

# EDT Application Programming Interface

Engineering Design Team, Inc.

<http://www.edt.com>

April 30, 2018



a HEICO company

## Contents

Introduction . . . . .	1
Libraries . . . . .	1
Terms of Use . . . . .	1
Copyright, Trademarks . . . . .	2
<b>EDT DMA Library</b>	<b>3</b>
Startup / Shutdown . . . . .	8
Function Documentation . . . . .	8
edt_open . . . . .	8
edt_open_channel . . . . .	9
edt_open_device . . . . .	10
edt_open_quiet . . . . .	10
Initialization . . . . .	11
Function Documentation . . . . .	11
edt_bitload . . . . .	11
FIFO Flushing . . . . .	12
Function Documentation . . . . .	12
edt_disable_channel . . . . .	12
edt_disable_channels . . . . .	13
edt_enable_channel . . . . .	13
edt_enable_channels . . . . .	13
edt_flush_fifo . . . . .	14
edt_get_firstflush . . . . .	14
edt_set_firstflush . . . . .	14
Input/Output . . . . .	16
Function Documentation . . . . .	21
edt_abort_current_dma . . . . .	21
edt_abort_dma . . . . .	22
edt_allocated_size . . . . .	22
edt_buffer_addresses . . . . .	22

<a href="#">edt_check_for_buffers</a> . . . . .	23
<a href="#">edt_configure_block_buffers</a> . . . . .	23
<a href="#">edt_configure_block_buffers_mem</a> . . . . .	24
<a href="#">edt_configure_ring_buffers</a> . . . . .	24
<a href="#">edt_disable_ring_buffers</a> . . . . .	25
<a href="#">edt_do_timeout</a> . . . . .	25
<a href="#">edt_done_count</a> . . . . .	26
<a href="#">edt_enddma_action</a> . . . . .	26
<a href="#">edt_enddma_reg</a> . . . . .	27
<a href="#">edt_get_burst_enable</a> . . . . .	27
<a href="#">edt_get_bytecount</a> . . . . .	28
<a href="#">edt_get_current_dma_buf</a> . . . . .	28
<a href="#">edt_get_direction</a> . . . . .	28
<a href="#">edt_get_goodbits</a> . . . . .	28
<a href="#">edt_get_numbufs</a> . . . . .	29
<a href="#">edt_get_reftime</a> . . . . .	29
<a href="#">edt_get_rtimeout</a> . . . . .	29
<a href="#">edt_get_timeout_count</a> . . . . .	30
<a href="#">edt_get_timeout_goodbits</a> . . . . .	30
<a href="#">edt_get_timestamp</a> . . . . .	30
<a href="#">edt_get_todo</a> . . . . .	31
<a href="#">edt_get_total_bufsize</a> . . . . .	31
<a href="#">edt_get_wtimeout</a> . . . . .	31
<a href="#">edt_last_buffer</a> . . . . .	32
<a href="#">edt_last_buffer_timed</a> . . . . .	32
<a href="#">edt_next_writebuf</a> . . . . .	33
<a href="#">edt_next_writebuf_index</a> . . . . .	33
<a href="#">edt_read</a> . . . . .	33
<a href="#">edt_read_end_action</a> . . . . .	34
<a href="#">edt_read_start_action</a> . . . . .	35
<a href="#">edt_ref_tmstamp</a> . . . . .	35

edt_remove_event_func . . . . .	36
edt_reset_ring_buffers . . . . .	36
edt_ring_buffer_overflow . . . . .	37
edt_set_buffer . . . . .	37
edt_set_buffer_size . . . . .	38
edt_set_burst_enable . . . . .	38
edt_set_direction . . . . .	39
edt_set_dmy_reg_read_callback . . . . .	39
edt_set_dmy_reg_write_callback . . . . .	39
edt_set_dmy_wait_for_buffers_callback . . . . .	40
edt_set_event_func . . . . .	40
edt_set_rtimeout . . . . .	41
edt_set_timeout_action . . . . .	42
edt_set_wtimeout . . . . .	42
edt_start_buffers . . . . .	43
edt_startdma_action . . . . .	43
edt_startdma_reg . . . . .	44
edt_stop_buffers . . . . .	44
edt_timeouts . . . . .	44
edt_wait_buffers_timed . . . . .	45
edt_wait_for_buffers . . . . .	45
edt_wait_for_next_buffer . . . . .	46
edt_write . . . . .	46
edt_write_end_action . . . . .	47
edt_write_start_action . . . . .	47
Register Access . . . . .	49
Function Documentation . . . . .	50
edt_bar1_read . . . . .	50
edt_bar1_write . . . . .	50
edt_intfc_read . . . . .	51
edt_intfc_read_32 . . . . .	51

