

Freescal Touch

Touch Sensing Software - Freescal Touch

Document Number: FTUG
1.0.0
Apr 2015



Contents

Chapter 1 Introduction

Chapter 2 Key Library Elements

2.1	System	5
2.2	Modules	5
2.2.1	Module Types	5
2.2.2	TSI details	5
2.3	Key Detectors	7
2.3.1	Types	7
2.4	Controls	8
2.5	Electrodes	11

Chapter 3 Directory Structure

Chapter 4 Configuring the Library

4.1	Configuration Example	15
4.1.1	Key Detectors	15
4.1.2	Electrodes	15
4.1.3	Modules	16
4.1.3.1	TSI module	16
4.1.4	Controls	16
4.1.4.1	Analog slider	16
4.1.4.2	Slider	17

Section number	Title	Page
4.1.4.3	Keypad	17
4.1.4.4	Analog rotary	17
4.1.4.5	Rotary	18
4.1.5	System	18

Chapter 5 Your First Application

5.1	Creating the Freescale Touch Application	19
5.1.1	Adding Library Files into the Project	19
5.1.2	Setting 'include' Search Paths	20
5.1.3	Setting 'linker' path	22
5.1.4	Application Setup	22
5.1.5	The main() Function	25

Chapter 6 Release Notes

Chapter 7 Freescale Touch User API

7.1	Overview	33
7.1.1	Analog Rotary Control	34
7.1.1.1	Overview	34
7.1.1.2	Data Structure Documentation	35
7.1.1.2.1	struct ft_control_arotary	35
7.1.1.3	Typedef Documentation	36
7.1.1.3.1	ft_control_arotary_callback	36
7.1.1.4	Enumeration Type Documentation	36
7.1.1.4.1	ft_control_arotary_event	36
7.1.1.5	Variable Documentation	36
7.1.1.5.1	ft_control_arotary_interface	36
7.1.1.6	Analog Rotary Control API	37
7.1.1.6.1	Overview	37
7.1.1.6.2	Function Documentation	38
7.1.1.6.2.1	ft_control_arotary_get_direction	38
7.1.1.6.2.2	ft_control_arotary_get_invalid_position	39
7.1.1.6.2.3	ft_control_arotary_get_position	40
7.1.1.6.2.4	ft_control_arotary_is_touched	40
7.1.1.6.2.5	ft_control_arotary_movement_detected	41

Section number	Title	Page
7.1.1.6.2.6	ft_control_arotary_register_callback	42
7.1.2	Analog Slider Control	44
7.1.2.1	Overview	44
7.1.2.2	Data Structure Documentation	45
7.1.2.2.1	struct ft_control_aslider	45
7.1.2.3	Typedef Documentation	46
7.1.2.3.1	ft_control_aslider_callback	46
7.1.2.4	Enumeration Type Documentation	46
7.1.2.4.1	ft_control_aslider_event	46
7.1.2.5	Variable Documentation	46
7.1.2.5.1	ft_control_aslider_interface	46
7.1.2.6	Analog Slider Control API	47
7.1.2.6.1	Overview	47
7.1.2.6.2	Function Documentation	48
7.1.2.6.2.1	ft_control_aslider_get_direction	48
7.1.2.6.2.2	ft_control_aslider_get_invalid_position	49
7.1.2.6.2.3	ft_control_aslider_get_position	50
7.1.2.6.2.4	ft_control_aslider_is_touched	50
7.1.2.6.2.5	ft_control_aslider_movement_detected	51
7.1.2.6.2.6	ft_control_aslider_register_callback	52
7.1.3	Keypad Control	54
7.1.3.1	Overview	54
7.1.3.2	Data Structure Documentation	55
7.1.3.2.1	struct ft_control_keypad	55
7.1.3.3	Typedef Documentation	56
7.1.3.3.1	ft_control_keypad_callback	56
7.1.3.4	Enumeration Type Documentation	56
7.1.3.4.1	ft_control_keypad_event	56
7.1.3.5	Variable Documentation	57
7.1.3.5.1	ft_control_keypad_interface	57
7.1.3.6	Keypad Control API	58
7.1.3.6.1	Overview	58
7.1.3.6.2	Function Documentation	59
7.1.3.6.2.1	ft_control_keypad_get_autorepeat_rate	59
7.1.3.6.2.2	ft_control_keypad_is_button_touched	60
7.1.3.6.2.3	ft_control_keypad_only_one_key_valid	61
7.1.3.6.2.4	ft_control_keypad_register_callback	61
7.1.3.6.2.5	ft_control_keypad_set_autorepeat_rate	62
7.1.4	Matrix Control	64
7.1.4.1	Overview	64
7.1.4.2	Matrix Control API	65
7.1.5	Rotary Control	66
7.1.5.1	Overview	66
7.1.5.2	Typedef Documentation	67
7.1.5.2.1	ft_control_rotary_callback	67

Section number	Title	Page
7.1.5.3	Enumeration Type Documentation	67
7.1.5.3.1	ft_control_rotary_event	67
7.1.5.4	Variable Documentation	67
7.1.5.4.1	ft_control_rotary_interface	67
7.1.5.5	Rotary Control API	68
7.1.5.5.1	Overview	68
7.1.5.5.2	Function Documentation	68
7.1.5.5.2.1	ft_control_rotary_get_direction	68
7.1.5.5.2.2	ft_control_rotary_get_invalid_position	69
7.1.5.5.2.3	ft_control_rotary_get_position	70
7.1.5.5.2.4	ft_control_rotary_is_touched	70
7.1.5.5.2.5	ft_control_rotary_movement_detected	71
7.1.5.5.2.6	ft_control_rotary_register_callback	72
7.1.6	Slider control	74
7.1.6.1	Overview	74
7.1.6.2	Typedef Documentation	75
7.1.6.2.1	ft_control_slider_callback	75
7.1.6.3	Enumeration Type Documentation	75
7.1.6.3.1	ft_control_slider_event	75
7.1.6.4	Variable Documentation	75
7.1.6.4.1	ft_control_slider_interface	75
7.1.6.5	Slider Control API	76
7.1.6.5.1	Overview	76
7.1.6.5.2	Function Documentation	76
7.1.6.5.2.1	ft_control_slider_get_direction	76
7.1.6.5.2.2	ft_control_slider_get_invalid_position	77
7.1.6.5.2.3	ft_control_slider_get_position	78
7.1.6.5.2.4	ft_control_slider_is_touched	78
7.1.6.5.2.5	ft_control_slider_movement_detected	79
7.1.6.5.2.6	ft_control_slider_register_callback	80
7.2	Controls	82
7.2.1	Overview	82
7.2.2	General API	83
7.2.2.1	Overview	83
7.2.2.2	Data Structure Documentation	83
7.2.2.2.1	union ft_control_params	83
7.2.2.2.2	struct ft_control	84
7.2.2.3	API Functions	86
7.2.2.3.1	Overview	86
7.2.2.3.2	Function Documentation	86
7.2.2.3.2.1	ft_control_count_electrodes	86
7.2.2.3.2.2	ft_control_disable	86
7.2.2.3.2.3	ft_control_enable	87
7.2.2.3.2.4	ft_control_get_electrode	88

Section number	Title	Page
7.2.2.3.2.5	ft_control_get_electrodes_state	88
7.2.2.3.2.6	ft_control_get_touch_button	89
7.3	Electrodes	91
7.3.1	Overview	91
7.3.2	Data Structure Documentation	92
7.3.2.1	struct ft_electrode_status	92
7.3.2.2	struct ft_electrode	92
7.3.3	Enumeration Type Documentation	94
7.3.3.1	ft_electrode_state	94
7.3.4	API Functions	95
7.3.4.1	Overview	95
7.3.4.2	Function Documentation	95
7.3.4.2.1	ft_electrode_disable	95
7.3.4.2.2	ft_electrode_enable	96
7.3.4.2.3	ft_electrode_get_last_status	96
7.3.4.2.4	ft_electrode_get_last_time_stamp	97
7.3.4.2.5	ft_electrode_get_raw_signal	97
7.3.4.2.6	ft_electrode_get_signal	98
7.3.4.2.7	ft_electrode_get_time_offset	99
7.4	Filters	100
7.4.1	Overview	100
7.4.2	Data Structure Documentation	100
7.4.2.1	struct ft_filter_fbutt	100
7.4.2.2	struct ft_filter_iir	101
7.4.2.3	struct ft_filter_dctracker	101
7.4.2.4	struct ft_filter_moving_average	102
7.4.3	Macro Definition Documentation	102
7.4.3.1	FT_FILTER_MOVING_AVERAGE_MAX_ORDER	102
7.4.4	Advanced Filtering and Integrating Detection	103
7.4.4.1	Overview	103
7.4.4.2	Data Structure Documentation	103
7.4.4.2.1	struct ft_keydetector_afid_asc	103
7.4.4.2.2	struct ft_keydetector_afid	104
7.4.4.3	Macro Definition Documentation	105
7.4.4.3.1	FT_KEYDETECTOR_AFID_ASC_DEFAULT	105
7.4.4.4	Variable Documentation	106
7.4.4.4.1	ft_keydetector_afid_interface	106
7.5	Key Detectors	107
7.5.1	Overview	107
7.5.2	GPIO module	108
7.5.2.1	Overview	108
7.5.2.2	Data Structure Documentation	108

Section number	Title	Page
7.5.2.2.1	struct ft_module_gpio_user_interface	108
7.5.2.2.1.1	Field Documentation	109
7.5.2.2.1.1.1	get_pin_value	109
7.5.2.2.1.1.2	init_timer	109
7.5.2.2.1.1.3	set_pin_default_state	110
7.5.2.2.1.1.4	set_pin_high	110
7.5.2.2.1.1.5	set_pin_input	110
7.5.2.2.1.1.6	set_pin_low	110
7.5.2.2.1.1.7	set_pin_output	110
7.5.2.2.1.1.8	start_timer	110
7.5.2.2.1.1.9	stop_timer	110
7.5.2.2.1.1.10	timer_get_counter	110
7.5.2.2.1.1.11	timer_get_overrun	110
7.5.2.2.1.1.12	timer_reset_counter	110
7.5.2.2.2	struct ft_module_gpio_params	110
7.5.2.3	Variable Documentation	111
7.5.2.3.1	ft_module_gpio_interface	111
7.5.3	GPIO interrupt module	112
7.5.3.1	Overview	112
7.5.3.2	Data Structure Documentation	112
7.5.3.2.1	struct ft_module_gpoint_user_interface	112
7.5.3.2.1.1	Field Documentation	113
7.5.3.2.1.1.1	clear_pin_interrupt	113
7.5.3.2.1.1.2	init_pin	113
7.5.3.2.1.1.3	init_timer	114
7.5.3.2.1.1.4	set_pin_high	114
7.5.3.2.1.1.5	set_pin_input	114
7.5.3.2.1.1.6	set_pin_interrupt	114
7.5.3.2.1.1.7	set_pin_low	114
7.5.3.2.1.1.8	set_pin_output	114
7.5.3.2.1.1.9	start_timer	114
7.5.3.2.1.1.10	stop_timer	114
7.5.3.2.1.1.11	timer_get_counter	114
7.5.3.2.1.1.12	timer_reset_counter	114
7.5.3.2.2	struct ft_module_gpoint_params	114
7.5.3.3	Function Documentation	115
7.5.3.3.1	ft_module_gpoint_isr	115
7.5.3.3.2	ft_module_gpoint_overflow_isr	116
7.5.3.4	Variable Documentation	117
7.5.3.4.1	ft_module_gpoint_interface	117
7.5.4	TSI module	118
7.5.4.1	Overview	118
7.5.4.2	Data Structure Documentation	118
7.5.4.2.1	struct ft_module_tsi_noise	118
7.5.4.2.2	struct ft_module_tsi_params	119

Section number	Title	Page
7.5.4.3	Variable Documentation	120
7.5.4.3.1	ft_module_tsi_interface	120
7.6	Modules	121
7.6.1	Overview	121
7.6.2	General API	122
7.6.2.1	Overview	122
7.6.2.2	Data Structure Documentation	122
7.6.2.2.1	union ft_module_params	122
7.6.2.2.2	struct ft_module	124
7.6.2.3	Enumeration Type Documentation	126
7.6.2.3.1	ft_module_flags	126
7.6.2.3.2	ft_module_mode	126
7.6.2.4	API Functions	127
7.6.2.4.1	Overview	127
7.6.2.4.2	Function Documentation	127
7.6.2.4.2.1	ft_module_change_mode	127
7.6.2.4.2.2	ft_module_load_configuration	128
7.6.2.4.2.3	ft_module_recalibrate	129
7.6.2.4.2.4	ft_module_save_configuration	130
7.7	System	131
7.7.1	Overview	131
7.7.2	Data Structure Documentation	132
7.7.2.1	struct ft_system	132
7.7.3	Typedef Documentation	133
7.7.3.1	ft_error_callback	133
7.7.3.2	ft_system_callback	133
7.7.4	Enumeration Type Documentation	134
7.7.4.1	ft_system_event	134
7.7.5	API Functions	135
7.7.5.1	Overview	135
7.7.5.2	Function Documentation	135
7.7.5.2.1	ft_error_register_callback	135
7.7.5.2.2	ft_init	136
7.7.5.2.3	ft_mem_get_free_size	137
7.7.5.2.4	ft_system_get_time_counter	138
7.7.5.2.5	ft_system_register_callback	138
7.7.5.2.6	ft_task	139
7.7.5.2.7	ft_trigger	140
7.8	General Types	142
7.8.1	Overview	142
7.8.2	Macro Definition Documentation	143
7.8.2.1	FT_ASSERT	143

Section number	Title	Page
7.8.2.2	FT_DEBUG	143
7.8.2.3	FT_FLAGS_SPECIFIC_SHIFT	143
7.8.2.4	FT_FLAGS_SYSTEM_SHIFT	143
7.8.2.5	FT_FREEMASTER_SUPPORT	143
7.8.2.6	NULL	143
7.8.3	Enumeration Type Documentation	143
7.8.3.1	ft_result	143
7.8.4	Analog Rotary Control	144
7.8.4.1	Overview	144
7.8.4.2	Data Structure Documentation	145
7.8.4.2.1	struct ft_control_arotary_temp_data	145
7.8.4.2.2	struct ft_control_arotary_data	146
7.8.4.3	Macro Definition Documentation	147
7.8.4.3.1	FT_AROTARY_INVALID_POSITION_VALUE	147
7.8.4.4	Enumeration Type Documentation	147
7.8.4.4.1	ft_control_arotary_flags	147
7.8.5	Analog Slider Control	148
7.8.5.1	Overview	148
7.8.5.2	Data Structure Documentation	149
7.8.5.2.1	struct ft_control_aslider_data	149
7.8.5.2.2	struct ft_control_aslider_temp_data	150
7.8.5.3	Macro Definition Documentation	150
7.8.5.3.1	FT_ASLIDER_INVALID_POSITION_VALUE	150
7.8.5.4	Enumeration Type Documentation	151
7.8.5.4.1	ft_control_aslider_flags	151
7.8.6	Keypad Control	152
7.8.6.1	Overview	152
7.8.6.2	Data Structure Documentation	153
7.8.6.2.1	struct ft_control_keypad_data	153
7.8.6.3	Enumeration Type Documentation	155
7.8.6.3.1	ft_control_keypad_flags	155
7.8.7	Rotary Control	156
7.8.7.1	Overview	156
7.8.7.2	Data Structure Documentation	157
7.8.7.2.1	struct ft_control_rotary_data	157
7.8.7.3	Enumeration Type Documentation	158
7.8.7.3.1	ft_control_rotary_flags	158
7.8.8	Slider Control	159
7.8.8.1	Overview	159
7.8.8.2	Data Structure Documentation	160
7.8.8.2.1	struct ft_control_slider_data	160
7.8.8.3	Enumeration Type Documentation	161
7.8.8.3.1	ft_control_slider_flags	161
7.9	Controls	162

Section number	Title	Page
7.9.1	Overview	162
7.9.2	General API	163
7.9.2.1	Overview	163
7.9.2.2	Data Structure Documentation	163
7.9.2.2.1	union ft_control_special_data	163
7.9.2.2.2	struct ft_control_data	165
7.9.2.2.3	struct ft_control_interface	166
7.9.2.2.3.1	Field Documentation	167
7.9.2.2.3.1.1	init	167
7.9.2.2.3.1.2	name	167
7.9.2.2.3.1.3	process	167
7.9.2.3	Enumeration Type Documentation	167
7.9.2.3.1	ft_control_flags	167
7.9.2.4	API Functions	168
7.9.2.4.1	Overview	168
7.9.2.4.2	Function Documentation	168
7.9.2.4.2.1	_ft_control_check_data	168
7.9.2.4.2.2	_ft_control_check_edge_electrodes	169
7.9.2.4.2.3	_ft_control_check_neighbours_electrodes	169
7.9.2.4.2.4	_ft_control_clear_flag	169
7.9.2.4.2.5	_ft_control_clear_flag_all_elec	170
7.9.2.4.2.6	_ft_control_get_data	170
7.9.2.4.2.7	_ft_control_get_electrode	172
7.9.2.4.2.8	_ft_control_get_electrodes_state	172
7.9.2.4.2.9	_ft_control_get_first_elec_touched	174
7.9.2.4.2.10	_ft_control_get_flag	174
7.9.2.4.2.11	_ft_control_get_last_elec_touched	174
7.9.2.4.2.12	_ft_control_get_touch_count	174
7.9.2.4.2.13	_ft_control_init	174
7.9.2.4.2.14	_ft_control_overrun	176
7.9.2.4.2.15	_ft_control_set_flag	177
7.9.2.4.2.16	_ft_control_set_flag_all_elec	177
7.10	Electrodes	178
7.10.1	Overview	178
7.10.2	Data Structure Documentation	179
7.10.2.1	union ft_electrode_special_data	179
7.10.2.2	struct ft_electrode_data	179
7.10.3	Enumeration Type Documentation	181
7.10.3.1	ft_electrode_flags	181
7.10.4	API Functions	182
7.10.4.1	Overview	182
7.10.4.2	Function Documentation	183
7.10.4.2.1	_ft_electrode_clear_flag	183
7.10.4.2.2	_ft_electrode_get_data	183

Section number	Title	Page
7.10.4.2.3	_ft_electrode_get_delta	184
7.10.4.2.4	_ft_electrode_get_flag	185
7.10.4.2.5	_ft_electrode_get_index_from_module	185
7.10.4.2.6	_ft_electrode_get_last_status	185
7.10.4.2.7	_ft_electrode_get_last_time_stamp	186
7.10.4.2.8	_ft_electrode_get_raw_signal	187
7.10.4.2.9	_ft_electrode_get_shield	187
7.10.4.2.10	_ft_electrode_get_signal	188
7.10.4.2.11	_ft_electrode_get_status	189
7.10.4.2.12	_ft_electrode_get_time_offset	189
7.10.4.2.13	_ft_electrode_get_time_offset_period	190
7.10.4.2.14	_ft_electrode_get_time_stamp	191
7.10.4.2.15	_ft_electrode_init	191
7.10.4.2.16	_ft_electrode_is_touched	192
7.10.4.2.17	_ft_electrode_normalization_process	193
7.10.4.2.18	_ft_electrode_set_flag	194
7.10.4.2.19	_ft_electrode_set_raw_signal	194
7.10.4.2.20	_ft_electrode_set_signal	195
7.10.4.2.21	_ft_electrode_set_status	195
7.10.4.2.22	_ft_electrode_shielding_process	196
7.11	Filters	198
7.11.1	Overview	198
7.11.2	Data Structure Documentation	198
7.11.2.1	struct ft_filter_fbutt_data	198
7.11.2.2	struct ft_filter_moving_average_data	199
7.11.3	Enumeration Type Documentation	200
7.11.3.1	ft_filter_state	200
7.11.4	API Functions	201
7.11.4.1	Overview	201
7.11.4.2	Function Documentation	201
7.11.4.2.1	_ft_abs_int32	201
7.11.4.2.2	_ft_filter_abs	202
7.11.4.2.3	_ft_filter_deadrange_u	202
7.11.4.2.4	_ft_filter_fbutt_init	203
7.11.4.2.5	_ft_filter_fbutt_process	204
7.11.4.2.6	_ft_filter_iir_process	204
7.11.4.2.7	_ft_filter_is_deadrange_u	205
7.11.4.2.8	_ft_filter_limit_u	205
7.11.4.2.9	_ft_filter_moving_average_init	206
7.11.4.2.10	_ft_filter_moving_average_process	206
7.11.5	Advanced Filtering and Integrating Detection	208
7.11.5.1	Overview	208
7.11.5.2	Data Structure Documentation	208
7.11.5.2.1	struct ft_keydetector_afid_asc_data	208

