allocated
uint_t ring_buffers_configured
u_int spi_reg_base
unsigned char * tmpbuf
uint_t tmpbufsize
uint_t todo
u_int totalsize
u_int unit_no
u_int use_RT_for_event_func
u_char wait_mode
uint_t write_flag

edt_directDMA_t Struct Reference

Definition at line 866 of file libedt.h.

Data Fields

u_char ** bufs
int bufsize
uint64_t done_count
int initialized
int next_ringbuf
int numbufs
u_int * regmap
u_int sg_list [DDMA_FIFOSIZE *2]

edt_dma_info Struct Reference

Definition at line 652 of file libedt.h.

Data Fields

uint_t active_dma
uint_t active_list_size
uint_t alloc_dma
uint_t direct_reads [256]
uint_t direct_writes [256]
uint_t dma_reads [8]
uint_t dma_writes [8]
uint_t free_list_size
uint_t indirect_reads [256]
uint_t indirect_writes [256]
uint_t interrupts
uint_t lock_array [MAX_LOCK_SRC+1]
uint64_t lock_time
uint_t locks
uint_t used_dma
uint64_t wait_time

Edt_embinfo Struct Reference

Definition at line 689 of file libedt.h.

Data Fields

int clock
char ifx [11]
char maclist [MACLIST_SIZE]
char opt [15]
char optsn [11]
char pn [11]
int rev
char sn [11]

edt_event_handler Struct Reference

Definition at line 768 of file libedt.h.

Data Fields

u_char active
EdtEventFunc callback
u_char continuous
void * data
uint_t event_type
edt_event_handler * next
edt_device * owner

edt_ioctl_struct Struct Reference

Definition at line 1687 of file libedt.h.

Data Fields

uint32_t bytesReturned
uint32_t controlCode
HANDLE device
void * inBuffer
uint32_t inSize
void * outBuffer
uint32_t outSize

edt_ioctl_struct32 Struct Reference

Definition at line 1664 of file libedt.h.

Data Fields

uint_t bytesReturned
uint_t controlCode
HANDLE device
u_int inBuffer
uint_t inSize
uint32_t outBuffer
uint_t outSize

edt_merge_args Struct Reference

Definition at line 1743 of file libedt.h.

Data Fields

uint_t line_count
uint_t line_offset
uint_t line_size
int line_span

edt_pll Struct Reference

Definition at line 615 of file libedt.h.

Data Fields

int h
int l
int m
int n
int r
int v
int x

Edt_prominfo Struct Reference

Definition at line 700 of file libedt.h.

Data Fields

char busdesc [8]
int defaultseg
char fpga [32]
int ftype
int load_seg0
int load_seg1
int magic
u_int nsegments

char promdesc [32]
u_int sectorsize
u_int sectsperseg
u_short stat
u_short statx

edt_sdh_e1_buf Struct Reference

Definition at line 51 of file lib_sdh.h.

Data Fields

u_char e1_buf [32]
u_int e1_number: 7
u_int frame_lock: 1
u_int odd_frame: 1
u_int pad: 3
u_int time_fsecs: 20
u_int time_secs: 32

edt_sdh_e1_buf_v2 Struct Reference

Definition at line 69 of file lib_sdh.h.

Data Fields

u_char e1_buf [36]
u_int e1_number: 6
u_int e1_tag: 3
u_int frame_lock: 1
u_int length: 16
u_int odd_frame: 1
u_int raw_stm1: 1
u_int reserved1: 1
u_int reserved2: 2
u_int time_fsecs: 32
u_int time_secs: 32
u_int vc12_buf: 1

edt_sdh_t Struct Reference

Definition at line 34 of file lib_sdh.h.

Data Fields

int current_channel
int current_stm1
int dma_channel_count_per_board
int e1_count_per_dma_channel
int e1buf_version
EdtDev * edt_p
int loss_of_light
int m_numRingBufs
int m_ringBufSize
int unitNo
int active
EdtDev * edt_p

edt_sized_buffer Struct Reference

Definition at line 1763 of file libedt.h.

Data Fields

u_int data [SIZED_DATASIZE/4]
u_int size

EdtBitfile Struct Reference

`#include <edt_bitload.h>`

Retrieve the possibilities for a particular board for UI bitfile.

Definition at line 51 of file edt_bitload.h.

Data Fields

u_int buffer_allocated
u_int cur_index
u_char * data

u_int data_size
HANDLE f
char * filename
u_char * full_buffer
u_int full_buffer_size
EdtBitfileHeader hdr
int is_file

EdtBitfileHeader Struct Reference

Definition at line 30 of file edt_bitload.h.

Data Fields

u_int data_start
u_char date [16]
u_int dsize
u_char extra [BFH_EXTRASIZE]
u_char fi [8]
char filename [MAXPATH]
u_int filesize
u_char id [32]
u_int key
int magic
u_char ncdname [MAXPATH]
char promstr [256]
u_char time [16]

EdtBoardFpgas Struct Reference

Definition at line 80 of file edt_bitload.h.

Data Fields

char * fpga_0 [MAX_CHIPS_PER_ID]
char * fpga_1 [MAX_CHIPS_PER_ID]
u_int id

EdtPromData Struct Reference

Definition at line 731 of file libedt.h.

Data Fields

Edt_embinfo ei
char esn [ESN_SIZE]
u_char extra_buf [PROM_EXTRA_SIZE]
int extra_size
char id [PCI_ID_SIZE]
char maclist [MACLIST_SIZE]
int nblocks
char optsn [ESN_SIZE]
char osn [OSN_SIZE]

EdtPromParmBlock Struct Reference

Definition at line 721 of file libedt.h.

Data Fields

u_int size
char type [4]

EdtRingBuffer Struct Reference

Definition at line 819 of file libedt.h.

Data Fields

int allocated_size
char owned
int size
char write_flag

EdtThreePClocks Struct Reference

Definition at line 299 of file edt_threep.h.

Data Fields

double ch0_clock_freq
EdtSI570 ch0_clock_values
double ch1_clock_freq
EdtSI570 ch1_clock_values
double ch2_clock_freq
EdtSI570 ch2_clock_values
double xmt_clock_freq
int xmt_clock_source
EdtSI53xx xmt_clock_values

frame_summary Struct Reference

Definition at line 36 of file cl_logic_lib.h.

Data Fields

int frame_blank
int height
int line_blank
int width

line_delta Struct Reference

Definition at line 20 of file cl_logic_lib.h.

Data Fields

int delta
int n
int newval

p53b_test Struct Reference

Definition at line 1753 of file libedt.h.

Data Fields

u_int addr
u_int cnt
u_int inc
u_int mask
u_int size

Pdma_t Struct Reference

Definition at line 14 of file libpdma.h.

Data Fields

u_char * data_p
u_int dma_count
u_int dma_intr_en
u_int * dma_sglist
u_int dma_sglist_copy [PDMA_SG_SIZE]
u_int * dmaaddr
u_int * dmacfg
u_int * dmacmd
u_int * dmacnt
u_char * off_p
u_char ** pdma_databufs
u_int pdma_size
u_int sg_paddr
u_int sv_dma_cfg

ser_buf Struct Reference

Definition at line 1723 of file libedt.h.

Data Fields

char buf [EDT_SERBUF_SIZE]
uint_t flags
uint_t misc
uint_t size
uint_t unit

si_info Struct Reference

Definition at line 110 of file edt_si5326.h.

Data Fields

int bwsel
int n1_hs
int n1_ls
int n2_hs
int n2_ls
int n3
double output