edt_intfc_read_short . . . . .	52
edt_intfc_write_32 . . . . .	53
edt_intfc_write_short . . . . .	53
edt_reg_and . . . . .	54
edt_reg_clearset . . . . .	54
edt_reg_or . . . . .	54
edt_reg_read . . . . .	55
edt_reg_setclear . . . . .	55
edt_reg_write . . . . .	55
Utility . . . . .	57
Function Documentation . . . . .	61
edt_access . . . . .	61
edt_device_id . . . . .	61
edt_errno . . . . .	61
edt_find_xpn . . . . .	62
edt_get_bitname . . . . .	62
edt_get_bitpath . . . . .	63
edt_get_board_id . . . . .	63
edt_get_dma_info . . . . .	63
edt_get_driver_buildid . . . . .	64
edt_get_driver_version . . . . .	65
edt_get_esn . . . . .	65
edt_get_full_board_id . . . . .	66
edt_get_library_buildid . . . . .	67
edt_get_library_version . . . . .	68
edt_get_mezz_bitpath . . . . .	68
edt_get_mezz_chan_bitpath . . . . .	68
edt_get_osn . . . . .	69
edt_get_sns_sector . . . . .	69
edt_get_xref_info . . . . .	70
edt_idstr . . . . .	71

edt_idstring . . . . .	72
edt_parse_devinfo . . . . .	72
edt_parse_unit . . . . .	73
edt_parse_unit_channel . . . . .	73
edt_perror . . . . .	74
edt_set_bitpath . . . . .	74
edt_set_mezz_bitpath . . . . .	74
edt_set_mezz_chan_bitpath . . . . .	75
edt_system . . . . .	75
<b>EDT Digital Imaging Library</b>	<b>76</b>
Startup / Shutdown . . . . .	80
Function Documentation . . . . .	80
pdv_close . . . . .	80
pdv_open . . . . .	80
pdv_open_channel . . . . .	81
Settings . . . . .	83
Function Documentation . . . . .	89
pdv_auto_set_roi . . . . .	89
pdv_camera_type . . . . .	89
pdv_check_framesync . . . . .	90
pdv_cl_set_base_channels . . . . .	91
pdv_enable_external_trigger . . . . .	91
pdv_enable_framesync . . . . .	92
pdv_enable_lock . . . . .	92
pdv_enable_roi . . . . .	93
pdv_extra_headersize . . . . .	93
pdv_framesync_mode . . . . .	93
pdv_get_allocated_size . . . . .	94
pdv_get_blacklevel . . . . .	94
pdv_get_bytes_per_image . . . . .	95

<a href="#">pdv_get_cam_height</a>	95
<a href="#">pdv_get_cam_width</a>	95
<a href="#">pdv_get_camera_class</a>	96
<a href="#">pdv_get_camera_info</a>	96
<a href="#">pdv_get_camera_model</a>	97
<a href="#">pdv_get_cameratype</a>	97
<a href="#">pdv_get_depth</a>	98
<a href="#">pdv_get_dmasize</a>	98
<a href="#">pdv_get_exposure</a>	99
<a href="#">pdv_get_extdepth</a>	99
<a href="#">pdv_get_firstpixel_counter</a>	99
<a href="#">pdv_get_frame_height</a>	100
<a href="#">pdv_get_frame_period</a>	100
<a href="#">pdv_get_gain</a>	101
<a href="#">pdv_get_header_dma</a>	101
<a href="#">pdv_get_header_offset</a>	101
<a href="#">pdv_get_header_position</a>	102
<a href="#">pdv_get_header_size</a>	103
<a href="#">pdv_get_header_within</a>	103
<a href="#">pdv_get_height</a>	103
<a href="#">pdv_get_imagesize</a>	104
<a href="#">pdv_get_invert</a>	104
<a href="#">pdv_get_max_gain</a>	104
<a href="#">pdv_get_max_offset</a>	105
<a href="#">pdv_get_max_shutter</a>	105
<a href="#">pdv_get_min_gain</a>	105
<a href="#">pdv_get_min_offset</a>	106
<a href="#">pdv_get_min_shutter</a>	106
<a href="#">pdv_get_pitch</a>	106
<a href="#">pdv_get_shutter_method</a>	107
<a href="#">pdv_get_width</a>	107