Section number	Title	Page
7.11.5.2.2	struct ft_keydetector_afid_data	209
7.11.5.3	Macro Definition Documentation	211
7.11.5.3.1	FT_KEYDETECTOR_AFID_INITIAL_INTEGRATOR_VALUE	211
7.11.5.3.2	FT_KEYDETECTOR_AFID_INITIAL_RESET_RELEASE_COUNTER_- _VALUE	211
7.11.5.3.3	FT_KEYDETECTOR_AFID_INITIAL_RESET_TOUCH_COUNTER_- VALUE	211
7.12	Key Detectors	212
7.12.1	Overview	212
7.12.2	Data Structure Documentation	212
7.12.2.1	union ft_keydetector_data	212
7.12.3	GPIO module	214
7.12.3.1	Overview	214
7.12.3.2	Data Structure Documentation	214
7.12.3.2.1	struct ft_module_gpio_data	214
7.12.4	GPIO interrupt module	216
7.12.4.1	Overview	216
7.12.4.2	Data Structure Documentation	216
7.12.4.2.1	struct ft_module_gpoint_data	216
7.12.5	TSI module	218
7.12.5.1	Overview	218
7.12.5.2	Data Structure Documentation	218
7.12.5.2.1	struct ft_module_tsi_noise_data	218
7.12.5.2.2	struct ft_module_tsi_data	219
7.12.5.3	Macro Definition Documentation	220
7.12.5.3.1	FT_TSI_NOISE_INITIAL_TOUCH_THRESHOLD	220
7.12.5.3.2	FT_TSI_NOISE_TOUCH_RANGE	220
7.12.5.4	Enumeration Type Documentation	220
7.12.5.4.1	ft_module_tsi_flags	220
7.13	Modules	221
7.13.1	Overview	221
7.13.2	General API	222
7.13.2.1	Overview	222
7.13.2.2	Data Structure Documentation	222
7.13.2.2.1	union ft_module_special_data	222
7.13.2.2.2	struct ft_module_data	223
7.13.2.2.3	struct ft_module_interface	225
7.13.2.2.3.1	Field Documentation	226
7.13.2.2.3.1.1	change_mode	226
7.13.2.2.3.1.2	electrode_disable	226
7.13.2.2.3.1.3	electrode_enable	226
7.13.2.2.3.1.4	init	226
7.13.2.2.3.1.5	load_configuration	226

Section number	Title	Page
7.13.2.2.3.1.6	name	226
7.13.2.2.3.1.7	process	226
7.13.2.2.3.1.8	recalibrate	226
7.13.2.2.3.1.9	save_configuration	226
7.13.2.2.3.1.10	trigger	226
7.13.2.3	API functions	227
7.13.2.3.1	Overview	227
7.13.2.3.2	Function Documentation	227
7.13.2.3.2.1	_ft_module_clear_flag	227
7.13.2.3.2.2	_ft_module_get_data	228
7.13.2.3.2.3	_ft_module_get_flag	229
7.13.2.3.2.4	_ft_module_get_instance	230
7.13.2.3.2.5	_ft_module_get_mode	230
7.13.2.3.2.6	_ft_module_init	230
7.13.2.3.2.7	_ft_module_process	231
7.13.2.3.2.8	_ft_module_set_flag	232
7.13.2.3.2.9	_ft_module_set_mode	233
7.13.2.3.2.10	_ft_module_trigger	234
7.14	FreeMASTER support	236
7.14.1	Overview	236
7.14.2	API functions	237
7.14.2.1	Overview	237
7.14.2.2	Function Documentation	237
7.14.2.2.1	_ft_freemaster_add_variable	237
7.14.2.2.2	_ft_freemaster_init	238
7.15	Memory Management	239
7.15.1	Overview	239
7.15.2	Data Structure Documentation	239
7.15.2.1	struct ft_mem	239
7.15.3	API functions	241
7.15.3.1	Overview	241
7.15.3.2	Function Documentation	241
7.15.3.2.1	_ft_mem_alloc	241
7.15.3.2.2	_ft_mem_deinit	242
7.15.3.2.3	_ft_mem_init	242

Chapter 8 Freescale Touch Private API

8.1	Overview	245
8.2	System	247

Section number	Title	Page
8.2.1	Overview	247
8.2.2	Data Structure Documentation	247
8.2.2.1	struct ft_kernel	247
8.2.3	Enumeration Type Documentation	249
8.2.3.1	ft_system_control_call	249
8.2.3.2	ft_system_module_call	249
8.2.4	API Functions	250
8.2.4.1	Overview	250
8.2.4.2	Function Documentation	250
8.2.4.2.1	_ft_system_control_function	250
8.2.4.2.2	_ft_system_get	251
8.2.4.2.3	_ft_system_get_module	253
8.2.4.2.4	_ft_system_get_time_offset	254
8.2.4.2.5	_ft_system_get_time_offset_from_period	255
8.2.4.2.6	_ft_system_get_time_period	256
8.2.4.2.7	_ft_system_increment_time_counter	257
8.2.4.2.8	_ft_system_init	257
8.2.4.2.9	_ft_system_invoke_callback	258
8.2.4.2.10	_ft_system_module_function	259
8.2.4.2.11	_ft_system_modules_data_ready	261
8.2.4.2.12	ft_error	261

Chapter 9 Data Structure Documentation

9.0.5	ft_keydetector_interface Struct Reference	263
9.0.5.1	Detailed Description	263
9.0.5.2	Field Documentation	263
9.0.5.2.1	ft_keydetector_init	263
9.0.5.2.2	ft_keydetector_measure	264
9.0.5.2.3	ft_keydetector_process	264
9.0.5.2.4	name	264

Chapter 1 Introduction

This document describes the Freescale software library for implementing the touch-sensing applications on Freescale MCU platforms. The touch-sensing algorithms contained in the library utilize either the dedicated touch-sensing interface (TSI) module available on most of the Freescale Kinetis MCUs, or the generic-pin I/O module to detect finger touch, movement, or gestures. Please read the license agreement document for terms and conditions, under which you can use the software. **Thank you for choosing the Freescale solution!**

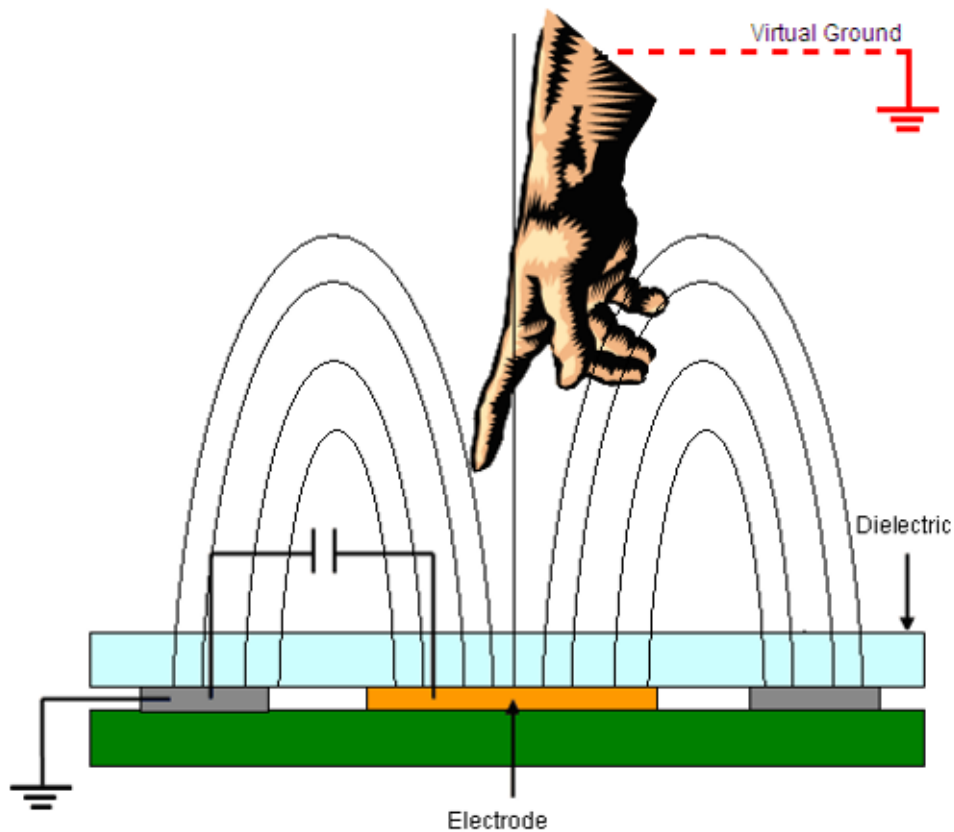


Figure 1: Capacitive Touch Sensing principle

The touch-detection algorithms and operation principles are based on the Freescale TSS library version 3.1 released in 2013, which is still available for download at www.freescale.com. However, the code in this Freescale Touch library was completely rewritten to simplify the application coding and to make better use of the 32-bit Freescale MCU features. Unlike the TSS 3.1, the new Freescale Touch (FT) library does not use pre-processor configuration macros, but it uses plain C data types to configure [Electrodes](#), [Modules](#),

and [Controls](#). The library code is also more suitable for use in RTOS-based multi-tasking applications and in the C++ object-oriented applications.

The Freescale Touch library is provided in both the binary library form and the source code form. When used as a binary library, you must link your code to the library statically, and use the library header files to make use of the [Freescale Touch User API](#). When using the source code form, you can put the source files directly into the application project for easy use and debugging. You can also use the source files to build your own version of a statically-linked library.

In any case, your application code uses the same library API and data types to configure, initialize, and use the touch-sensing algorithms implemented in the library. All the steps needed to successfully use the library are described in this document.

Further Reading

You can find more details on using the Freescale Touch software library in the following sections:

- [Key Library Elements](#) - Describes the main building blocks of the Freescale Touch library
- [Your First Application](#) - Helps you to create your first Freescale Touch application
- [Configuring the Library](#) - Explains the configuration structures that must be defined in the application
- [Freescale Touch User API](#) - Reference manual describing structures and functions you must use when building the touch-sensing applications Further divided into subsections:
 - [System](#) - The system contains general functionality for the whole FT and basic FT API user interface.
 - [Modules](#) - The hardware interaction system that secures and gets the raw data from hardware, and handles it over key detectors to the electrode structure.
 - [Key Detectors](#) - The algorithms that secure recognition of touch-sensing events (Touch and Release) in input raw data from modules.
 - [Electrodes](#) - The basic data container for the individual electrode data, and some basic electrode API.
 - [Controls](#) - The high-level layer of FT that transforms the captured data into logical controls like keypad, slider, and so on.

Chapter 2

Key Library Elements

This section explains the key building blocks of the Freescale Touch Library that are used in the user application.

The Freescale Touch library is based on a layered architecture with data types resembling an object-oriented approach, but still implemented in a plain C language. The basic building blocks are outlined in the figure below. Each of the blocks is further described in the **Reference** section.

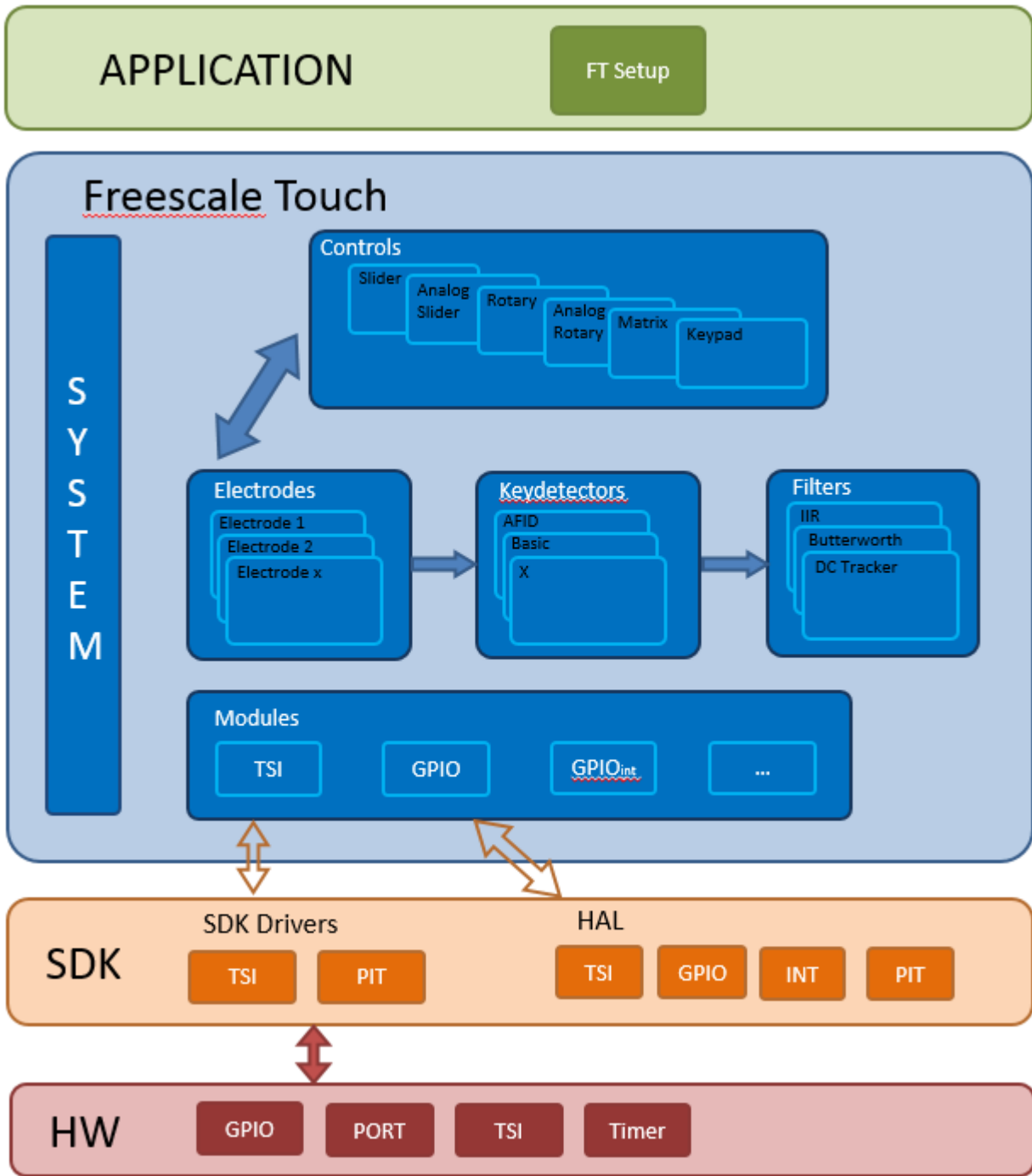


Figure 2: Object structure

- **System (System API)** - The general code of the FT, providing basic functionality for user application.
- **Modules (Modules API)** - The modules are designed for hardware interaction and gathering capacitance data.
- **Key Detectors (Key Detectors API)** - The key detectors are designed to recognize the touch / release events in the data measured by modules.

- **Controls** (**Controls** API) - The high-level objects representing the so-called “Decoder”, and they are used to create the virtual devices (controls) from the individual electrodes.
- **Electrodes** (**Electrodes** API) - The electrode is a basic data container with functionality for general control of individual electrodes.

2.1 System

The main object encapsulating lists of other key objects used in the Freescale Touch system, such as the measurement modules and controls. There is always only one active instance of the System object in the application. The System function API covers initialization, global timing, and uniform access to touch detection modules.

2.2 Modules

This section describes different **Modules** supported by the Freescale Touch library.

The modules are part of the FT code, securing the gathering of raw data from different hardware sources, such as GPIO pins or the Freescale TSI peripheral. The GPIO methods are implemented in two different ways: the first is using the standard polling system, and the second is interrupt-driven. The TSI module describes the hardware configuration and control of elementary functionality of the TSI peripheral; it covers all versions of the TSI peripheral using the generic low-level driver API. The TSI module is designed for processors that have the hardware TSI module version 1, 2 or 4 (for example Kinetis L). The module also handles the NOISE mode supported by TSI v4 (Kinetis L). All the different modules are implemented using the same API function. You do not need to take care about the differences between individual TSI versions. Basically, all modules behave like TSI without any difference.

2.2.1 Module Types

The FT library defines objects of modules compound of the following division:

- Modules divided according to hardware used
 - **TSI module** - TSI module. This module gathers physical electrode capacitance data from the TSI peripheral. It is based on the KSDK TSI driver. The TSI module also contains a simple key detector for the noise mode (if it is enabled and running on the TSI v4 peripheral version). (**Noise** - Special noise mode is a hardware feature of the TSI module version 4. This module is implemented mainly in the Kinetis L family of MCUs.)
 - **GPIO module** - GPIO module. This module gathers physical electrode capacitance data using a simple GPIO toggle-pin-polling method.
 - **GPIO interrupt module** - GPIOINT module. This module gathers physical electrode capacitance data using a simple GPIO toggle pin interrupt-driven method.

2.2.2 TSI details

The Touch-Sensing Input (TSI) module provides a capacitive touch-sensing detection with high sensitivity and enhanced robustness. Each TSI pin implements the capacitive measurement of an electrode having

Modules

individual result registers. The TSI module can be functional in several low-power modes with an ultra-low current adder. Freescale currently provides two versions of TSI modules. The first version of the TSI module is implemented in the Coldfire+ and ARM®Cortex®-M4 MCUs. This TSI module can wake the CPU up in case of a touch event, measure all enabled electrodes in one automatic cycle, and provide automatic triggering inside the module. For more details about the TSI module features, see the arbitrary reference manual of the MCU containing the TSI module inside. Figure 5-1 shows a block diagram of the TSI module.

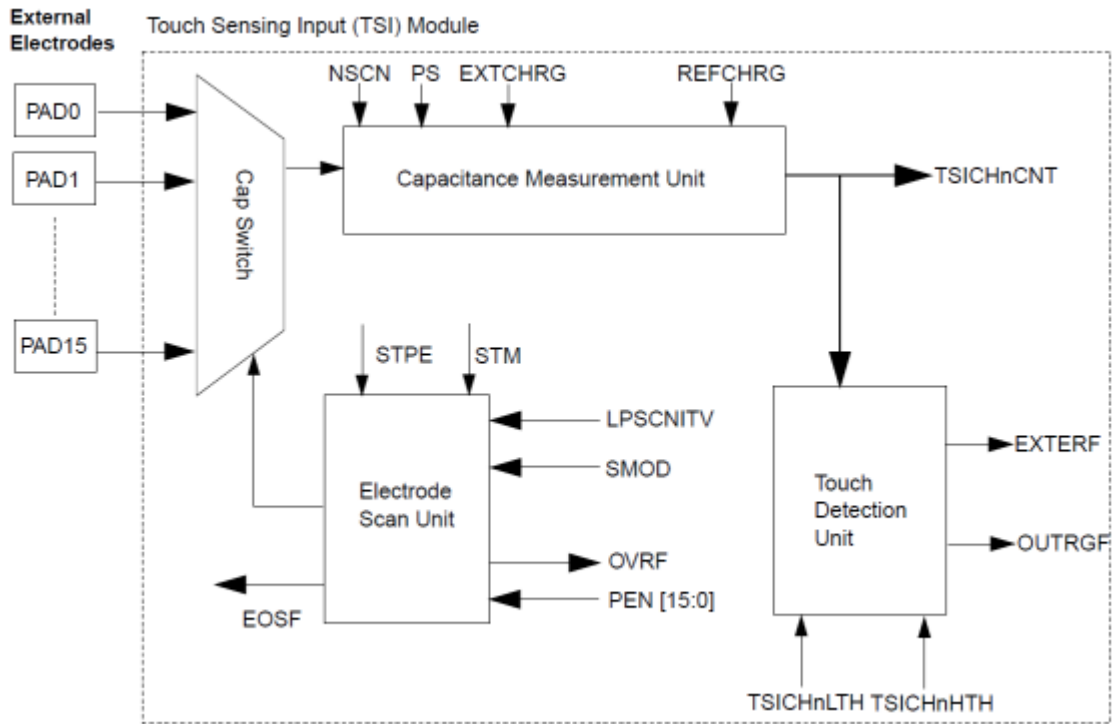


Figure 2.2.1: TSI peripheral version 1 & 2

The second version of the TSI module is a simplified version of the first generation with some additional features. It is currently implemented in the S08PTxx and ARM®Cortex®-M0+ MCUs. This kind of TSI module measures just one enabled electrode in one measurement cycle, and provides automatic triggering externally, using the RTC or LPTMR timer. For more information about the TSI module features, see the reference manual of the MCU containing the TSI module.