<a href="#">pdv_image_size</a>	107
<a href="#">pdv_invert</a>	108
<a href="#">pdv_invert_fval_interrupt</a>	108
<a href="#">pdv_picture_timeout</a>	108
<a href="#">pdv_read_response</a>	109
<a href="#">pdv_set_binning</a>	109
<a href="#">pdv_set_binning_dvc</a>	110
<a href="#">pdv_set_blacklevel</a>	110
<a href="#">pdv_set_cam_height</a>	111
<a href="#">pdv_set_cam_width</a>	111
<a href="#">pdv_set_cameratype</a>	111
<a href="#">pdv_set_depth</a>	112
<a href="#">pdv_set_depth_extdepth</a>	113
<a href="#">pdv_set_depth_extdepth_dpath</a>	113
<a href="#">pdv_set_exposure</a>	114
<a href="#">pdv_set_exposure_duncan_ch</a>	115
<a href="#">pdv_set_exposure_mcl</a>	116
<a href="#">pdv_set_extdepth</a>	116
<a href="#">pdv_set_firstpixel_counter</a>	117
<a href="#">pdv_set_frame_period</a>	117
<a href="#">pdv_set_full_bayer_parameters</a>	118
<a href="#">pdv_set_gain</a>	119
<a href="#">pdv_set_gain_duncan_ch</a>	120
<a href="#">pdv_set_header_dma</a>	120
<a href="#">pdv_set_header_offset</a>	121
<a href="#">pdv_set_header_position</a>	121
<a href="#">pdv_set_header_size</a>	122
<a href="#">pdv_set_header_type</a>	122
<a href="#">pdv_set_height</a>	123
<a href="#">pdv_set_roi</a>	123
<a href="#">pdv_set_shutter_method</a>	124



pdv_set_width . . . . .	126
pdv_setsize . . . . .	126
pdv_shutter_method . . . . .	126
Initialization . . . . .	128
Function Documentation . . . . .	128
pdv_alloc_dependent . . . . .	128
pdv_auto_set_timeout . . . . .	129
pdv_initcam . . . . .	129
pdv_readcfg . . . . .	130
Acquisition . . . . .	132
Function Documentation . . . . .	138
pdv_buffer_addresses . . . . .	138
pdv_cl_get_fv_counter . . . . .	138
pdv_cl_reset_fv_counter . . . . .	139
pdv_flush_fifo . . . . .	139
pdv_force_single . . . . .	140
pdv_get_last_image . . . . .	140
pdv_get_last_raw . . . . .	141
pdv_get_lines_xferred . . . . .	141
pdv_get_timeout . . . . .	141
pdv_get_width_xferred . . . . .	142
pdv_image . . . . .	142
pdv_image_raw . . . . .	143
pdv_in_continuous . . . . .	143
pdv_interlace_method . . . . .	143
pdv_multibuf . . . . .	145
pdv_overrun . . . . .	145
pdv_read . . . . .	146
pdv_set_buffers . . . . .	146
pdv_set_fval_done . . . . .	147
pdv_set_timeout . . . . .	148

---

pdv_setup_continuous	148
pdv_setup_continuous_channel	149
pdv_setup_dma	149
pdv_start_expose	149
pdv_start_hardware_continuous	149
pdv_start_image	150
pdv_start_images	150
pdv_stop_continuous	151
pdv_stop_hardware_continuous	151
pdv_timeout_cleanup	152
pdv_timeout_restart	152
pdv_timeouts	153
pdv_wait_image	154
pdv_wait_image_raw	155
pdv_wait_image_timed	156
pdv_wait_image_timed_raw	157
pdv_wait_images	158
pdv_wait_images_raw	159
pdv_wait_images_timed	159
pdv_wait_images_timed_raw	160
pdv_wait_last_image	161
pdv_wait_last_image_raw	162
pdv_wait_last_image_timed	162
pdv_wait_last_image_timed_raw	163
pdv_wait_next_image	164
pdv_wait_next_image_raw	164
Communications/Control	165
Function Documentation	169
pdv_get_baud	169
pdv_get_serial_block_size	169
pdv_get_waitchar	169