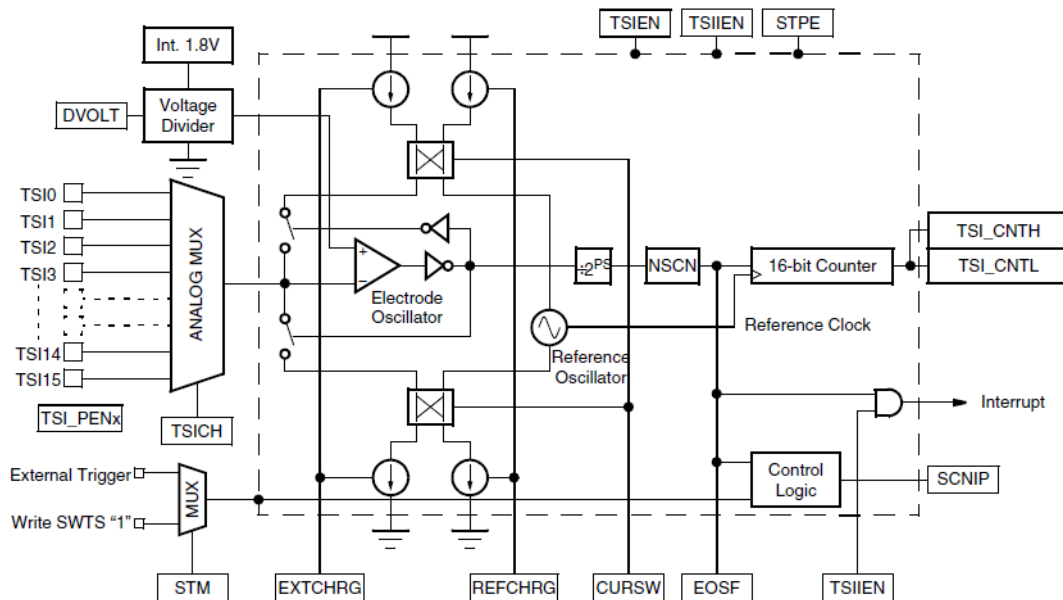


Figure 2.2.2: TSI peripheral version 4

2.3 Key Detectors

This section describes different [Key Detectors](#) supported by the Freescale Touch library. The key detector module determines, whether an electrode has been touched or released, based on the values obtained by the capacitance sensing layer. Along with this detection, the key detector module uses a debounce algorithm that prevents the library from false touches.

2.3.1 Types

The FT library defines objects of keydetectors compound of following division. (In the current version of FT (FT 1.0 GA), only one key detector is available, and it is called AFID.)

- AFID** - The AFID (Advanced Filtering and Integrating Detection) key detector operates using two IIR filters with different depths (one being short / fast, the other being long / slow) and, then, integrating the difference between the two filtered signals. Although this algorithm is more immune to noise, it is not compatible with other noise-cancellation techniques, such as shielding. The AFID key detector can be selected manually in the FT configuration. The key detector also provides automatic sensitivity calibration. The calibration periodically adjusts the level of electrode sensitivity, which is calculated according to the touch-tracking information. Although the sensitivity no longer has to be manually set, the settings are still available for more precise tuning. The standard baseline, which is set according to the low-pass IIR filter, is also calculated for analog decoders and proximity function. A debounce function is implemented in this module to eliminate false detections caused by instantaneous noise.

The main functions of the AFID key detector module are as follows:

Controls

- Two filters with integration for touch detection
- Electrode detection debouncing
- Baseline generation
- IIR filtering of current capacitance signal
- Proximity detection
- Sensitivity autocalibration
- Electrode status reporting
- Fault reporting

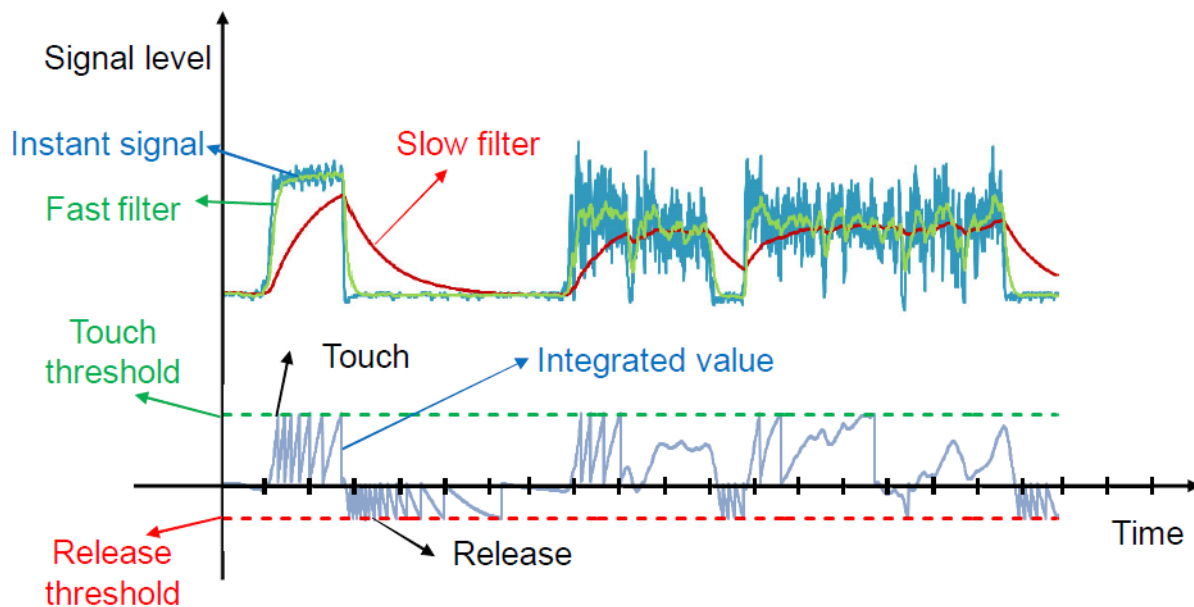


Figure 2.3.1: AFID key detector signal diagram

2.4 Controls

This section describes different [Controls](#) supported by the Freescale Touch library.

Decoders provide the highest level of abstraction in the library. In this layer, the information regarding touched and untouched electrodes is interpreted, and it shows the status of control in a behavioral way. Additional functionalities can be provided by the decoders. Note that the decoder-related code exists only once in the memory, which implies that despite the number of rotary controls in the system, only one rotary decoder resides in the memory. Decoders can be described as classes of an object-oriented language, where each control has a decoder associated to it. Therefore, the control becomes an instance of the decoder (an object). However, not all decoders are necessarily instantiated in every system. The decoder types supported by the library are:

- Rotary
- Slider
- Keypad
- Analog rotary

- Analog slider

Types:

- **Analog Slider Control** The analog slider control is similar to the standard slider, but it works with less electrodes, and the calculated position has a higher resolution. For example, a two-electrode analog slider can provide an analog position in the range of 128. The shape of the electrodes must meet the condition that increases and decreases the signal during the finger movement, which needs to be linear. The figure shows an arrangement of electrodes used for a typical analog slider. The analog slider control provides the following callback events:
 - direction change
 - movement
 - touch
 - release



Figure 2.4.1: Analog slider

- **Slider control** The linear slider control is similar to the rotary slider. The same parameters must be reported in both. The figure shows an arrangement of electrodes used for a typical linear slider. Like the rotary slider, the shape of the electrodes can be changed, but their position must remain as shown in the figure. The slider control provides the following callback events:
 - direction change
 - movement
 - touch
 - release



Figure 2.4.2: Slider

- **Keypad Control** Keypad is a basic configuration for the arranged electrodes shown in the figure, because all that matters is to determine, which one of the electrodes has been touched. The Keypad Decoder is the module handling the boundary checking, controlling the events buffer, and reporting

Controls

of events, depending on the user's configuration. The Keypad Decoder must be used when the application needs the electrodes to behave like keyboard keys. If the user needs to detect movement, another type of decoder must be used. The Keypad decoder is capable of using groups of electrodes, that must be touched simultaneously for reporting the defined key. This allows users to create a control interface with more user inputs than the number of physical electrodes. The slider control provides the following callback events:

- touch
- release

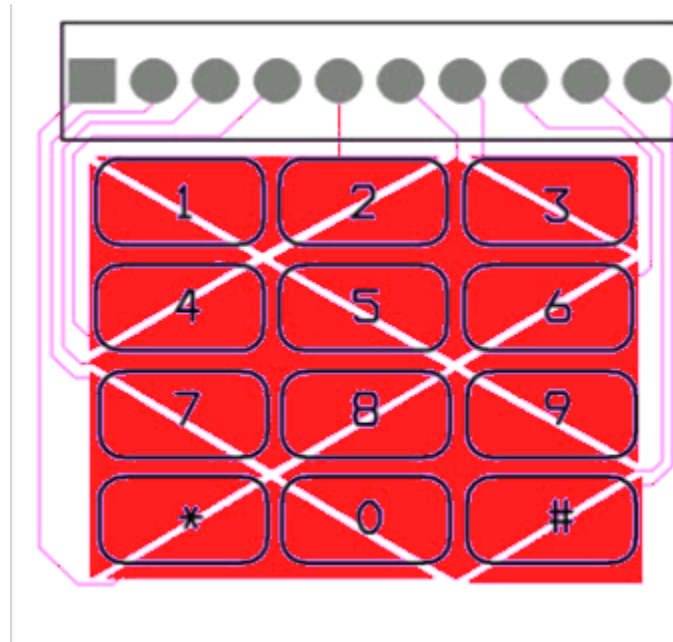


Figure 2.4.3: Keypad

- **Rotary Control** Capacitive sensors provide an opportunity to control a device, such as a potentiometer. To achieve this, you must use a special electrode configuration. The figure shows the electrode configuration needed to implement a rotary slider. The shape of the electrodes can vary, but the configuration must stay the same. In other words, the electrodes intended to form a rotary slider must be placed one after another, forming a circle. The rotary control provides the following callback events:
 - direction change
 - movement
 - touch
 - release

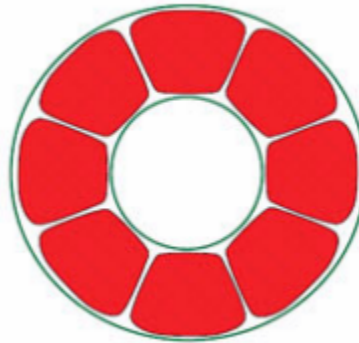


Figure 2.4.4: Rotary

- **Analog Rotary Control** The analog rotary control is similar to the standard rotary control, but with less electrodes, and the calculated position has a higher resolution. For example, a four-electrode analog rotary control can provide an analog position in the range of 64. The shape of the electrodes must meet the condition that increases and decreases the signal during the finger movement, which must be linear. The figure shows an arrangement of electrodes used for a typical analog rotary control. The configuration must be the same. In other words, the electrodes intended to form a rotary slider must be placed one after another, forming a circle. The analog rotary control provides the following callback events:
 - direction change
 - movement
 - touch
 - release

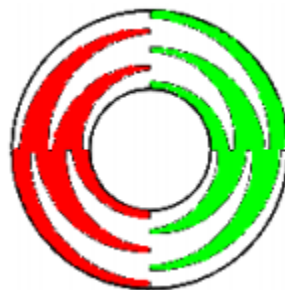


Figure 2.4.5: Analog rotary

2.5 Electrodes

Electrodes are data objects that are used by data acquisition algorithms to store per-electrode data, as well as the resulting signal and touch / timestamp information. Each Electrode provides signal value information. The baseline value and the touch / timestamp buffer contain the time of last few touch and release events. The electrode contains information about the key detector used to detect touches for this physical electrode. This brings the advantage that each electrode has its own key detector settings, independent on the module used. It contains information about the hardware pin, immediate touch status, and time stamps

Electrodes

of the last few touch or release events. The private electrodes API provide all the functionality needed to handle the private needs of the Freescale Touch Library.

It is you as the application programmer and Freescale Touch Library user who specifies what [Modules](#) and [Controls](#) will be instantiated in the system, and what electrodes will be serviced by each module. See more details about various modules and their [configuration](#) in this document.

Chapter 3

Directory Structure

Freescall Touch library is organized into the following files and folders:

- [content of the Freescall Touch Installation Folder]
 - **examples** - Ready-to-use example projects. Don't add files from this folder to a custom application project!
 - * **frdm_aslider_app**
 - * **twr_default_app**
 - * **twr_twrpi_app**
 - * ...
 - **freemaster** - The main Freescall Touch FreeMASTER application, which is able to show all the library parts in a real live application
 - * **src** - The FreeMASTER web page source code, including the JavaScript files
 - **ft** - The main Freescall Touch library directory
 - * **include** - Library header files
 - * **source** - Library source files
 - **controls** - Source files of the [decoders](#)
 - **electrodes** - Source files of the [electrodes](#)
 - **filters** - Source files of the filtering algorithms
 - **keydetectors** - Source files of the [key detectors](#)
 - **modules** - Source files of the [modules](#)
 - **system** - Source files of the [system](#) base code

Chapter 4

Configuring the Library

This section describes the library structures that must be initialized to use the Freescale Touch library in the application. Almost all library configuration parameters are passed to the library API in one of the [System](#), [Modules](#), keydetectors, [Electrodes](#), or [Controls](#) structures. The same structures that are used to keep the configuration data for library initialization are then used in the application code to access their runtime properties.

For example: the user initializes the `ft_aslider_control` structure to specify the set of electrodes making up the slider layout, the maximum possible resolution of the slider, and other options. In the application runtime, the code uses a pointer to the same structure to access calculated values such as the finger position.

4.1 Configuration Example

The next sections describe the minimum setup required for each of the library structures.

4.1.1 Key Detectors

Key Detector setup for AFID. Following setup is common in `ft_setup.c`

```
/* Key Detectors */
const struct ft_keydetector_afid keydec_afid =
{
    .signal_filter = 1,
    .fast_signal_filter = {
        .cutoff = 6
    },
    .slow_signal_filter = {
        .cutoff = 2
    },
    .base_avg = {.n2_order = 12},
    .reset_rate = 10,
    .asc = {
        .touch_treshold_fall_rate = 1000,
        .noise_resets_minimum = 256,
        .resets_for_touch = 5,
    },
};
```

4.1.2 Electrodes

Electrode setup. Following setup is common in `ft_setup.c`

```
const struct ft_electrode electrode_0 =
{
```

Configuration Example

```
.pin_input = BOARD_TSI_ELECTRODE_1,
.keydetector_interface = &ft_keydetector_afid_interface,
.keydetector_params.afid = &keydec_afid,
};
```

4.1.3 Modules

This object depends on the type of module. The module setup (including the hardware configuration) is shown below.

4.1.3.1 TSI module

For operation with [ft_module](#) see the code below.

```
const struct ft_electrode * const module_0_electrodes[] = {&electrode_0, &electrode_1,
&electrode_2, &electrode_3, NULL};

const tsi_config_t hw_config =
{
    .ps = kTsiElecOscPrescaler_16div,
    .extchrg = kTsiExtOscChargeCurrent_8uA,
    .refchrg = kTsiRefOscChargeCurrent_16uA,
    .nscn = kTsiConsecutiveScansNumber_32time,
    .lpclks = kTsiLowPowerInterval_100ms,
    .amclks = kTsiActiveClkSource_BusClock,
    .ampsc = kTsiActiveModePrescaler_8div,
    .lpscnitv = kTsiLowPowerInterval_100ms,
    .thresh = 100,
    .thresl = 200,
};

const struct ft_module tsi_module =
{
    .interface = &ft_module_tsi_module_interface,
    .electrodes = &module_0_electrodes[0],
    .config = (void*)&hw_config,
    .instance = 0,
    .module_params = NULL,
};
```

4.1.4 Controls

Control setup - see the description below.

4.1.4.1 Analog slider

For operation using the [ft_control_aslider](#) see the code below.

```
const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
&electrode_2, &electrode_3, NULL};

const struct ft_control_aslider aslider_params =
```



```

{
    .range = 100,
};

const struct ft_control aslider_0 =
{
    .interface = &ft_control_aslider_interface,
    .electrodes = control_0_electrodes,
    .control_params.aslider = &aslider_params,
};

```

4.1.4.2 Slider

For operation using the `ft_control_slider` see the code below.

```

const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
    &electrode_2, &electrode_3, NULL};

const struct ft_control slider_0 =
{
    .interface = &ft_control_slider_interface,
    .electrodes = control_0_electrodes,
};

```

4.1.4.3 Keypad

For operation using the `ft_control_keypad` see the code below.

```

const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
    &electrode_2, &electrode_3, NULL};

const struct ft_control_keypad keypad_params =
{
    .groups = NULL,
    .groups_size = 0,
};

const struct ft_control keypad_0 =
{
    .interface = &ft_control_keypad_interface,
    .electrodes = control_0_electrodes,
    .control_params.keypad = &keypad_params,
};

```

4.1.4.4 Analog rotary

For operation using the `ft_control_arotary` see the code below.

```

const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
    &electrode_2, &electrode_3, NULL};

const struct ft_control_arotary arotary_params =
{
    .range = 100,
};

```

Configuration Example

```
const struct ft_control arotary_0 =
{
    .interface = &ft_control_arotary_interface,
    .electrodes = control_0_electrodes,
    .control_params.arotary = &arotary_params,
};
```

4.1.4.5 Rotary

For operation using the `ft_control_rotary` see the code below.

```
const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
    &electrode_2, &electrode_3, NULL};

const struct ft_control rotary_0 =
{
    .interface = &ft_control_rotary_interface,
    .electrodes = control_0_electrodes,
};
```

4.1.5 System

The [System](#) is represented by the `ft_system` structure. This structure binds together all the other objects, so it must be initialized with the following parameters:

- list of controls
- list of modules (which indirectly provides list of all used electrodes)
- time period and initialization time

```
const struct ft_control * const controls[] = {&aslider_0, NULL};
const struct ft_module * const modules[] = {&tsi_module, NULL};

const struct ft_system system_0 = {
    .controls = &controls[0],
    .modules = &modules[0],
    .time_period = 5,
    .init_time = 50,
};
```

Chapter 5

Your First Application

This chapter shows how to integrate the Freescale Touch library into an existing application project. There are several ways of using the Freescale Touch library. This guide presents the simplest option, where all library files are added to the user application project, and compiled together with the application. The application demonstrated in this chapter uses two electrodes and implements a simple [slider](#) control, which is able to detect finger movement within a linear area.

5.1 Creating the Freescale Touch Application

You can use the library in these two ways:

- Put the library source files directly into the application project (*as described below*)
- Compile the library files into a statically-linked library, and use the library in your application project

5.1.1 Adding Library Files into the Project

The library can be easily integrated into your application by adding the FT source files into your project. See the [Directory Structure](#) section to understand the files and folders of the Freescale Touch library. There are two steps to take:

- The "include" search paths of your project must be extended to cover the directories with public header files (the `/ft/lib/include` folder).
- The source code files must be added into the project (the `/ft/lib/source` folder and all subfolders). Not all source files are always used in the application, but the linker should take care of optimizing the unused code out of the executable.