<a href="#">pdv_query_serial</a>	170
<a href="#">pdv_read_basler_frame</a>	170
<a href="#">pdv_read_duncan_frame</a>	170
<a href="#">pdv_reset_serial</a>	170
<a href="#">pdv_send_basler_command</a>	171
<a href="#">pdv_send_basler_frame</a>	171
<a href="#">pdv_send_break</a>	171
<a href="#">pdv_send_duncan_frame</a>	172
<a href="#">pdv_send_msg</a>	172
<a href="#">pdv_serial_binary_command</a>	172
<a href="#">pdv_serial_binary_command_flagged</a>	173
<a href="#">pdv_serial_command</a>	173
<a href="#">pdv_serial_command_flagged</a>	174
<a href="#">pdv_serial_command_hex</a>	175
<a href="#">pdv_serial_get_numbytes</a>	175
<a href="#">pdv_serial_prefix</a>	176
<a href="#">pdv_serial_read</a>	176
<a href="#">pdv_serial_read_blocking</a>	177
<a href="#">pdv_serial_read_nullterm</a>	177
<a href="#">pdv_serial_term</a>	178
<a href="#">pdv_serial_txx</a>	178
<a href="#">pdv_serial_wait</a>	178
<a href="#">pdv_serial_wait_next</a>	179
<a href="#">pdv_serial_write</a>	179
<a href="#">pdv_serial_write_available</a>	180
<a href="#">pdv_serial_write_single_block</a>	180
<a href="#">pdv_set_baud</a>	181
<a href="#">pdv_set_serial_block_size</a>	181
<a href="#">pdv_set_serial_delimiters</a>	181
<a href="#">pdv_set_serial_parity</a>	182
<a href="#">pdv_set_waitchar</a>	182

Utility	183
Function Documentation	186
dvu_exp_histeq	186
dvu_histeq	186
dvu_write_bmp	187
dvu_write_bmp_24	187
dvu_write_image	188
dvu_write_image24	188
dvu_write_rasfile	189
dvu_write_rasfile16	189
dvu_write_rasfile24	190
dvu_write_raw	190
pdv_access	190
pdv_alloc	191
pdv_bytes_per_line	191
pdv_cl_camera_connected	192
pdv_free	192
pdv_is_atmel	192
pdv_is_cameralink	193
pdv_is_dvc	193
pdv_is_hamamatsu	193
pdv_is_kodak_i	194
pdv_is_simulator	194
pdv_perror	194
pdv_update_values_from_camera	194
Debug	196
Function Documentation	196
pdv_debug	196
pdv_debug_level	196
EDT Camera Link Simulator Library	197
Function Documentation	200

<a href="#">pdv_cls_dep_sanity_check</a>	200
<a href="#">pdv_cls_dump_geometry</a>	200
<a href="#">pdv_cls_dump_state</a>	201
<a href="#">pdv_cls_frame_time</a>	201
<a href="#">pdv_cls_get_hgap</a>	201
<a href="#">pdv_cls_get_vgap</a>	201
<a href="#">pdv_cls_init_serial</a>	202
<a href="#">pdv_cls_set_clock</a>	202
<a href="#">pdv_cls_set_datacnt</a>	202
<a href="#">pdv_cls_set_dep</a>	203
<a href="#">pdv_cls_set_dvalid</a>	203
<a href="#">pdv_cls_set_fill</a>	203
<a href="#">pdv_cls_set_firstfc</a>	204
<a href="#">pdv_cls_set_height</a>	204
<a href="#">pdv_cls_set_intlven</a>	204
<a href="#">pdv_cls_set_led</a>	205
<a href="#">pdv_cls_set_line_timing</a>	205
<a href="#">pdv_cls_set_linescan</a>	206
<a href="#">pdv_cls_set_lvcont</a>	206
<a href="#">pdv_cls_set_readvalid</a>	207
<a href="#">pdv_cls_set_rven</a>	207
<a href="#">pdv_cls_set_size</a>	207
<a href="#">pdv_cls_set_smallok</a>	208
<a href="#">pdv_cls_set_trigframe</a>	208
<a href="#">pdv_cls_set_trigline</a>	209
<a href="#">pdv_cls_set_trigpol</a>	209
<a href="#">pdv_cls_set_trigsrc</a>	210
<a href="#">pdv_cls_set_uartloop</a>	210
<a href="#">pdv_cls_set_width</a>	210
<a href="#">pdv_cls_set_width_lval_rval</a>	211
<a href="#">pdv_cls_setup_interleave</a>	211