The library uses startup code, linker files, and some low-level driver code from the Kinetis SDK. This code is not considered to be part of the Freescale Touch library, but it serves as a basis for example applications. You can reuse the SDK linker files and drivers in your custom application also. However, it is better to get the latest SDK version from the Freescale web site.

The figure below shows the typical Freescale Touch application project in the IAR Embedded Workbench IDE:

Creating the Freescale Touch Application

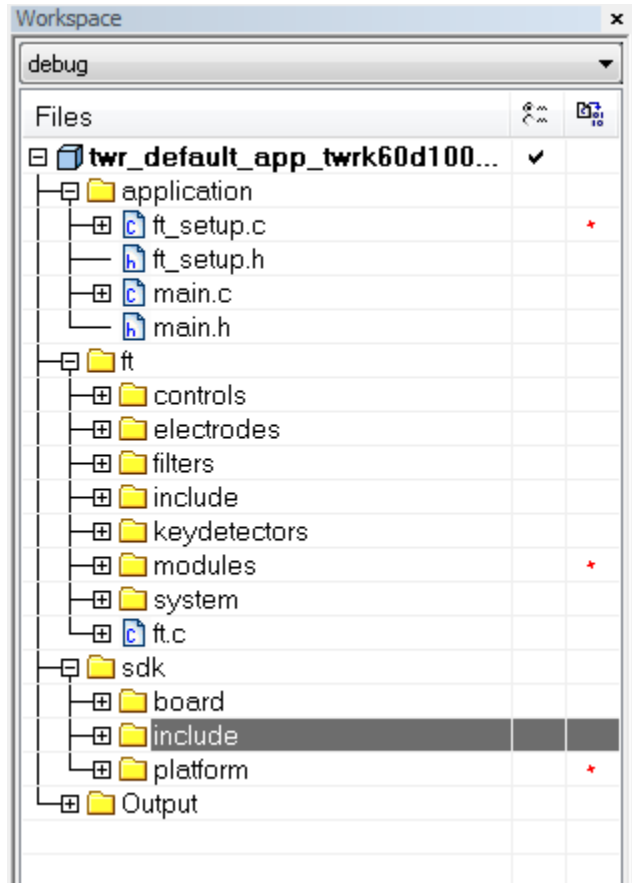


Figure 5.1.1: Workspace directory

5.1.2 Setting 'include' Search Paths

The figure shows how to set up the search paths in the IAR Embedded Workbench IDE. Only one include path is needed from the Freescale Touch point of view. The pre-processor symbols must be defined to identify the CPU and Board for the SDK low-level code. You can find the valid options in the `fsl_device_registers.h` file, located in the `KSDK/platform/CMSIS/Include/device/` directory.

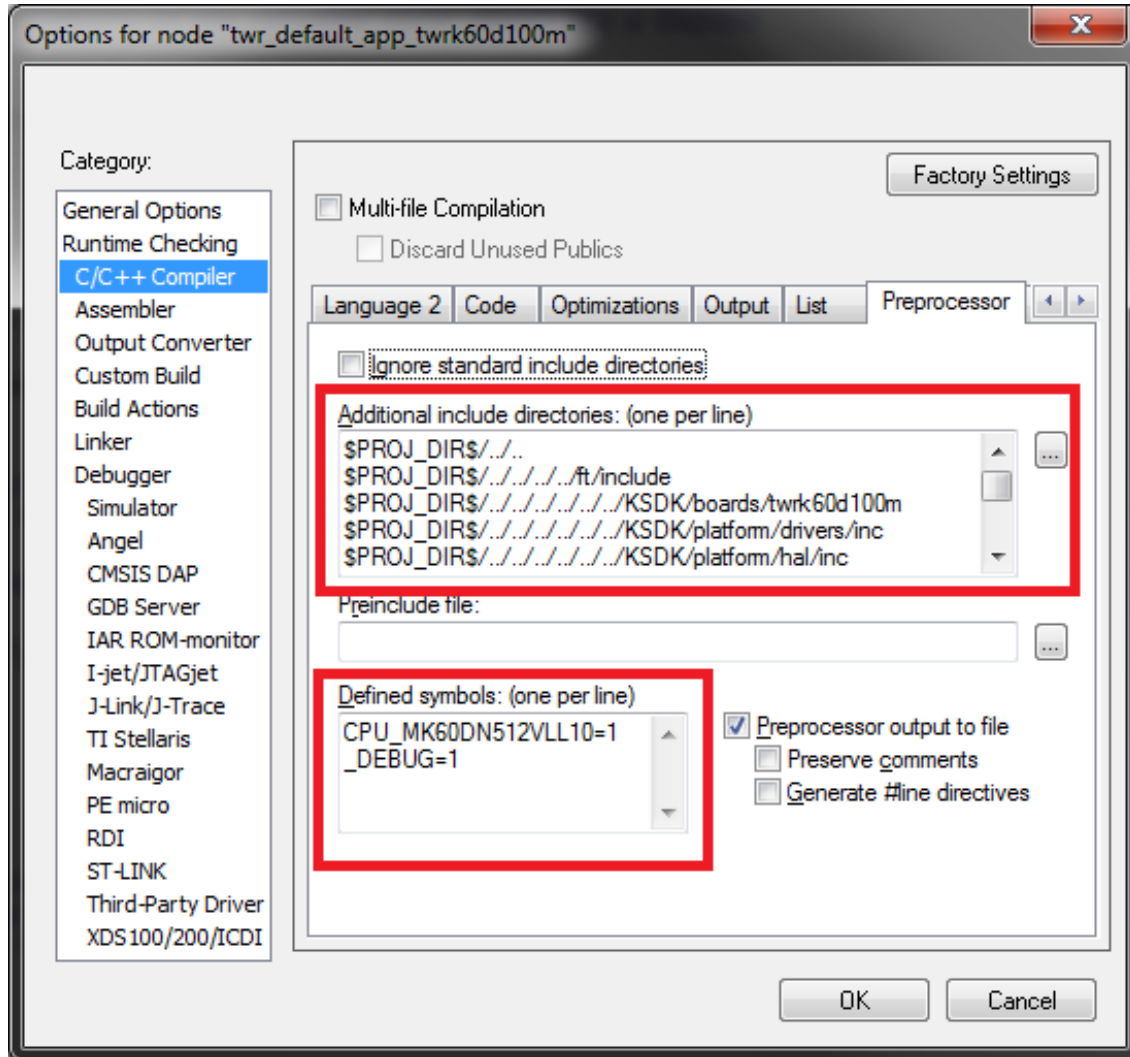


Figure 5.1.2: Include search path

All search paths: Only the first path is mandatory.

```

$PROJ_DIR$/../..
$PROJ_DIR$/../../../../ft/include
$PROJ_DIR$/../../../../KSDK/examples/twrk60d100m
$PROJ_DIR$/../../../../KSDK/platform/drivers/inc
$PROJ_DIR$/../../../../KSDK/platform/hal/inc
$PROJ_DIR$/../../../../KSDK/platform/CMSIS/Include
$PROJ_DIR$/../../../../KSDK/platform/system/inc
$PROJ_DIR$/../../../../KSDK/platform/osa/inc
$PROJ_DIR$/../../../../KSDK/platform/devices
$PROJ_DIR$/../../../../KSDK/platform/devices/MK60D10/startup
$PROJ_DIR$/../../../../KSDK/platform/devices/MK60D10/include
$PROJ_DIR$/../../../../KSDK/platform/utilities/inc
    
```

Creating the Freescale Touch Application

5.1.3 Setting 'linker' path

The figure shows how to set up the linker file in the IAR Embedded Workbench IDE. This is the file reused from the Kinetis SDK. You can use your own linker file to have a full control over the linker process.

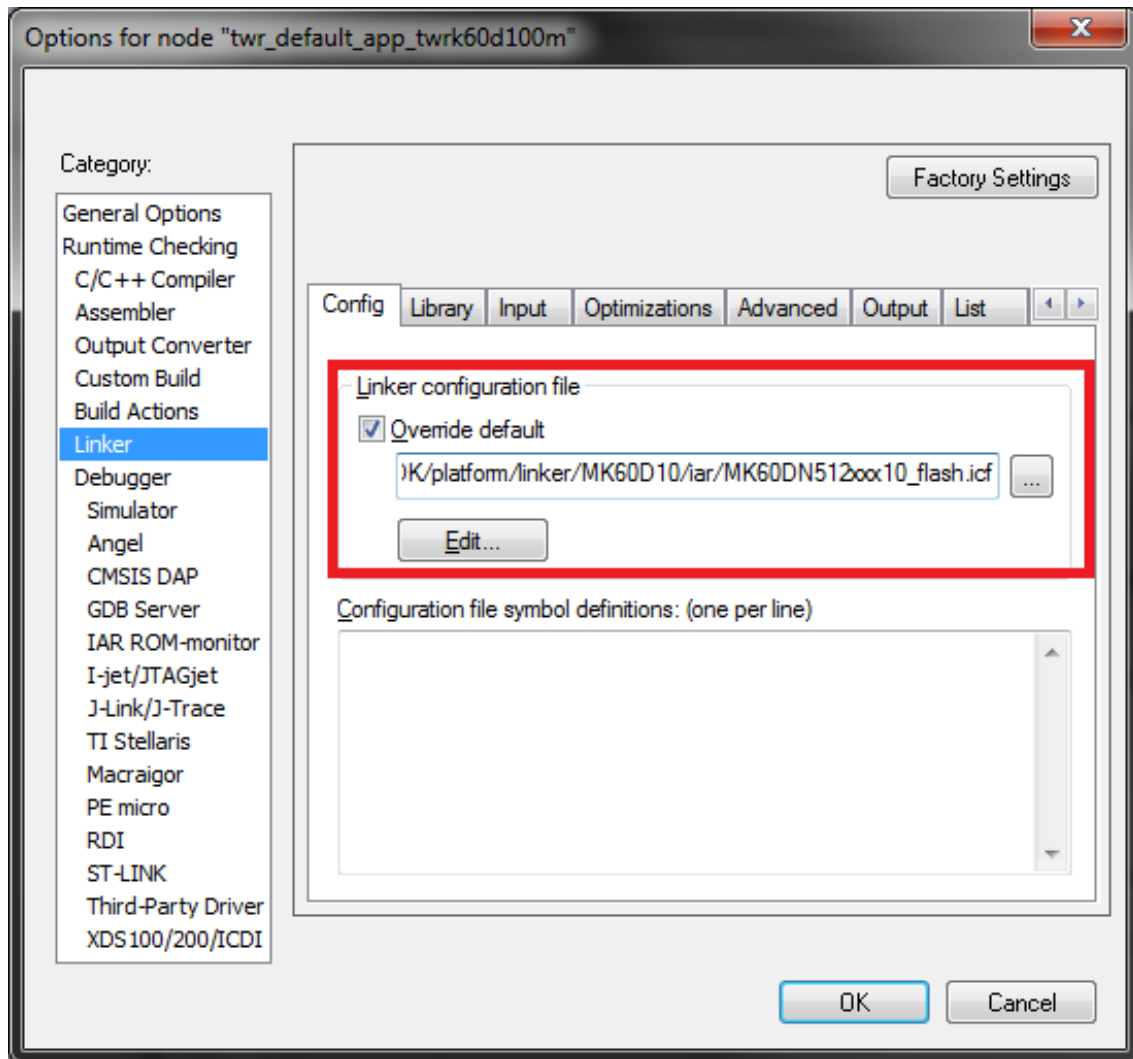


Figure 5.1.3: Linker search path

5.1.4 Application Setup

To define modules, electrodes, controls, and system, you must create initialized instances of the structure types, as described in section [Configuring the Library](#).

The code below shows an example configuration of four electrodes on the TWR K60D100M board.

There are several [key detectors](#) (touch-evaluation algorithms) available in the Freescale Touch library. The electrode structure types must always match the module and algorithm types.

This is an example of the AFID key detector configuration.

```

/* Key Detectors */
const struct ft_keydetector_afid keydec_afid =
{
    .signal_filter = 1,
    .fast_signal_filter = {
        .cutoff = 6
    },
    .slow_signal_filter = {
        .cutoff = 2
    },
    .base_avrg = {.n2_order = 12},
    .reset_rate = 10,
    .asc = {
        .touch_treshold_fall_rate = 1000,
        .noise_resets_minimum = 256,
        .resets_for_touch = 5,
    },
};

```

The electrode structure type must match the hardware module used for the data-measurement algorithm in the application. In this case, it is the `ft_electrode` type. You must define the electrode parameters and `ft_keydetector` interface.

```

/* Electrodes */
const struct ft_electrode electrode_0 =
{
    .pin_input = BOARD_TSI_ELECTRODE_1,
    .keydetector_interface = &ft_keydetector_afid_interface,
    .keydetector_params.afid = &keydec_afid,
};

const struct ft_electrode electrode_1 =
{
    .pin_input = BOARD_TSI_ELECTRODE_2,
    .keydetector_interface = &ft_keydetector_afid_interface,
    .keydetector_params.afid = &keydec_afid,
};

const struct ft_electrode electrode_2 =
{
    .pin_input = BOARD_TSI_ELECTRODE_3,
    .keydetector_interface = &ft_keydetector_afid_interface,
    .keydetector_params.afid = &keydec_afid,
};

const struct ft_electrode electrode_3 =
{
    .pin_input = BOARD_TSI_ELECTRODE_4,
    .keydetector_interface = &ft_keydetector_afid_interface,
    .keydetector_params.afid = &keydec_afid,
};

```

The Kinetis L family of MCUs contains a different TSI module version than the one used in the Kinetis K family of MCUs. Distinguishing the Kinetis L TSI module, we internally refer to it as TSIL. The module must be configured for a proper operation. However, the FT library helps during the application development, and it is not necessary to deal with the TSI module differences. The TSI hardware setup is displayed below.

Creating the Freescale Touch Application

```

/* Modules */
const struct ft_electrode * const module_0_electrodes[] = {&electrode_0, &electrode_1,
    &electrode_2, &electrode_3, NULL};

const tsi_config_t hw_config =
{
    .ps = kTsiElecOscPrescaler_16div,
    .extchrg = kTsiExtOscChargeCurrent_8uA,
    .refchrg = kTsiRefOscChargeCurrent_16uA,
    .nscn = kTsiConsecutiveScansNumber_32time,
    .lpclks = kTsiLowPowerInterval_100ms,
    .amclks = kTsiActiveClkSource_BusClock,
    .ampsc = kTsiActiveModePrescaler_8div,
    .lpscnitv = kTsiLowPowerInterval_100ms,
    .thresh = 100,
    .thresl = 200,
};

const struct ft_module tsi_module =
{
    .interface = &ft_module_tsi_module_interface,
    .electrodes = &module_0_electrodes[0],
    .config = (void*)&hw_config,
    .instance = 0,
    .module_params = NULL,
};

```

Once the modules and electrodes are set up, you can define the **Controls**. In this case, the control_0 is the **Analog Slider** control.

```

/* Controls */
const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
    &electrode_2, &electrode_3, NULL};

const struct ft_control_keypad keypad_params =
{
    .groups = NULL,
    .groups_size = 0,
};

const struct ft_control keypad_0 =
{
    .interface = &ft_control_keypad_interface,
    .electrodes = control_0_electrodes,
    .control_params.keypad = &keypad_params,
};

```

Now we are ready to connect all the pieces together in the **system** structure.

```

/* System */
const struct ft_control * const controls[] = {&keypad_0, NULL};
const struct ft_module * const modules[] = {&tsi_module, NULL};

const struct ft_system system_0 = {
    .controls = &controls[0],
    .modules = &modules[0],
    .time_period = 5,
    .init_time = 50,
};

```


5.1.5 The main() Function

The minimal application code must look like this:

```
#include <stdio.h>
#include <stdlib.h>
#include "fsl_device_registers.h"
#include "fsl_debug_console.h"
#include "fsl_clock_manager.h"
#include "fsl_interrupt_manager.h"
#include "fsl_pit_driver.h"
#include "fsl_os_abstraction.h"
#include "ft_setup.h"
#include "main.h"
#include "board.h"

static void port_led_init(void);
static void keypad_callback(const struct ft_control *control,
                           enum ft_control_keypad_event event,
                           uint32_t index);

uint8_t ft_memory_pool[2048];

const pit_user_config_t my_pit_config = {
    .isInterruptEnabled = true,
    .periodUs = 5000
};

int main(void)
{
    int32_t result;

    hardware_init();
    port_led_init();

    // Configure TSI pins
    configure_tsi_pins(0u);

    // Initialize the OS abstraction layer
    OSA_Init();

    if ((result = ft_init(&system_0, ft_memory_pool, sizeof(ft_memory_pool))) !=
        FT_SUCCESS)
    {
        switch(result)
        {
            case FT_FAILURE:
                printf("\nCannot initialize the Freescale Touch due to a non-specific error.\n");
                break;
            case FT_OUT_OF_MEMORY:
                printf("\nCannot initialize the Freescale Touch due to not enough memory.\n");
                break;
        }
        while(1); /* add code to handle this error */
    }

    printf("\nThe Freescale Touch has been successfully initialized.\n");

    printf("Unused memory: %d bytes, you can make the memory pool smaller without affecting the
        functionality.\n", (int)ft_mem_get_free_size());

    ft_electrode_enable(&electrode_0);
    ft_electrode_enable(&electrode_1);
    ft_electrode_enable(&electrode_2);
    ft_electrode_enable(&electrode_3);
    ft_control_enable(&keypad_0);
}
```

Creating the Freescale Touch Application

```

ft_control_keypad_set_autorepeat_rate(&keypad_0, 100, 1000);
ft_control_keypad_register_callback(&keypad_0, &keypad_callback);

// Run the PIT driver to generate 5 ms events
PIT_DRV_Init(0, false);

// Init PIT channel
PIT_DRV_InitChannel(0, 0, &my_pit_config);

// Start the PIT timer
PIT_DRV_StartTimer(0, 0);

while(1)
{
    ft_task();
}

void PIT_DRV_CallBack(uint32_t channel)
{
    (void) channel;
    ft_trigger();
}

static void port_led_init(void)
{
    /* LED Init */
    LED1_EN;
    LED2_EN;
    LED3_EN;
    LED4_EN;

    LED1_OFF;
    LED2_OFF;
    LED3_OFF;
    LED4_OFF;
}

static void keypad_callback(const struct ft_control *control,
                           enum ft_control_keypad_event event,
                           uint32_t index)
{
    switch(event)
    {
    case FT_KEYPAD_RELEASE:
        printf("Release %d.\n", (int)index);
        switch (index) {
            case 0:
                LED1_OFF;
                break;
            case 1:
                LED2_OFF;
                break;
            case 2:
                LED3_OFF;
                break;
            case 3:
                LED4_OFF;
                break;
            default:
                break;
        }
        break;
    case FT_KEYPAD_TOUCH:
        printf("Touch %d.\n", (int)index);
        switch (index) {
            case 0:
                LED1_ON;
        }
    }
}

```

```

        break;
    case 1:
        LED2_ON;
        break;
    case 2:
        LED3_ON;
        break;
    case 3:
        LED4_ON;
        break;
    default:
        break;
}
break;

case FT_KEYPAD_AUTOREPEAT:
    printf("AutoRepeat %d.\n", (int)index);
    switch (index) {
        case 0:
            LED1_TOGGLE;
            break;
        case 1:
            LED2_TOGGLE;
            break;
        case 2:
            LED3_TOGGLE;
            break;
        case 3:
            LED4_TOGGLE;
            break;
        default:
            break;
    }
    break;
}
}
}

```


Chapter 6

Release Notes

Freescle Touch v1.0

Overview

These are the release notes for the Freescle Touch software (FT) version 1.0. This is full function library to handle the touch events measured by the FSL TSI module or GPIO methods. It also has the capability to represent the measured results in various controls, such as: Keypad, Rotary, Slider, Analog Slider or Analog Rotary.