---

pdv_cls_sim_start . . . . .	212
pdv_cls_sim_stop . . . . .	212
<b>EDT Message Handler Library</b>	<b>213</b>
Typedef Documentation . . . . .	217
EdtMsgFunction . . . . .	217
Function Documentation . . . . .	217
edt_msg . . . . .	217
edt_msg_add_level . . . . .	218
edt_msg_close . . . . .	218
edt_msg_default_handle . . . . .	219
edt_msg_default_level . . . . .	219
edt_msg_get_level . . . . .	219
edt_msg_init . . . . .	219
edt_msg_init_files . . . . .	220
edt_msg_init_names . . . . .	220
edt_msg_output . . . . .	220
edt_msg_output_perror . . . . .	221
edt_msg_output_printf_perror . . . . .	221
edt_msg_perror . . . . .	222
edt_msg_printf_perror . . . . .	222
edt_msg_set_file . . . . .	223
edt_msg_set_function . . . . .	224
edt_msg_set_level . . . . .	224
edt_msg_set_name . . . . .	224
edt_msg_set_target . . . . .	225
<b>OCM/OC192 Library</b>	<b>226</b>
OCM Mezzanine Access Functions . . . . .	228
OC192 Mezzanine Access Functions . . . . .	229
OC192 LIU Access Functions . . . . .	230
IRIG-B Timecode Library . . . . .	231

---

Configuration Functions . . . . .	232
Display Functions . . . . .	233
Firmware Update Functions . . . . .	234
SDH to E1 Firmware Demultiplex Library . . . . .	235
<b>EDT Time Library</b>	<b>236</b>
Function Documentation . . . . .	239
edt_sstm_adjuster_start . . . . .	239
edt_sstm_adjuster_stop . . . . .	239
edt_sstm_disable_adjust . . . . .	240
edt_sstm_enable_adjust . . . . .	240
edt_sstm_get_adj_sample_secs . . . . .	240
edt_sstm_get_adj_samples . . . . .	240
edt_sstm_get_adjust_enabled . . . . .	240
edt_sstm_get_adjust_sign . . . . .	241
edt_sstm_get_adjust_ticks . . . . .	241
edt_sstm_get_counts . . . . .	241
edt_sstm_get_seconds . . . . .	241
edt_sstm_get_time_parts . . . . .	241
edt_sstm_get_usecs . . . . .	242
edt_sstm_latch_time . . . . .	242
edt_sstm_launch_adjuster . . . . .	242
edt_sstm_measure_drift . . . . .	242
edt_sstm_set . . . . .	243
edt_sstm_set_adj_from_drift . . . . .	243
edt_sstm_set_adj_sign . . . . .	243
edt_sstm_set_adj_ticks . . . . .	243
edt_sstm_set_drift_sampling . . . . .	244
edt_sstm_set_secs . . . . .	244
edt_sstm_set_to_sys . . . . .	244
edt_sstm_set_to_sys_error . . . . .	244