1. What is new

This is the first public release, and it brings the following features:

- Supported devices
 - MK60D10
 - MKL25Z4
 - MKL26Z4
 - MKL46Z4
- Supported modules (peripherals / methods)
 - TSI (v2 / v4)
 - GPIO (also in interrupt mode)
- Supported controls
 - Keypad
 - Slider
 - Rotary
 - Analog Slider
 - Analog Rotary
- Supported Development tools
 - IAR 7.4
 - KEIL 5.1
 - KDS 2.0

2. Development tools

The Freescle Touch 1.0 was compiled and tested using the following development tools: -IAR Embedded Workbench for ARM® version 7.40 -MDK-ARM Microcontroller Development Kit (Keil)® 5.11 -KDS Freescle Kinetis Design Studio v2.00

3. Supported development systems

This release supports the following boards and devices:

- MK60D10
 - TWR-K60D100M
- MKL25Z4
 - FRDM-KL25Z
- MKL26Z4
 - FRDM-KL26Z
- MKL46Z4
 - FRDM-KL46Z

4. Release contents

This table describes the release contents.

PBSDeliverable	Location
Documentation	{install_dir}/doc/
SW Examples	{install_dir}/sw/examples/
FreeMASTER application	{install_dir}/sw/freemaster/FreescaleTouch.pmp
FT Library source code	{install_dir}/sw/ft/

5. Freescale Touch release overview

The FT is intended for use with Freescale’s Kinetis MCU product family, based on the ARM Cortex®-M series architectures. The whole library is designed to work with KSDK 1.2.0. The release consists of:

- Documentation - Reference manual
- Example application for supported boards
- FreeMASTER application for FT
- FT library source code

5.1 Documentation

The release contains the generated documentation that describes the whole library with block schemes, and the whole API is described either as public, or as all private internal API. The documentation is provided in HTML format.

5.2 Example application for supported boards

There is couple of example application that’s runs on standard Freescale development boards, that’s show all aspects of the FT capabilities. All applications are prepared for all supported Development tools.

5.3 FreeMASTER application for FT

The FreeMASTER application is provided for better understanding and tuning of the final values of the key detector settings, and it automatically generates HTML pages based on the current configuration of the embedded application using TSA tables.

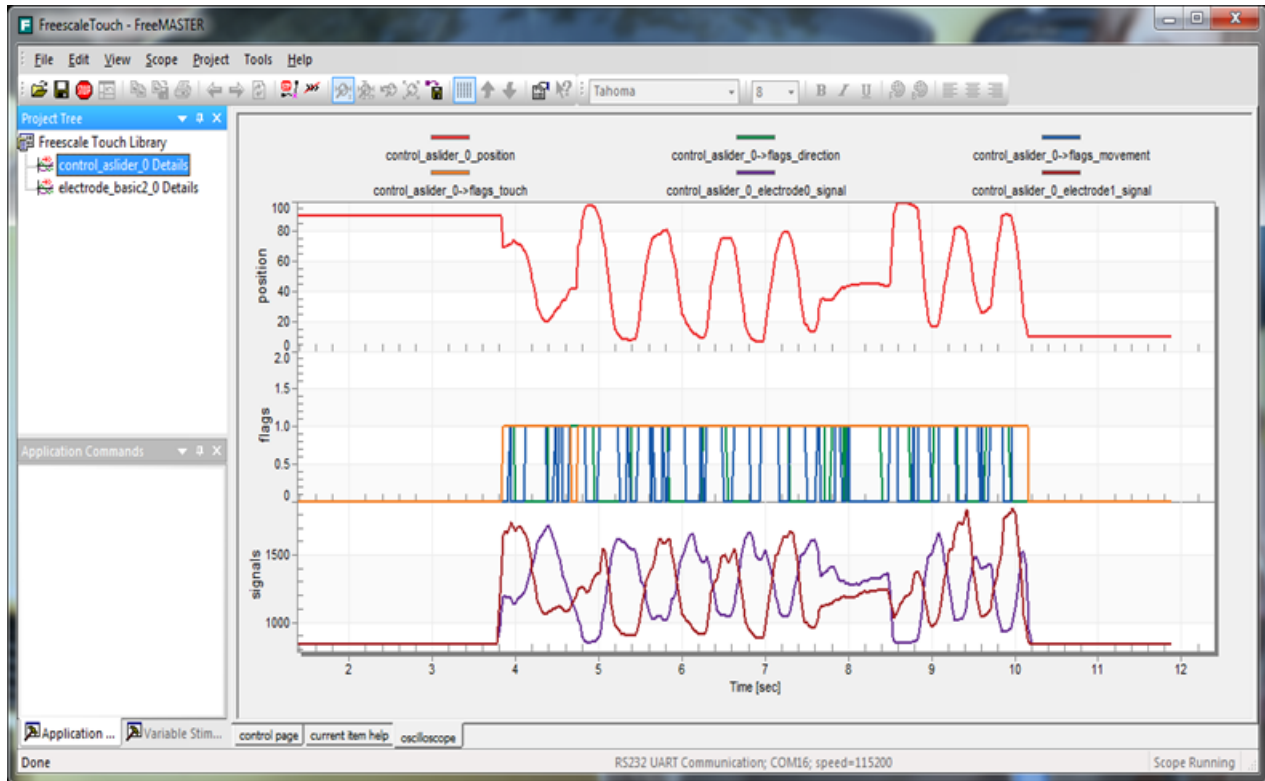


Figure 4: FreeMASTER example

5.4 Freescale Touch library software

The FT is distributed as a source code to allow you to check the code, and extend it using your own modules / key detectors / controls to meet the specific needs. The code is divided to two main folders:

- Includes - all public include header files - this is only folder that must be added to the preprocessor path for the FT library
- Source - Source code and private header files for the FT library

6 Known issues

6.1 Project paths

All the project paths are setup during the installation as an absolute path, relative to the KSDK installation. This allows you to have the project installed on the PC in any folder where the FT was installed, but it can cause issues on any other PC.

6.2 Low power support

The low power support in the FT is enabled by switching the mode of the FT module into the low-power mode. This functionality does not have any example in the current release package.

6.3 Shielding (water drops tolerance)

The current package has implemented the water tolerance in the same way as the TSS software. The implementation is done by shielding the electrode that will compensate for the signal drift caused by water drops. This functionality is not tested yet.

6.4 Proximity sensing

This version does not contain any high-level proximity-sensing API. To get started, you can just switch the FT module to the proximity mode.

7. Revision history

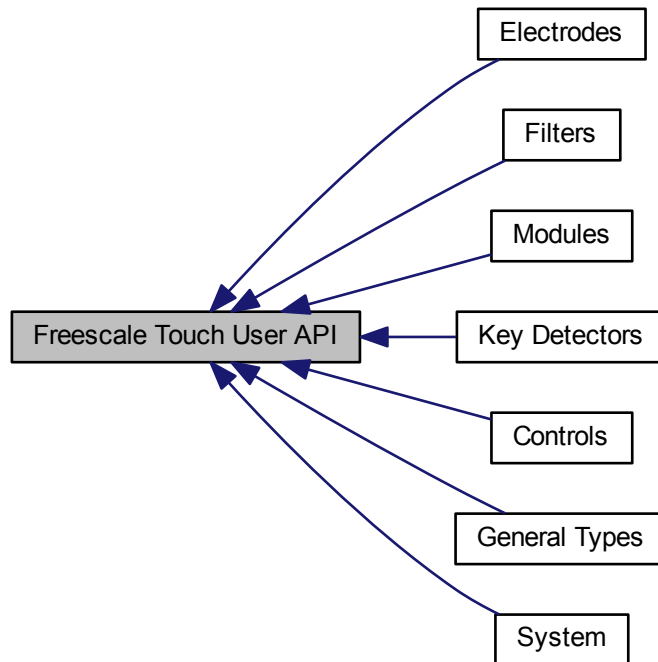
PBSRevision number	Date	Substantial changes
1.0	04/2015	Initial external release
1.0 Beta	12/2014	Internal release (VIP customers)
1.0 Alpha	10/2014	Internal Initial release

Chapter 7

Freescale Touch User API

7.1 Overview

The functions documented in this module are the primary functions used in the user application that uses the Freescale Touch library. The user calls the API functions to give run-time for the Freescale Touch measurement and data-processing algorithms. All library callbacks are executed in a context of one of these API calls. Collaboration diagram for Freescale Touch User API:



Modules

- [Controls](#)
- [Electrodes](#)
- [Filters](#)
- [Key Detectors](#)
- [Modules](#)
- [System](#)
- [General Types](#)

Overview

7.1.1 Analog Rotary Control

7.1.1.1 Overview

Analog Rotary enables the detection of jog-dial-like finger movement using three or more electrodes; it is represented by the [ft_control](#) structure.

The Analog Rotary Control uses three or more specially-shaped electrodes to enable the calculation of finger position within a circular area. The position algorithm uses the ratio of sibling electrode signals to estimate the finger position with the required precision.

The Analog Rotary works similarly to the "standard" Rotary, but requires less electrodes, while achieving a higher resolution of the calculated position. For example, a four-electrode Analog Rotary can provide the finger position detection in the range of 0-64. The shape of the electrodes needs to be designed specifically to achieve a stable signal with a linear dependence on finger movement.

The Analog Rotary Control provides Position, Direction, and Displacement values. It is able to generate event callbacks when finger Movement, Initial-touch, or Release is detected.

The image below shows a typical four-electrode Analog Rotary electrode placement.

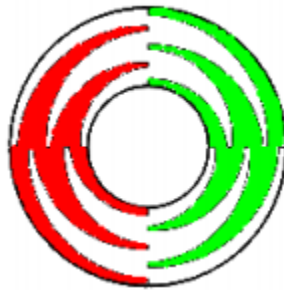


Figure 7.1.1: Analog Rotary Electrodes

Collaboration diagram for Analog Rotary Control:



Modules

- [Analog Rotary Control API](#)

Data Structures

- struct [ft_control_arotary](#)

Typedefs

- typedef void(* [ft_control_arotary_callback](#))(const struct [ft_control](#) *control, enum [ft_control_arotary_event](#) event, uint32_t position)

Enumerations

- enum [ft_control_arotary_event](#) {
[FT_AROTARY_MOVEMENT](#),
[FT_AROTARY_ALL_RELEASE](#),
[FT_AROTARY_INITIAL_TOUCH](#) }

Variables

- struct [ft_control_interface](#) [ft_control_arotary_interface](#)

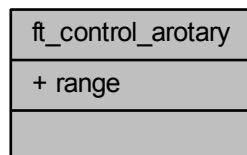
7.1.1.2 Data Structure Documentation

7.1.1.2.1 struct [ft_control_arotary](#)

The main structure representing the Analog Rotary Control.

An instance of this data type represents the Analog Rotary Control. You are responsible to initialize all the members before registering the control in the system. This structure can be allocated in ROM.

Collaboration diagram for [ft_control_arotary](#):



Overview

Data Fields

uint8_t	range	Range.
---------	-------	--------

7.1.1.3 Typedef Documentation

7.1.1.3.1 typedef void(* ft_control_arotary_callback)(const struct ft_control *control, enum ft_control_arotary_event event, uint32_t position)

Analog Rotary event callback function pointer type.

7.1.1.4 Enumeration Type Documentation

7.1.1.4.1 enum ft_control_arotary_event

Analog Rotary event types.

Enumerator

FT_AROTARY_MOVEMENT Finger movement event.

FT_AROTARY_ALL_RELEASE Release event.

FT_AROTARY_INITIAL_TOUCH Initial-touch event.

7.1.1.5 Variable Documentation

7.1.1.5.1 struct ft_control_interface ft_control_arotary_interface

An interface structure, which contains pointers to the entry points of the Analog Rotary algorithms. A pointer to this structure must be assigned to any instance of the [ft_control](#) to define the control behavior. Can't be NULL.

7.1.1.6 Analog Rotary Control API

7.1.1.6.1 Overview

These functions can be used to set or get the Analog Rotary control properties.

A common example definition of the Analog Rotary control for all source code examples is as follows:

```
* // definition of the electrode array used by the control (more info in electrodes )
* const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*   &electrode_2, &electrode_3, NULL};
*
* // Define additional parameters of the Analog Rotary
* const struct ft_control_arotary my_arotary_params =
* {
*   .range = 255,
* };
*
* // Definition of the Analog Rotary control
* const struct ft_control my_arotary_control =
* {
*   .interface = &ft_control_arotary_control_interface,
*   .electrodes = control_0_electrodes,
*   .control_params.arotary = &my_arotary_params,
* };
*
*
```

Collaboration diagram for Analog Rotary Control API:



Functions

- void `ft_control_arotary_register_callback` (const struct `ft_control` *control, `ft_control_arotary_callback` callback)
Registers the Analog Rotary events handler function.
- uint32_t `ft_control_arotary_get_position` (const struct `ft_control` *control)
Get the Analog Rotary 'Position' value.
- uint32_t `ft_control_arotary_is_touched` (const struct `ft_control` *control)
Get 'Touched' state.
- uint32_t `ft_control_arotary_movement_detected` (const struct `ft_control` *control)
Get 'Movement' flag.
- uint32_t `ft_control_arotary_get_direction` (const struct `ft_control` *control)
Get 'Direction' flag.
- uint32_t `ft_control_arotary_get_invalid_position` (const struct `ft_control` *control)
Returns invalid position flag.

Overview

7.1.1.6.2 Function Documentation

7.1.1.6.2.1 `uint32_t ft_control_arotary_get_direction (const struct ft_control * control)`

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

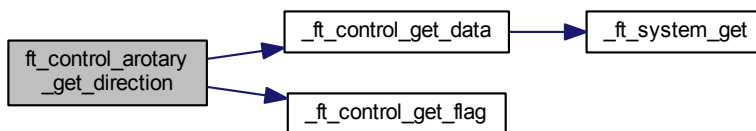
- Non-zero value, if a movement towards higher values is detected.
- Returns zero, if a movement towards zero is detected. Example:

```

* uint32_t direction;
* // Get direction of arotary control
* direction = ft_control_arotary_get_direction(&my_arotary_control);
* if(direction)
*     printf("The Analog Rotary direction is left.");
* else
*     printf("The Analog Rotary direction is right.");
*

```

Here is the call graph for this function:



7.1.1.6.2.2 uint32_t ft_control_arotary_get_invalid_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value, if an invalid touch is detected.

This flag is set, if the algorithm detects two or more fingers touching the electrodes, which are not adjacent to each other. Example:

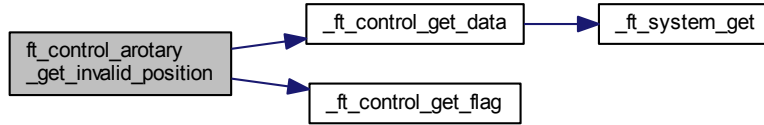
```

* uint32_t invalid_position;
* // Get invalid position of arotary control
* invalid_position = ft_control_arotary_get_invalid_position(&
    my_arotary_control);
* if(invalid_position)
*     printf("The Analog Rotary control has an invalid position (two fingers touch ?).");
* else
*     printf("The Analog Rotary control has a valid position.");
*

```

Overview

Here is the call graph for this function:



7.1.1.6.2.3 uint32_t ft_control_arotary_get_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Position. The returned value is in the range of zero to the maximum value configured in the [ft_control](#) structure.

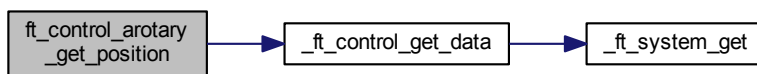
This function retrieves the actual finger position value. Example:

```

* uint32_t position;
* // Get position of arotary control
* position = ft_control_arotary_get_position(&my_arotary_control);
* printf("Position of Analog Rotary control is: %d.", position);
*

```

Here is the call graph for this function:



7.1.1.6.2.4 uint32_t ft_control_arotary_is_touched (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

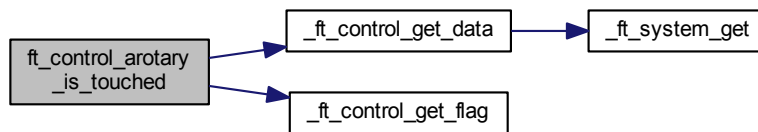
Non-zero value, if the control is currently touched. Example:

```

* uint32_t touched;
* // Get state of arotary control
* touched = ft_control_arotary_is_touched(&my_arotary_control);
* if(touched)
*     printf("The Analog Rotary control is currently touched.");
* else
*     printf("The Analog Rotary control is currently not touched.");
*

```

Here is the call graph for this function:



7.1.1.6.2.5 uint32_t ft_control_arotary_movement_detected (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value, if the control currently detects finger movement. Example:

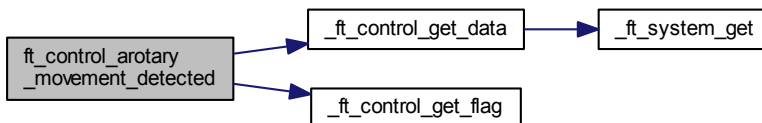
```

* uint32_t movement;
* // Get state of arotary control
* movement = ft_control_arotary_movement_detected(&my_arotary_control);
* if(movement)
*     printf("The Analog Rotary control is currently moving.");
* else
*     printf("The Analog Rotary control is currently not moving.");
*

```

Overview

Here is the call graph for this function:



7.1.1.6.2.6 void ft_control_arotary_register_callback (const struct ft_control * control, ft_control_arotary_callback callback)

Parameters

<i>control</i>	Pointer to the control.
<i>callback</i>	Address of function to be invoked.

Returns

none

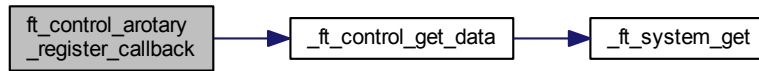
Register the specified callback function as the Analog Rotary events handler. If the callback parameter is NULL, the callback is disabled. Example:

```

*
* //Create the callback function for arotary
* static void my_arotary_cb(const struct ft_control *control,
*                          enum ft_control_arotary_event event,
*                          uint32_t position)
* {
*   (void)control;
*   char* event_names[] =
*   {
*     "FT_AROTARY_MOVEMENT",
*     "FT_AROTARY_ALL_RELEASE",
*     "FT_AROTARY_INITIAL_TOUCH",
*   };
*   printf("New analog rotary control event %s on position: %d.", event_names[event], position);
* }
*
* // register the callback function for arotary
* ft_control_arotary_register_callback(&my_arotary_control,
*   my_arotary_cb);
*

```

Here is the call graph for this function:



Overview

7.1.2 Analog Slider Control

7.1.2.1 Overview

Analog Slider enables detection of linear finger movement using two or more electrodes; it is represented by the [ft_control_aslider](#) structure.

The Analog Slider Control uses two or more specially-shaped electrodes to enable the calculation of finger position within a linear area. The position algorithm uses ratio of electrode signals to estimate finger position with required precision.

The Analog Slider works similarly to the "standard" Slider, but requires less electrodes, while achieving a higher resolution of the calculated position. For example, a two-electrode analog slider can provide finger position detection in the range of 0-127. The shape of the electrodes needs to be designed specifically to achieve a stable signal with a linear dependence on finger movement.

The Analog Slider Control provides Position, Direction, and Displacement values. It is able to generate event callbacks when finger Movement, Initial-touch, or Release is detected.

The figure below shows a typical two-electrode Analog Slider electrode placement.



Figure 7.1.2: Analog Slider Electrodes

Collaboration diagram for Analog Slider Control:



Modules

- [Analog Slider Control API](#)

Data Structures

- struct [ft_control_aslider](#)

Typedefs

- typedef void(* [ft_control_aslider_callback](#))(const struct [ft_control](#) *control, enum [ft_control_aslider_event](#) event, uint32_t position)

Enumerations

- enum [ft_control_aslider_event](#) {
[FT_ASLIDER_MOVEMENT](#),
[FT_ASLIDER_ALL_RELEASE](#),
[FT_ASLIDER_INITIAL_TOUCH](#) }

Variables

- struct [ft_control_interface](#) [ft_control_aslider_interface](#)

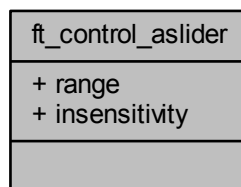
7.1.2.2 Data Structure Documentation

7.1.2.2.1 struct [ft_control_aslider](#)

The main structure representing the Analog Slider Control.

An instance of this data type represents the Analog Slider Control. You are responsible to initialize all the members before registering the control in the system. This structure can be allocated in ROM.

Collaboration diagram for [ft_control_aslider](#):



Data Fields

Overview

uint8_t	insensitivity	Insensitivity for the callbacks invokes when the position is changed.
uint8_t	range	Maximum range for the ram->position value.

7.1.2.3 Typedef Documentation

7.1.2.3.1 typedef void(* ft_control_aslider_callback)(const struct ft_control *control, enum ft_control_aslider_event event, uint32_t position)

Analog Slider event callback function pointer type.

7.1.2.4 Enumeration Type Documentation

7.1.2.4.1 enum ft_control_aslider_event

Analog Slider event types.

Enumerator

FT_ASLIDER_MOVEMENT Finger movement event

FT_ASLIDER_ALL_RELEASE Release event

FT_ASLIDER_INITIAL_TOUCH Initial-touch event

7.1.2.5 Variable Documentation

7.1.2.5.1 struct ft_control_interface ft_control_aslider_interface

An interface structure, which contains pointers to the entry points of the Analog Slider algorithms. A pointer to this structure must be assigned to any instance of the [ft_control_aslider](#) to define the control behavior.

7.1.2.6 Analog Slider Control API

7.1.2.6.1 Overview

These functions can be used to set or get the Analog Slider control properties.

Common example definition of the Analog Slider control for all source code examples is as follows:

```
* // definition of the electrode array used by the control (more info in electrodes )
* const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*   NULL};
*
* // Define additional parameters of Analog Slider
* const struct ft_control_aslider my_aslider_params =
* {
*   .range = 100,
* };
*
* // Definition of the Analog Slider control
* const struct ft_control my_aslider_control =
* {
*   .interface = &ft_control_aslider_control_interface,
*   .electrodes = control_0_electrodes,
*   .control_params.aslider = &my_aslider_params,
* };
*
*
```

Collaboration diagram for Analog Slider Control API:



Functions

- void [ft_control_aslider_register_callback](#) (const struct [ft_control](#) *control, [ft_control_aslider_callback](#) callback)
Registers the Analog Slider events handler function.
- uint32_t [ft_control_aslider_get_position](#) (const struct [ft_control](#) *control)
Get the Analog Slider 'Position' value.
- uint32_t [ft_control_aslider_is_touched](#) (const struct [ft_control](#) *control)
Get 'Touched' state.
- uint32_t [ft_control_aslider_movement_detected](#) (const struct [ft_control](#) *control)
Get 'Movement' flag.
- uint32_t [ft_control_aslider_get_direction](#) (const struct [ft_control](#) *control)
Get 'Direction' flag.
- uint32_t [ft_control_aslider_get_invalid_position](#) (const struct [ft_control](#) *control)
Returns invalid position flag.

Overview

7.1.2.6.2 Function Documentation

7.1.2.6.2.1 `uint32_t ft_control_aslider_get_direction (const struct ft_control * control)`

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

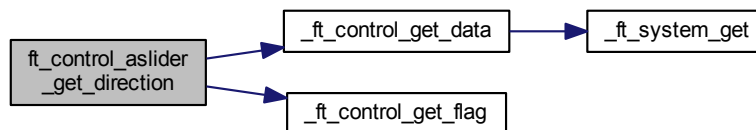
Non-zero value, when a movement towards higher values is detected. Returns zero when a movement towards zero is detected. Example:

```

* uint32_t direction;
* // Get direction of aslider control
* direction = ft_control_aslider_get_direction(&my_aslider_control);
* if(direction)
*     printf("The Analog Slider direction is left.");
* else
*     printf("The Analog Slider direction is right.");
*

```

Here is the call graph for this function:



7.1.2.6.2.2 uint32_t ft_control_aslider_get_invalid_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value, when an invalid touch is detected.

This function works only in the Analog Slider controls, consisting of at least three electrodes. This flag is set when the algorithm detects two or more fingers touching the electrodes that are not adjacent to each other. Example:

```

* uint32_t invalid_position;
* // Get invalid position of aslider control
* invalid_position = ft_control_aslider_get_invalid_position(&
*     my_aslider_control);

```

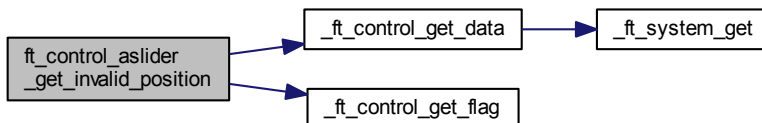
Overview

```

* if(invalid_position)
*     printf("The Analog Slider control has an invalid position (two fingers touch ?).");
* else
*     printf("The Analog Slider control has a valid position.");
*

```

Here is the call graph for this function:



7.1.2.6.2.3 uint32_t ft_control_aslider_get_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Position. The returned value is in the range of zero to maximum value configured in the [ft_control_aslider](#) structure.

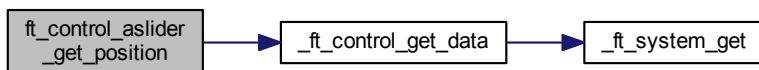
This function retrieves the actual finger position value. Example:

```

* uint32_t position;
* // Get position of aslider control
* position = ft_control_aslider_get_position(&my_aslider_control);
* printf("Position of analog slider control is: %d.", position);
*

```

Here is the call graph for this function:



7.1.2.6.2.4 uint32_t ft_control_aslider_is_touched (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

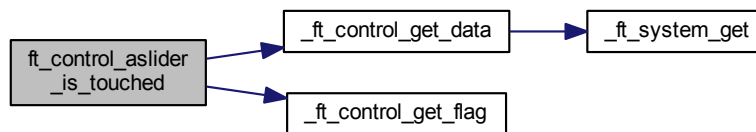
Non-zero value, when the control is currently touched. Example:

```

* uint32_t touched;
* // Get state of aslider control
* touched = ft_control_aslider_is_touched(&my_aslider_control);
* if(touched)
*     printf("The Analog Slider control is currently touched.");
* else
*     printf("The Analog Slider control is currently not touched.");
*

```

Here is the call graph for this function:



7.1.2.6.2.5 uint32_t ft_control_aslider_movement_detected (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value, if the control currently detects finger movement. Example:

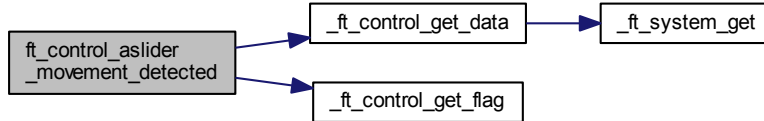
```

* uint32_t movement;
* // Get state of aslider control
* movement = ft_control_aslider_movement_detected(&my_aslider_control);
* if(movement)
*     printf("The Analog Slider control is currently moving.");
* else
*     printf("The Analog Slider control is currently not moving.");
*

```

Overview

Here is the call graph for this function:



7.1.2.6.2.6 void ft_control_aslider_register_callback (const struct ft_control * control, ft_control_aslider_callback callback)

Parameters

<i>control</i>	Pointer to the control.
<i>callback</i>	Address of the function to be invoked.

Returns

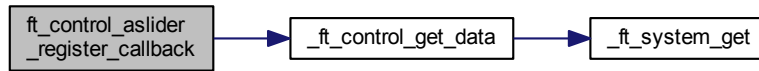
none

Register the specified callback function as the Analog Slider events handler. If the callback parameter is NULL, the callback is disabled. Example:

```

*
* //Create the callback function for aslider
* static void my_aslider_cb(const struct ft_control *control,
*                          enum ft_control_aslider_event event,
*                          uint32_t position)
* {
*   (void)control;
*   char* event_names[] =
*   {
*     "FT_ASLIDER_MOVEMENT",
*     "FT_ASLIDER_ALL_RELEASE",
*     "FT_ASLIDER_INITIAL_TOUCH",
*   };
*   printf("New analog slider control event %s on position: %d.", event_names[event], position);
* }
*
* // register the callback function for aslider movement
ft_control_aslider_register_callback(&my_aslider_control,
my_aslider_cb);
  
```

Here is the call graph for this function:



Overview

7.1.3 Keypad Control

7.1.3.1 Overview

Keypad implements the keyboard-like functionality on top of an array of electrodes; it is represented by the `ft_control_keypad` structure.

The application may use the Electrode API to determine the touch or release states of individual electrodes. The Keypad simplifies this task, and extends this simple scenario by introducing a concept of a "key". The "key" is represented by one or more physical electrodes, therefore the Keypad control enables sharing of one electrode by several keys. Each key is defined by a set of electrodes that all must be touched, in order to report the "key press" event.

The Keypad Control provides the Key status values and is able to generate Key Touch, Auto-repeat, and Release events.

The figures below show simple and grouped Keypad electrode layouts.

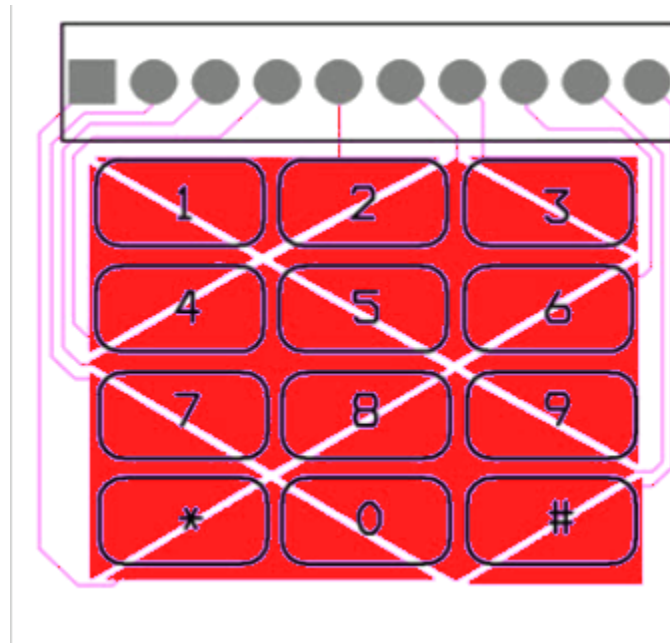


Figure 7.1.3: Keypad Electrodes

Collaboration diagram for Keypad Control:



Modules

- [Keypad Control API](#)

Data Structures

- [struct ft_control_keypad](#)

Typedefs

- typedef void(* [ft_control_keypad_callback](#))(const struct [ft_control](#) *control, enum [ft_control_keypad_event](#) event, uint32_t index)

Enumerations

- enum [ft_control_keypad_event](#) {
[FT_KEYPAD_RELEASE](#),
[FT_KEYPAD_TOUCH](#),
[FT_KEYPAD_AUTOREPEAT](#) }

Variables

- [struct ft_control_interface ft_control_keypad_interface](#)

7.1.3.2 Data Structure Documentation

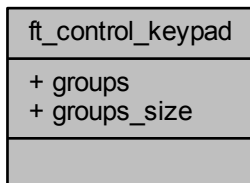
7.1.3.2.1 struct ft_control_keypad

The main structure representing the Keypad Control.

An instance of this data type represents the Keypad Control. You must initialize all the members before registering the control in the system. This structure can be allocated in ROM.

Overview

Collaboration diagram for ft_control_keypad:



Data Fields

uint32_t const *	groups	Pointer to the group definitions. An array of integers, where bits in the integer represents electrodes in a group.
uint8_t	groups_size	Number of groups.

7.1.3.3 Typedef Documentation

7.1.3.3.1 `typedef void(* ft_control_keypad_callback)(const struct ft_control *control, enum ft_control_keypad_event event, uint32_t index)`

Keypad event callback function pointer type.

7.1.3.4 Enumeration Type Documentation

7.1.3.4.1 `enum ft_control_keypad_event`

Keypad event types.

Enumerator

FT_KEYPAD_RELEASE Release event
FT_KEYPAD_TOUCH Key-touch event
FT_KEYPAD_AUTOREPEAT Auto-repeat event

7.1.3.5 Variable Documentation

7.1.3.5.1 struct ft_control_interface ft_control_keypad_interface

An interface structure, which contains pointers to the entry points of the Keypad algorithms. A pointer to this structure must be assigned to any instance of the [ft_control_keypad](#), to define the control behavior.

Overview

7.1.3.6 Keypad Control API

7.1.3.6.1 Overview

These functions can be used to set or get the Keypad control properties.

A common example definition of the Keypad control for all source code examples is as follows:

```
* // definition of electrode array used by control (more info in electrodes )
* const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*   &electrode_2, &electrode_3, NULL};
*
* const struct ft_control_keypad keypad_params =
* {
*   .groups = NULL,
*   .groups_size = 0,
* };
*
* // Definition of rotary control
* const struct ft_control my_keypad_control =
* {
*   .interface = &ft_control_keypad_control_interface,
*   .electrodes = control_0_electrodes,
*   .control_params.keypad = keypad_params,
* };
*
*
```

Collaboration diagram for Keypad Control API:



Functions

- void [ft_control_keypad_only_one_key_valid](#) (const struct [ft_control](#) *control, uint32_t enable)
Enable or disable the functionality that only one key press is valid.
- void [ft_control_keypad_register_callback](#) (const struct [ft_control](#) *control, [ft_control_keypad_callback](#) callback)
Registers the Keypad event handler function.
- void [ft_control_keypad_set_autorepeat_rate](#) (const struct [ft_control](#) *control, uint32_t value, uint32_t start_value)
Set the auto-repeat rate.
- uint32_t [ft_control_keypad_get_autorepeat_rate](#) (const struct [ft_control](#) *control)
Get the auto-repeat rate.
- uint32_t [ft_control_keypad_is_button_touched](#) (const struct [ft_control](#) *control, uint32_t index)
Get the button touch status.

7.1.3.6.2 Function Documentation

7.1.3.6.2.1 `uint32_t ft_control_keypad_get_autorepeat_rate (const struct ft_control * control)`

Overview

Parameters

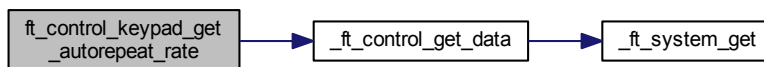
<i>control</i>	Pointer to the Keypad control.
----------------	--------------------------------

Returns

The auto-repeat value or 0 when this feature is disabled. Example:

```
* uint32_t autorepeat_rate;
* //Get autorepeat rate
* autorepeat_rate = ft_control_keypad_get_autorepeat_rate(&
*   my_keypad_control);
* printf("Auto-repeat rate of my keypad control is set to : %d.", autorepeat_rate);
*
```

Here is the call graph for this function:



7.1.3.6.2.2 uint32_t ft_control_keypad_is_button_touched (const struct ft_control * control, uint32_t index)

Parameters

<i>control</i>	Pointer to the Keypad control.
<i>index</i>	The button's number (index) in the control.

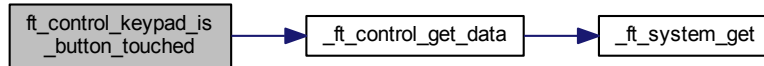
Returns

1 if the button is touched, 0 otherwise.

Returns the state of the keypad button. In case there are groups defined, the touch state reflects that all electrodes forming one button are touched. Otherwise, a button is in the release state. Example:

```
* uint32_t touched;
* // Get the state of first key Keypad control
* touched = ft_control_keypad_is_button_touched(&my_keypad_control, 0);
* if (touched)
*   printf("The first key of the Keypad control is currently touched.");
* else
*   printf("The first key of the Keypad control is currently not touched.");
*
```

Here is the call graph for this function:



7.1.3.6.2.3 void ft_control_keypad_only_one_key_valid (const struct ft_control * control, uint32_t enable)

Parameters

<i>control</i>	Pointer to the control.
<i>enable</i>	enable the only one key pressed is valid.

Returns

none

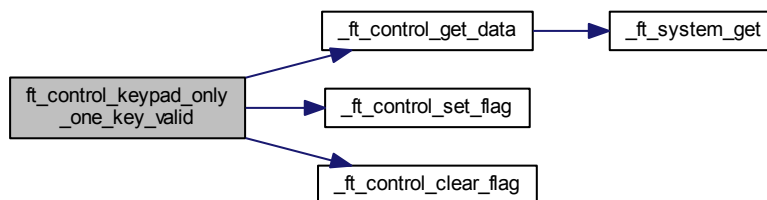
Enable or Disable the only one key press is valid at once. The behavior is following: Once the feature is enabled the first touched key is valid and all other are ignored since the active electrode is pressed. Example:

```

*
* // switch off the only one key is valid functionality
* ft_control_keypad_only_one_key_valid(&my_keypad_control, 0);
*

```

Here is the call graph for this function:



7.1.3.6.2.4 void ft_control_keypad_register_callback (const struct ft_control * control, ft_control_keypad_callback callback)

Overview

Parameters

<i>control</i>	Pointer to the control.
<i>callback</i>	Address of function to be invoked.

Returns

none

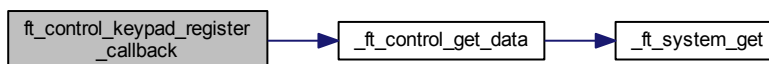
Register the specified callback function as the Keypad event handler. If the callback parameter is NULL, the callback is disabled. Example:

```

*
* //Create the callback function for keypad
* static void my_keypad_cb(const struct ft_control *control,
*                          enum ft_control_keypad_event event,
*                          uint32_t index)
* {
*     (void)control;
*     char* event_names[] =
*     {
*         "FT_KEYPAD_RELEASE",
*         "FT_KEYPAD_TOUCH",
*         "FT_KEYPAD_AUTOREPEAT",
*     };
*     printf("New keypad control event %s on key: %d.", event_names[event], index);
* }
*
* // register the callback function for keypad
* ft_control_keypad_register_touch_callback(&my_keypad_control, my_keypad_touch_cb);
*

```

Here is the call graph for this function:



7.1.3.6.2.5 void ft_control_keypad_set_autorepeat_rate (const struct ft_control * control, uint32_t value, uint32_t start_value)

Parameters

<i>control</i>	Pointer to the Keypad control.
<i>value</i>	Auto-repeat value. Value 0 disables the auto-repeat feature.
<i>value</i>	Auto-repeat start value. Value 0 disables the auto-repeat start feature.

Returns

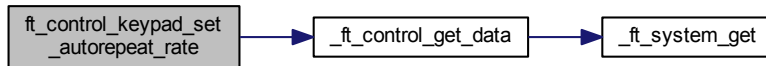
none Example:

```

*
* //Set autorepeat rate to 100 ticks and start after 1000 ticks
* ft_control_keypad_set_autorepeat_rate(&my_keypad_control, 100, 1000
* );
*

```

Here is the call graph for this function:



Overview

7.1.4 Matrix Control

7.1.4.1 Overview

Matrix enables the detection of... It is currently not yet implemented.
 The figure below shows a typical Matrix electrode placement.

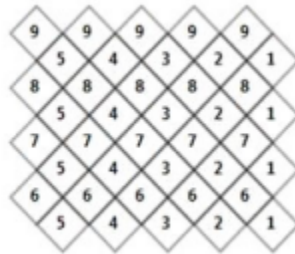
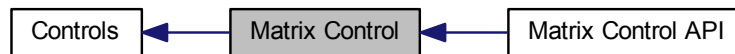


Figure 7.1.4: Rotary Electrodes

Collaboration diagram for Matrix Control:



Modules

- [Matrix Control API](#)

7.1.4.2 Matrix Control API

These functions can be used to set or get the Matrix control properties.

A common example definition of the Matrix control for all source code examples is as follows:

*

Collaboration diagram for Matrix Control API:



Overview

7.1.5 Rotary Control

7.1.5.1 Overview

The Rotary control enables the detection of jog-dial-like finger movement using discrete electrodes; it is represented by the `ft_control_rotary_control` structure.

The Rotary control uses a set of discrete electrodes to enable the calculation of finger position within a circular area. The position algorithm localizes the touched electrode and its sibling electrodes, to estimate the finger position. A Rotary control consisting of N electrodes enables the rotary position to be calculated in $2N$ steps.

The Rotary control provides Position, Direction, and Displacement values. It is able to generate event callbacks when the finger Movement, Initial-touch, or Release is detected.

The figure below shows a typical Rotary electrode placement.