edt_sstm_setup . . . . .	245
edt_sstm_strobe . . . . .	245
edt_sstm_sys_error . . . . .	245
edt_sstm_ticks_from_drift . . . . .	246
edt_sstm_timestamp . . . . .	246
Prominfo . . . . .	247
Edt_undoc . . . . .	248
Function Documentation . . . . .	249
pdv_set_gain_ch . . . . .	249
pdv_set_interlace . . . . .	249
pdv_set_mode . . . . .	249
pdv_set_strobe_counters . . . . .	249
pdv_set_strobe_dac . . . . .	250
pdv_strobe . . . . .	250
pdv_strobe_method . . . . .	251
pdv_variable_size . . . . .	251
<b>Data Structure Documentation</b> . . . . .	<b>252</b>
_bitfile_list Struct Reference . . . . .	252
_dma_data_block Struct Reference . . . . .	252
_edt_msg_handler Struct Reference . . . . .	252
_EdtBitfileDescriptor Struct Reference . . . . .	253
_EdtMezzDescriptor Struct Reference . . . . .	253
_EdtPostProc Struct Reference . . . . .	253
_optionstr_fields Struct Reference . . . . .	254
_PdvDependent Struct Reference . . . . .	254
_prom_addr Struct Reference . . . . .	259
_si5326_regs Struct Reference . . . . .	260
_sim_control Struct Reference . . . . .	262
_tagDVCTestState Struct Reference . . . . .	263
_tap_descriptor Struct Reference . . . . .	263



<a href="#">_timeregs Struct Reference</a>	264
<a href="#">buf_args Struct Reference</a>	264
<a href="#">cl_logic_summary Struct Reference</a>	265
<a href="#">CILogicStat Struct Reference</a>	265
<a href="#">cmdop Struct Reference</a>	266
<a href="#">Edt_bdinfo Struct Reference</a>	266
<a href="#">edt_bitfile_desc Struct Reference</a>	266
<a href="#">edt_board_desc Struct Reference</a>	267
<a href="#">edt_buf Struct Reference</a>	267
<a href="#">edt_device Struct Reference</a>	267
<a href="#">edt_directDMA_t Struct Reference</a>	269
<a href="#">edt_dma_info Struct Reference</a>	270
<a href="#">Edt_embinfo Struct Reference</a>	270
<a href="#">edt_event_handler Struct Reference</a>	270
<a href="#">edt_ioctl_struct Struct Reference</a>	271
<a href="#">edt_ioctl_struct32 Struct Reference</a>	271
<a href="#">edt_merge_args Struct Reference</a>	272
<a href="#">edt_pll Struct Reference</a>	272
<a href="#">Edt_prominfo Struct Reference</a>	272
<a href="#">edt_sdh_e1_buf Struct Reference</a>	273
<a href="#">edt_sdh_e1_buf_v2 Struct Reference</a>	273
<a href="#">edt_sdh_t Struct Reference</a>	274
<a href="#">edt_sized_buffer Struct Reference</a>	274
<a href="#">EdtBitfile Struct Reference</a>	274
<a href="#">EdtBitfileHeader Struct Reference</a>	275
<a href="#">EdtBoardFpgas Struct Reference</a>	275
<a href="#">EdtPromData Struct Reference</a>	276
<a href="#">EdtPromParmBlock Struct Reference</a>	276
<a href="#">EdtRingBuffer Struct Reference</a>	276
<a href="#">EdtThreePClocks Struct Reference</a>	276
<a href="#">frame_summary Struct Reference</a>	277

---

<a href="#">line_delta Struct Reference</a> . . . . .	277
<a href="#">p53b_test Struct Reference</a> . . . . .	277
<a href="#">Pdma_t Struct Reference</a> . . . . .	278
<a href="#">ser_buf Struct Reference</a> . . . . .	278
<a href="#">si_info Struct Reference</a> . . . . .	279

## 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 merchantability 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**

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

Copyright ©2007-2013 Engineering Design Team, Inc. All rights reserved.

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

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

## 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 `edtlb` calls may be necessary in DV applications, however we can not provide support for applications that directly call `edtlb` 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 `ioctl`s.

**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, examples, 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 printf's 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 other `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_t` 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\_overflow (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(*call-
Back)(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(*call-  
Back)(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(*call-  
Back)(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 success, -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 success, -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 aligned 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, unsigned 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 aligned 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_FUNCNT, 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 separate, 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.

synopsis: 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 ioctl.