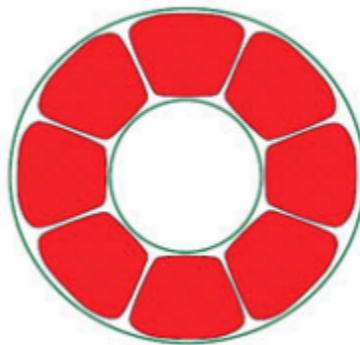
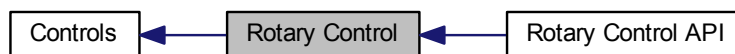


Figure 7.1.5: Rotary Electrodes

Collaboration diagram for Rotary Control:



Modules

- [Rotary Control API](#)

Typedefs

- typedef void(* [ft_control_rotary_callback](#))(const struct [ft_control](#) *control, enum [ft_control_rotary_event](#), uint32_t position)

Enumerations

- enum [ft_control_rotary_event](#) {
[FT_ROTARY_MOVEMENT](#),
[FT_ROTARY_ALL_RELEASE](#),
[FT_ROTARY_INITIAL_TOUCH](#) }

Variables

- struct [ft_control_interface](#) [ft_control_rotary_interface](#)

7.1.5.2 Typedef Documentation

7.1.5.2.1 typedef void(* [ft_control_rotary_callback](#))(const struct [ft_control](#) *control, enum [ft_control_rotary_event](#), uint32_t position)

Rotary event callback function pointer type.

7.1.5.3 Enumeration Type Documentation

7.1.5.3.1 enum [ft_control_rotary_event](#)

Rotary event types.

Enumerator

FT_ROTARY_MOVEMENT Finger movement event
FT_ROTARY_ALL_RELEASE Release event
FT_ROTARY_INITIAL_TOUCH Initial-touch event

7.1.5.4 Variable Documentation

7.1.5.4.1 struct [ft_control_interface](#) [ft_control_rotary_interface](#)

The interface structure, which contains pointers to the entry points of the Rotary algorithms. A pointer to this structure must be assigned to any instance of [ft_control_rotary_control](#) to define the control behavior.

Overview

7.1.5.5 Rotary Control API

7.1.5.5.1 Overview

These functions can be used to set or get the Rotary control properties.

The common example definition of Rotary control for all source code examples is as follows:

```
* // definition of electrode array used by control (more info in electrodes )
* const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*   &electrode_2, &electrode_3, NULL};
*
* // Definition of the Rotary control
* const struct ft_control my_rotary_control =
* {
*   .interface = &ft_control_rotary_interface,
*   .electrodes = control_0_electrodes,
* };
*
*
```

Collaboration diagram for Rotary Control API:



Functions

- void [ft_control_rotary_register_callback](#) (const struct [ft_control](#) *control, [ft_control_rotary_callback](#) callback)
Registers the events handler function.
- uint32_t [ft_control_rotary_get_position](#) (const struct [ft_control](#) *control)
Get the Rotary 'Position' value.
- uint32_t [ft_control_rotary_is_touched](#) (const struct [ft_control](#) *control)
Get 'Touched' state.
- uint32_t [ft_control_rotary_movement_detected](#) (const struct [ft_control](#) *control)
Get 'Movement' flag.
- uint32_t [ft_control_rotary_get_direction](#) (const struct [ft_control](#) *control)
Get 'Direction' flag.
- uint32_t [ft_control_rotary_get_invalid_position](#) (const struct [ft_control](#) *control)
Get 'Invalid' flag.

7.1.5.5.2 Function Documentation

7.1.5.5.2.1 uint32_t ft_control_rotary_get_direction (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

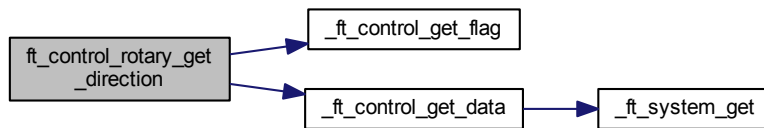
Non-zero value, when a movement is detected in a direction towards higher values. Returns zero, when a movement is detected towards zero. Example:

```

* uint32_t direction;
* // Get direction of rotary control
* direction = ft_control_rotary_get_direction(&my_rotary_control);
* if(direction)
*     printf("The Rotary direction is left.");
* else
*     printf("The Rotary direction is right.");
*

```

Here is the call graph for this function:



7.1.5.5.2.2 uint32_t ft_control_rotary_get_invalid_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value when an invalid position was detected, otherwise a zero value. Example:

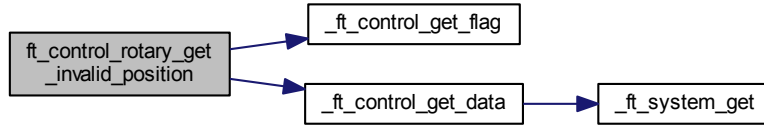
```

* uint32_t invalid_position;
* // Get invalid position of Rotary control
* invalid_position = ft_control_rotary_get_invalid_position(&
*     my_rotary_control);
* if(invalid_position)
*     printf("The Rotary control has an invalid position (two fingers touch ?).");
* else
*     printf("The Rotary control has a valid position.");
*

```

Overview

Here is the call graph for this function:



7.1.5.5.2.3 uint32_t ft_control_rotary_get_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Position. The returned value is in the range of zero to 2N-1, where N is the number of electrodes assigned to Rotary control.

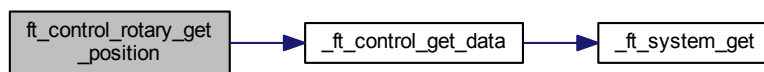
This function retrieves the actual finger position value. Example:

```

* uint32_t position;
* // Get position of Rotary control
* position = ft_control_rotary_get_position(&my_rotary_control);
* printf("Position of Rotary control is: %d.", position);
*

```

Here is the call graph for this function:



7.1.5.5.2.4 uint32_t ft_control_rotary_is_touched (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

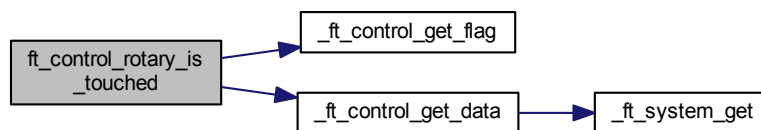
Non-zero value, when the control is currently touched. Example:

```

* uint32_t touched;
* // Get state of the Rotary control
* touched = ft_control_rotary_is_touched(&my_rotary_control);
* if(touched)
*     printf("The Rotary control is currently touched.");
* else
*     printf("The Rotary control is currently not touched.");
*

```

Here is the call graph for this function:



7.1.5.5.2.5 uint32_t ft_control_rotary_movement_detected (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value, when the control detects finger movement. Example:

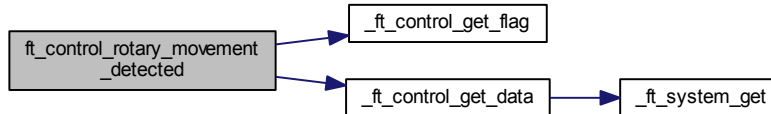
```

* uint32_t movement;
* // Get state of rotary control
* movement = ft_control_rotary_movement_detected(&my_rotary_control);
* if(movement)
*     printf("The Rotary control is currently moving.");
* else
*     printf("The Rotary control is currently not moving.");
*

```

Overview

Here is the call graph for this function:



7.1.5.5.2.6 void ft_control_rotary_register_callback (const struct ft_control * *control*, ft_control_rotary_callback *callback*)

Parameters

<i>control</i>	Pointer to the control.
<i>callback</i>	Adress of function to be invoked.

Returns

none

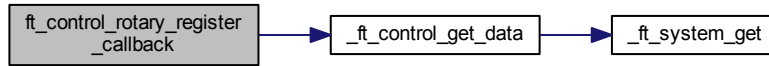
Register the specified callback function as the Rotary events handler. Example:

```

*
* //Create the callback function for arotary
* static void my_rotary_cb(const struct ft_control *control,
*                          enum ft_control_arotary_event event,
*                          uint32_t position)
* {
*   (void)control;
*   char* event_names[] =
*   {
*     "FT_ROTARY_MOVEMENT",
*     "FT_ROTARY_ALL_RELEASE",
*     "FT_ROTARY_INITIAL_TOUCH",
*   };
*   printf("New rotary control event %s on position: %d.", event_names[event], position);
* }
*
* // register the callback function for rotary
* ft_control_rotary_register_callback (&my_rotary_control, my_rotary_cb)
* ;
*

```


Here is the call graph for this function:



Overview

7.1.6 Slider control

7.1.6.1 Overview

Slider control enables the detection of linear finger movement using discrete electrodes; it is represented by the [ft_control](#) structure.

The Slider control uses a set of discrete electrodes to enable calculation of the finger position within a linear area. The position algorithm localizes the touched electrode and its sibling electrodes to estimate finger position. A Slider consisting of N electrodes enables the position to be calculated in $2N-1$ steps.

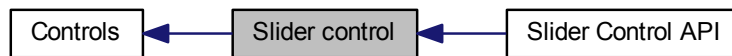
The Slider control provides Position, Direction, and Displacement values. It is able to generate event callbacks when finger Movement, Initial-touch, or Release is detected.

The image below shows a typical Slider electrode placement.



Figure 7.1.6: Slider Electrodes

Collaboration diagram for Slider control:



Modules

- [Slider Control API](#)

Typedefs

- typedef void(* [ft_control_slider_callback](#))(const struct [ft_control](#) *control, enum [ft_control_slider_event](#), uint32_t position)

Enumerations

- enum [ft_control_slider_event](#) {
[FT_SLIDER_MOVEMENT](#),
[FT_SLIDER_ALL_RELEASE](#),
[FT_SLIDER_INITIAL_TOUCH](#) }

Variables

- struct [ft_control_interface](#) [ft_control_slider_interface](#)

7.1.6.2 Typedef Documentation

7.1.6.2.1 typedef void(* [ft_control_slider_callback](#))(const struct [ft_control](#) *control, enum [ft_control_slider_event](#), uint32_t position)

Slider event callback function pointer type.

7.1.6.3 Enumeration Type Documentation

7.1.6.3.1 enum [ft_control_slider_event](#)

Slider event types.

Enumerator

FT_SLIDER_MOVEMENT Finger movement event.

FT_SLIDER_ALL_RELEASE Release event.

FT_SLIDER_INITIAL_TOUCH Initial-touch event.

7.1.6.4 Variable Documentation

7.1.6.4.1 struct [ft_control_interface](#) [ft_control_slider_interface](#)

An interface structure, which contains pointers to the entry points of Slider algorithms. A pointer to this structure must be assigned to any instance of [ft_control_slider_control](#) to define the control behavior.

Overview

7.1.6.5 Slider Control API

7.1.6.5.1 Overview

These functions can be used to set or get the Slider control properties.

A common example definition of the Slider control for all source code examples is as follows:

```
* // Definition of the electrode array used by the control (more info in electrodes)
* const struct ft_electrode * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*   NULL};
*
* // Definition of the Slider control
* const struct ft_control my_slider_control =
* {
*   .interface = &ft_control_slider_interface,
*   .electrodes = control_0_electrodes,
* };
*
*
```

Collaboration diagram for Slider Control API:



Functions

- void `ft_control_slider_register_callback` (const struct `ft_control` *control, `ft_control_slider_callback` callback)
Registers the events handler function.
- uint32_t `ft_control_slider_get_position` (const struct `ft_control` *control)
Get the Slider 'Position' value.
- uint32_t `ft_control_slider_is_touched` (const struct `ft_control` *control)
Get 'Touched' state.
- uint32_t `ft_control_slider_movement_detected` (const struct `ft_control` *control)
Get 'Movement' flag.
- uint32_t `ft_control_slider_get_direction` (const struct `ft_control` *control)
Get 'Direction' flag.
- uint32_t `ft_control_slider_get_invalid_position` (const struct `ft_control` *control)
Get 'Invalid' flag.

7.1.6.5.2 Function Documentation

7.1.6.5.2.1 uint32_t ft_control_slider_get_direction (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

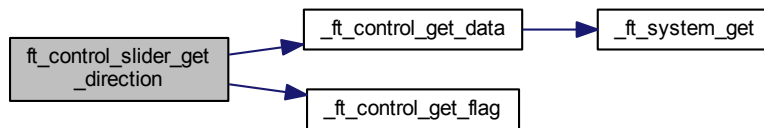
Non-zero value, when a movement is detected in a direction towards higher values. Returns zero, when a movement towards zero is detected. Example:

```

* uint32_t direction;
* // Get direction of Slider control
* direction = ft_control_slider_get_direction(&my_slider_control);
* if(direction)
*     printf("The Slider direction is left.");
* else
*     printf("The Slider direction is right.");
*

```

Here is the call graph for this function:



7.1.6.5.2.2 uint32_t ft_control_slider_get_invalid_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value, when an invalid position was detected, otherwise zero. Example:

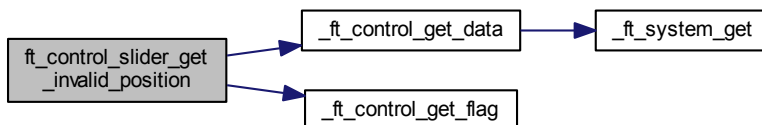
```

* uint32_t invalid_position;
* // Get invalid position of Slider control
* invalid_position = ft_control_slider_get_invalid_position(&
*     my_slider_control);
* if(invalid_position)
*     printf("The Slider control has an invalid position (two fingers touch ?).");
* else
*     printf("The Slider control has a valid position.");
*

```

Overview

Here is the call graph for this function:



7.1.6.5.2.3 uint32_t ft_control_slider_get_position (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Position. The returned value is in the range of zero to the maximum value configured in the [ft_control](#) structure.

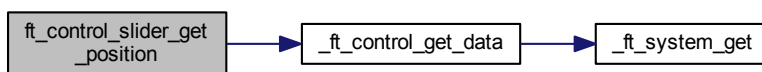
This function retrieves the actual finger position value. Example:

```

* uint32_t position;
* // Get position of Slider control
* position = ft_control_slider_get_position(&my_slider_control);
* printf("Position of Slider control is: %d.", position);
*

```

Here is the call graph for this function:



7.1.6.5.2.4 uint32_t ft_control_slider_is_touched (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

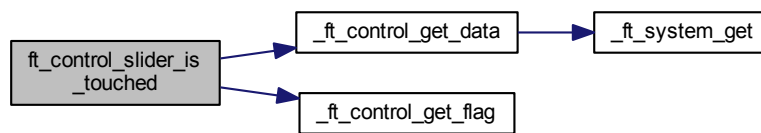
Non-zero value, when the control is currently touched. Example:

```

* uint32_t touched;
* // Get state of Slider control
* touched = ft_control_slider_is_touched(&my_slider_control);
* if(touched)
*     printf("The Slider control is currently touched.");
* else
*     printf("The Slider control is currently not touched.");
*

```

Here is the call graph for this function:



7.1.6.5.2.5 uint32_t ft_control_slider_movement_detected (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

Non-zero value, when the control detects finger movement. Example:

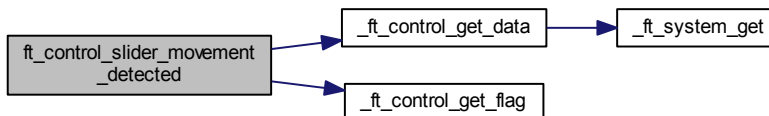
```

* uint32_t movement;
* // Get state of Slider control
* movement = ft_control_slider_movement_detected(&my_slider_control);
* if(movement)
*     printf("The Slider control is currently moving.");
* else
*     printf("The Slider control is currently not moving.");
*

```

Overview

Here is the call graph for this function:



7.1.6.5.2.6 void ft_control_slider_register_callback (const struct ft_control * control, ft_control_slider_callback callback)

Parameters

<i>control</i>	Pointer to the control.
<i>callback</i>	Adress of function to be invoked.

Returns

none

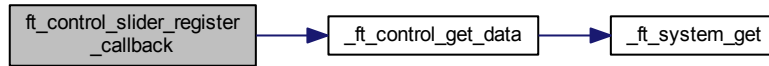
Register the specified callback function as the Slider events handler. Example:

```

*
* //Create the callback function for aslider
* static void my_slider_cb(const struct ft_control *control,
*                          enum ft_control_aslider_event event,
*                          uint32_t position)
* {
*   (void)control;
*   char* event_names[] =
*   {
*     "FT_SLIDER_MOVEMENT",
*     "FT_SLIDER_ALL_RELEASE",
*     "FT_SLIDER_INITIAL_TOUCH",
*   };
*   printf("New slider control event %s on position: %d.", event_names[event], position);
* }
*
* // register the callback function for slider
* ft_control_slider_register_callback(&my_slider_control, my_slider_cb)
* ;
*

```


Here is the call graph for this function:



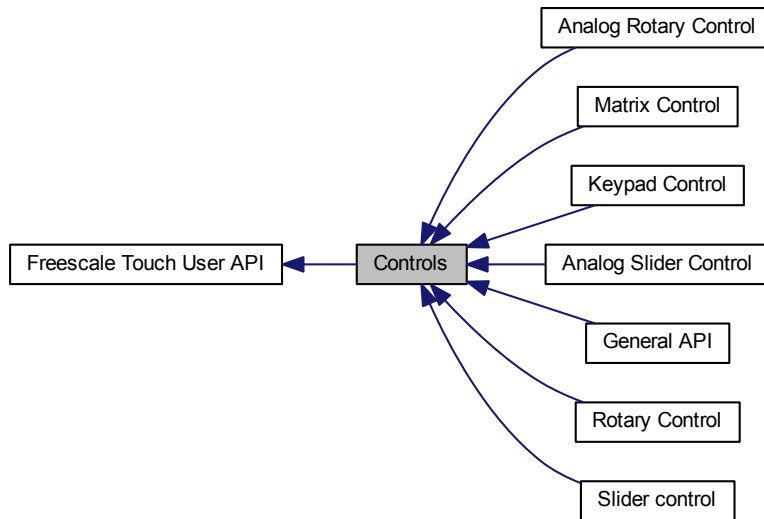
Controls

7.2 Controls

7.2.1 Overview

Controls represent the highest level of abstraction in the finger touch evaluation;

Based on the signal and status information coming from the Electrode layer, the controls calculate finger actions like movement, keyboard touch, hold, and so on. Collaboration diagram for Controls:



Modules

- [Analog Rotary Control](#)
- [Analog Slider Control](#)
- [Keypad Control](#)
- [Matrix Control](#)
- [Rotary Control](#)
- [Slider control](#)
- [General API](#)

7.2.2 General API

7.2.2.1 Overview

General Function definition of controls. Collaboration diagram for General API:



Modules

- [API Functions](#)

Data Structures

- union [ft_control_params](#)
- struct [ft_control](#)

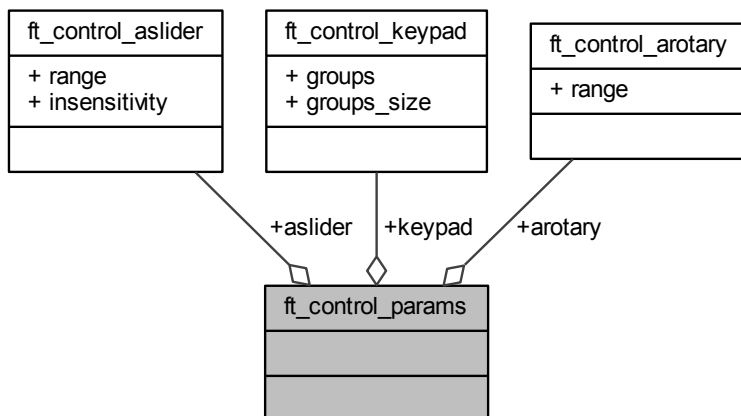
7.2.2.2 Data Structure Documentation

7.2.2.2.1 union ft_control_params

Container, which covers all possible variants of the control parameters. When defining the control setup structure, initialize only one member of this union. Use the member that corresponds to the control type.