***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 ioctl.

***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 ioctl's.

**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 ioctl's.

**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_fciousps(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\_pcd\_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 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 *edtdreg.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.\n");
    printf("It may not be safe to start DMA on this unit/channel outside the currently opened process.\n");
}
```

### 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:macaddr:

(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 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 information 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 subsubroutine 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 concatenation ("pcd0," "pcd1," etc.)

**dev** array to hold the device device string; filled in by this 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](#) (edtlb). 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 bookkeeping, error-recovery, triggering and timing functionality that is present on EDT Digital Imaging boards. Therefore direct calls to edtlb (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 edtlb 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](http://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](mailto: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 [edt-inc.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 [edt-inc.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 bandwidth, 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 footer 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

**vtaps** 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 Megaplug 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/disable 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 library, such as the Kodak Megaplug 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 specific 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 Megaplu<sup>s</sup> 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 footer position value.

The header position value can be one 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 height 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 acquire.

**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* \* p dv \_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* \* p dv \_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* \* p dv \_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 acquire.

**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 acquire.

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 usually 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 usually 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** exposure 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 other 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, calculate 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 appendix 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 vactv + 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 (including 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 hexadecimal 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 [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 available 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 to use these subroutines to read configuration 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, Edt\_info \* ei\_p, const char \* cfgfname, char \* bitdir, int pdv\_debug)***

Initializes the framegrabber board and camera.

This is the "guts" of the *intcam* 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\_initcam* 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 further 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.

**edinfo** 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, &edinfo);
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 edinfo 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 <a href="#">Camera Configuration guide</a> )	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 deprecated 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\_interface** 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 transferred 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 acquire.

***Parameters:***

**pdv\_p** pointer to pdv device structure returned by [pdv\\_open](#)

***Returns:***

Number of bytes of data remaining from last acquire. 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 acquiring 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 sub-routines 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 deprecated 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 condition. 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 problematic 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 **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](#), [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* \* *timestamp*, *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](#)

***timestamp*** 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 specified 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 \* n-Skipped*)**

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 (enabled 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 null-
term)
```

*Performs 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 receive – send a command and wait for the response.*

---

```
int pdv\_serial\_wait (PdvDev *pdv_p, int msec, 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 msec, 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 newsz)
```

*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 rflag, 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

**rflag** 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 bytes  
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 requirements.

### 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*)

Performs 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\\_txx](#) ([PdvDev](#) \* pdv\_p, *char* \* txbuf, *int* txcount, *char* \* rxbuf, *int* rxcount, *int* timeout, *u\_char* \* wchar)**

Serial send AND receive – 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* msec, *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 default timeout value is used, as specified by the **serial\_timeout** directive in the current configuration file. If no default timeout 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 newsiz)***

Sets the block size for serial writes if the default of 512 is not adequate.

***Parameters:***

**newsiz** 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 parity. 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 conceivably 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 cut-off)
```

*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 (re-  
gardless 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 (re-  
gardless 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 successive 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 Megaplug '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-initializes 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 Hfvs-  
tart, 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 polarity.*

---

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 hlwend, int hrvstart, int hrwend)
```

*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-initializes 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 requested 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 fourth 8-bit pixel of 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\\_smalllok](#)

**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 first 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 position

*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

**totalheight** 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, allowing 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 polarity.

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 [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 edtlb 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 pdv 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, initialized 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 initialized 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, initialized 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) {
    GtkWidget * 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, initialized 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, initialized 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*, *EdtMsg-Function* *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 function 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 summarizes 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\_adj\_just\_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 =  $(1 \ll 20) / 1000000$  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_interface (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 pe-  
riod)
```

*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 implemented) 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
CISimControl 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\_rbtfiler [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 pointer.

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](#)  
CILogicStat [current\\_width](#)  
CILogicStat [end\\_hblank](#)  
CILogicStat [frame\\_gap](#)  
CILogicStat [frameclocks](#)  
CFrameSummary \* [frames](#)  
CILogicStat [hblank](#)  
CILogicStat [hblank\\_frame](#)  
CILogicStat [height](#)  
CILogicStat [line\\_stats](#) [CL\_LOGIC\_MAXLINES]  
int [nframes](#)  
int [nframesallocated](#)  
int [nLines](#)  
int [numbufs](#)  
double [pixel\\_clock](#)  
CILogicStat [start\\_hblank](#)  
int [testmask](#)  
int [timeout](#)  
CILogicStat [totalframeclocks](#)  
CILogicStat [totallineclocks](#)  
CILogicStat [width](#)

**CILogicStat 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 sectspersseg
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
```