Controls

Collaboration diagram for ft_control_params:



Data Fields

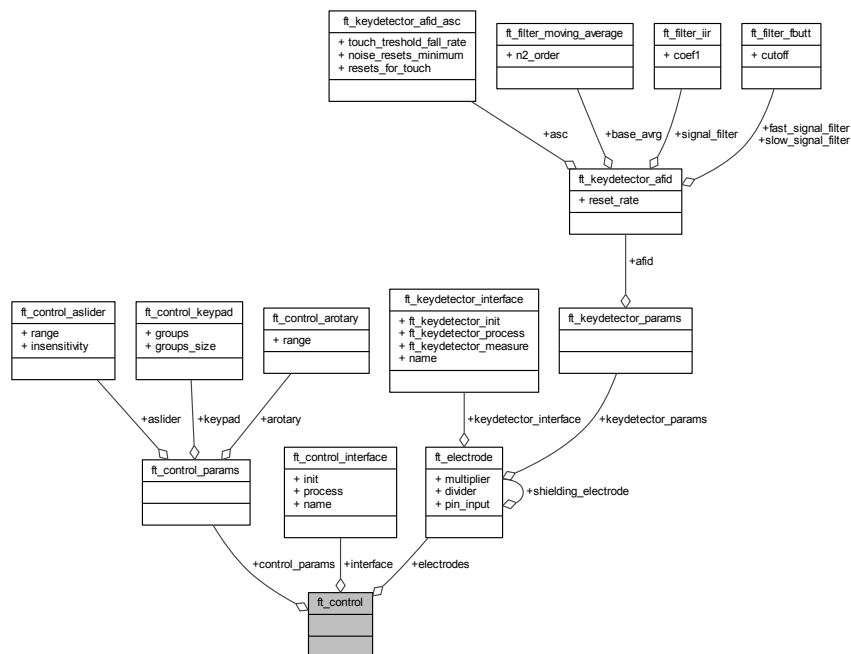
struct ft_ control_arotary *	arotary	
struct ft_ control_aslider *	aslider	
struct ft_ control_keypad *	keypad	

7.2.2.2.2 struct ft_control

The main structure representing the control instance; this structure is used for all control implementations. The type of the control is specified by the "interface" member, which defines the control behavior. Note that the "control_params" must correspond to the control type.

This structure can be allocated in ROM.

Collaboration diagram for ft_control:



Data Fields

union ft_ control_params	control_params	An instance of the control params. Cannot be NULL.
struct ft_electrode *const *	electrodes	List of electrodes. Cannot be NULL.
struct ft_control_ interface *	interface	An instance of the control interface. Cannot be NULL.

Controls

7.2.2.3 API Functions

7.2.2.3.1 Overview

General API functions of the controls. Collaboration diagram for API Functions:



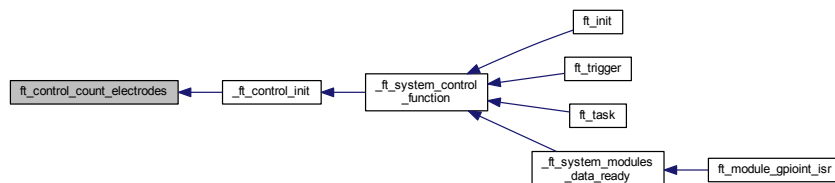
Functions

- void `ft_control_enable` (const struct `ft_control` *control)
Enable control.
- void `ft_control_disable` (const struct `ft_control` *control)
Disable control.
- int32_t `ft_control_get_touch_button` (const struct `ft_control` *control, uint32_t index)
Get touched electrode.
- uint32_t `ft_control_get_electrodes_state` (struct `ft_control` *control)
Get the state of all control electrodes.
- uint32_t `ft_control_count_electrodes` (const struct `ft_control` *control)
- struct `ft_electrode` * `ft_control_get_electrode` (const struct `ft_control` *control, uint32_t index)
Return the electrode by index.

7.2.2.3.2 Function Documentation

7.2.2.3.2.1 uint32_t ft_control_count_electrodes (const struct ft_control * control)

Here is the caller graph for this function:



7.2.2.3.2.2 void ft_control_disable (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control instance.
----------------	----------------------------------

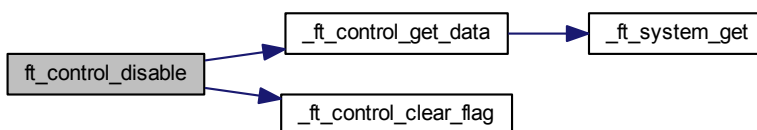
Returns

none

Disables the control operation by clearing the FT_CONTROL_ENABLE_FLAG. This is an example of disabling the control in the FT library:

```
* // The FT control my_ft_control_keypad is disabled
* ft_control_disable(&my_ft_control_keypad);
*
```

Here is the call graph for this function:


7.2.2.3.2.3 void ft_control_enable (const struct ft_control * control)

Parameters

<i>control</i>	Pointer to the control instance.
----------------	----------------------------------

Returns

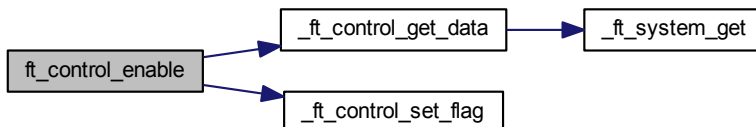
none

Enables the control operation by setting the FT_CONTROL_ENABLE_FLAG. This is an example of enabling the control in the FT library:

```
* // The FT control my_ft_control_keypad is enabled
* ft_control_enable(&my_ft_control_keypad);
*
```

Controls

Here is the call graph for this function:



7.2.2.3.2.4 struct ft_electrode* ft_control_get_electrode (const struct ft_control * control, uint32_t index)

Parameters

<i>control</i>	Pointer to the control.
<i>index</i>	

Returns

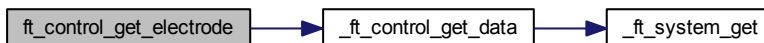
Pointer to the electrode instance retrieved from control's electrode list. This is an example of getting the electrode pointer of control by index in the FT library:

```

* // Get the pointer of electrode on index 2 for my_control
* ft_electrode *my_electrode = ft_control_get_electrode(&my_control, 2
* );
*

```

Here is the call graph for this function:



7.2.2.3.2.5 uint32_t ft_control_get_electrodes_state (struct ft_control * control)

Parameters

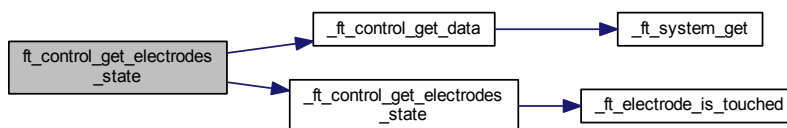
<i>control</i>	Pointer to the control data.
----------------	------------------------------

Returns

This function returns a bit-mask value, where each bit represents one control electrode. Logic 1 in the returned value represents a touched electrode.

```
* uint32_t touched_electrode = 0;
* touched_electrode = ft_control_get_electrodes_state(&my_control);
* printf("The electrode state is following: 0x%X in my control.", touched_electrode);
*
```

Here is the call graph for this function:


7.2.2.3.2.6 int32_t ft_control_get_touch_button (const struct ft_control * control, uint32_t index)
Parameters

<i>control</i>	Pointer to the control.
<i>index</i>	Index of the first electrode to be probed. Use 0 during the first call. Use the last-returned index+1 to get the next touched electrode.

Returns

Index of the touched electrode, or FT_FAILURE when no electrode is touched.

Use this function to determine, which control electrodes are currently touched. This is an example of getting the touched electrodes of control in the FT library:

```
* int32_t last_touched_electrode = 0;
* uint32_t electrode_count = ft_control_count_electrodes(&my_control);
* last_touched_electrode = ft_control_get_touch_button(&my_control,
*   last_touched_electrode);
* while(last_touched_electrode != FT_FAILURE)
* {
```

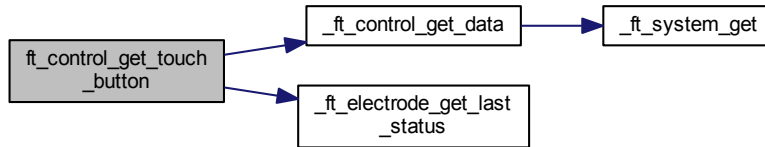
Controls

```

*   printf("The electrode %d in my control is touched", last_touched_electrode);
*   last_touched_electrode = ft_control_get_touch_button(&my_control,
*   last_touched_electrode);
* }
*

```

Here is the call graph for this function:



7.3 Electrodes

7.3.1 Overview

Electrodes are data objects that are used by data-acquisition algorithms to store the per-electrode data, as well as the resulting signal and touch / timestamp information.

Each Electrode provides at minimum the processed and normalized signal value, the baseline value, and touch / timestamp buffer containing the time of last few touch and release events. All such common information are contained in the [ft_electrode](#) structure type. Also, the electrode contains information about the key detector used to detect touches for this physical electrode (this is a mandatory field in the electrode definition). This has the advantage that each electrode has its own setting of key detector, independent on the module used. It contains information about hardware pin, immediate touch status, and time stamps of the last few touch or release events. Collaboration diagram for Electrodes:



Modules

- [API Functions](#)

Data Structures

- [struct ft_electrode_status](#)
- [struct ft_electrode](#)

Enumerations

- [enum ft_electrode_state](#) {
[FT_ELECTRODE_STATE_INIT](#),
[FT_ELECTRODE_STATE_RELEASE](#),
[FT_ELECTRODE_STATE_TOUCH](#) }

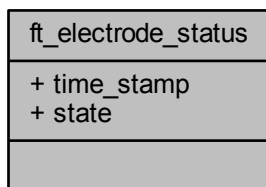
Electrodes

7.3.2 Data Structure Documentation

7.3.2.1 struct ft_electrode_status

Electrode status structure holding one entry in the touch-timestamp buffer. An array of this structure type is a part of each Electrode, and contains last few touch or release events detected on the electrode.

Collaboration diagram for ft_electrode_status:



Data Fields

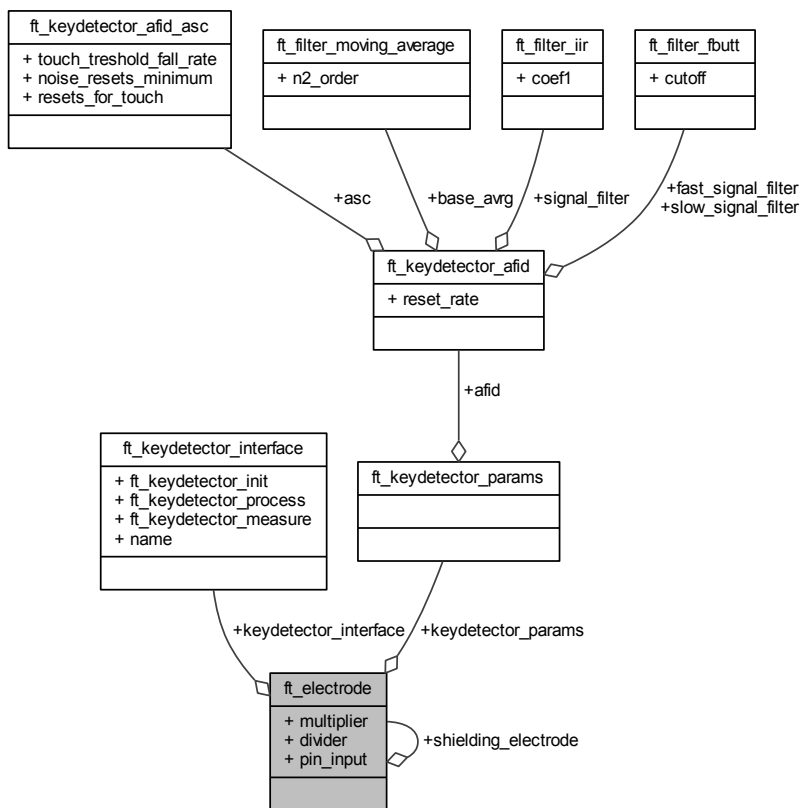
uint8_t	state	Electrode's state.
uint32_t	time_stamp	Time stamp.

7.3.2.2 struct ft_electrode

The main structure representing the Electrode instance. There are all the parameters needed to define the behavior of the Freescale Touch electrode, including its key detector, hardware pins, multiplier / divider to normalize the signal, and the optional shielding electrode.

This structure can be allocated in ROM.

Collaboration diagram for ft_electrode:



Data Fields

uint8_t	divider	Divider.
struct ft_keydetector_ _interface *	keydetector_- interface	Pointer to Key Detector interface.

Electrodes

union <code>ft_keydetector_</code> <code>_params</code>	<code>keydetector_</code> <code>params</code>	Pointer to Key Detector params.
<code>uint8_t</code>	<code>multiplier</code>	Multiplier.
<code>uint32_t</code>	<code>pin_input</code>	Input pin.
struct <code>ft_electrode *</code>	<code>shielding_</code> <code>electrode</code>	Shielding electrode.

7.3.3 Enumeration Type Documentation

7.3.3.1 enum `ft_electrode_state`

Electrode states.

Enumerator

FT_ELECTRODE_STATE_INIT Initial state; Not enough data for the touch-detection algorithm yet.

FT_ELECTRODE_STATE_RELEASE Release state; A signal is near to the baseline.

FT_ELECTRODE_STATE_TOUCH Touch state; the selected algorithm has decided that a finger is present.

7.3.4 API Functions

7.3.4.1 Overview

General Function definition of the electrodes. Collaboration diagram for API Functions:



Functions

- `int32_t ft_electrode_enable (const struct ft_electrode *electrode)`
Enable the electrode. The function is used to enable the electrode; it should be used after the FT initialization, because the default state after the startup of the FT is electrode disabled.
- `int32_t ft_electrode_disable (const struct ft_electrode *electrode)`
Disable the electrode.
- `uint32_t ft_electrode_get_signal (const struct ft_electrode *electrode)`
Get the normalized and processed electrode signal.
- `int32_t ft_electrode_get_last_status (const struct ft_electrode *electrode)`
Get the last known electrode status.
- `uint32_t ft_electrode_get_time_offset (const struct ft_electrode *electrode)`
Get the time from the last electrode event.
- `uint32_t ft_electrode_get_last_time_stamp (const struct ft_electrode *electrode)`
Get the last known electrode time stamp.
- `uint32_t ft_electrode_get_raw_signal (const struct ft_electrode *electrode)`
Get the raw electrode signal.

7.3.4.2 Function Documentation

7.3.4.2.1 `int32_t ft_electrode_disable (const struct ft_electrode * electrode)`

Parameters

<i>electrode</i>	Pointer to the electrode params that identify the electrode.
------------------	--

Electrodes

Returns

result of operation [ft_result](#). This is an example of using this function in code:

```
* // Disable electrode_0 that is defined in the setup of FT
* if(ft_electrode_disable(&electrode_0) != FT_SUCCESS)
* {
*     printf("Disable electrode_0 failed.");
* }
*
```

7.3.4.2.2 int32_t ft_electrode_enable (const struct ft_electrode * *electrode*)

Parameters

<i>electrode</i>	Pointer to the electrode params that identify the electrode.
------------------	--

Returns

result of operation [ft_result](#). This is an example of using this function in the code:

```
//The electrode that is defined in the setup of FT after the initialization must be enabled.
if (ft_init(&system_0, ft_memory_pool, sizeof(ft_memory_pool)) <
    FT_SUCCESS)
{
    while(1); // add code to handle this error
}
// Enable electrode_0 that is defined in the setup of FT
if(ft_electrode_enable(&electrode_0) != FT_SUCCESS)
{
    printf("Enable electrode_0 failed.");
}
```

7.3.4.2.3 int32_t ft_electrode_get_last_status (const struct ft_electrode * *electrode*)

Parameters

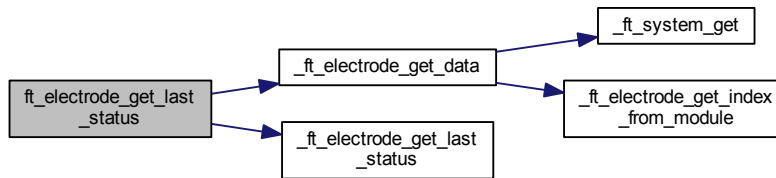
<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

Current electrode status.

```
* // Get the latest status of my_electrode
* char * electrode_state_name[3] =
* {
*     "Initialize",
*     "Released",
*     "Touched"
* };
* uint32_t state = ft_electrode_get_last_status(&my_electrode);
* printf("The my_electrode last status is: %s.", electrode_state_name[state]);
*
```


Here is the call graph for this function:



7.3.4.2.4 uint32_t ft_electrode_get_last_time_stamp (const struct ft_electrode * *electrode*)

Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

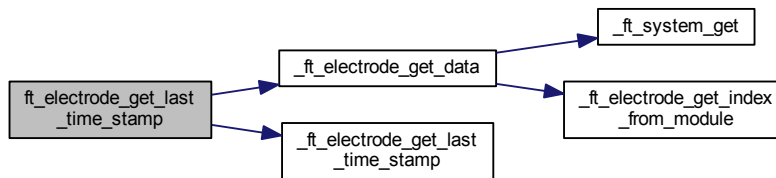
Current electrode status.

```

* // Get the time stamp of the last change of the electrode status
* uint32_t time = ft_electrode_get_last_time_stamp(&my_electrode);
* printf("The my_electrode last status change was at: %d ms .", time);
*

```

Here is the call graph for this function:



7.3.4.2.5 uint32_t ft_electrode_get_raw_signal (const struct ft_electrode * *electrode*)

Electrodes

Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

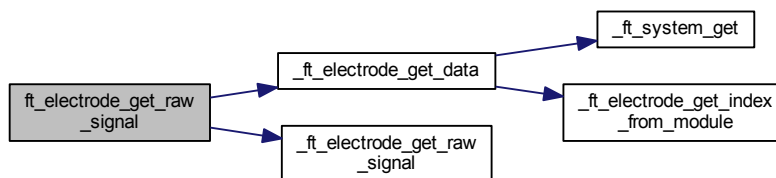
Returns

electrode Signal, as it is measured by the physical module.

The raw signal is used internally by the filtering and normalization algorithms to calculate the real electrode signal value, which is good to be compared with the signals coming from other electrodes.

```
* // Get the current raw signal of my_electrode
* printf("The my_electrode has raw signal: %d.", ft_electrode_get_raw_signal(&
*     my_electrode));
*
```

Here is the call graph for this function:



7.3.4.2.6 uint32_t ft_electrode_get_signal (const struct ft_electrode * *electrode*)

Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

electrode signal calculated from the last raw value measured.

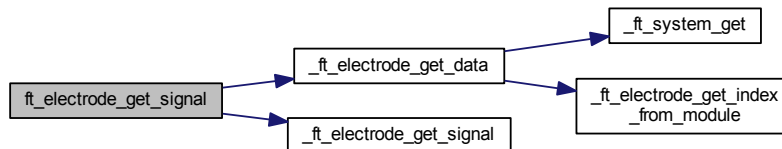
The signal value is calculated from the raw electrode capacitance or other physical signal by applying the filtering and normalization algorithms. This signal is used by the "analog" [Controls](#) that estimate the finger position based on the signal value, rather than on a simple touch / release status. This is an example of using this function in the code:

```

* // Get current signal of my_electrode
* printf("The my_electrode has signal: %d.", ft_electrode_get_signal(&my_electrode)
* );
*

```

Here is the call graph for this function:



7.3.4.2.7 uint32_t ft_electrode_get_time_offset (const struct ft_electrode * electrode)

Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

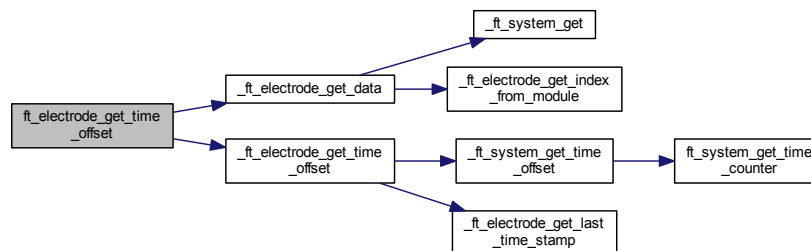
Time from the last electrode event.

```

* // Get the time offset from the last change of the electrode status
* uint32_t offset = ft_electrode_get_time_offset(&my_electrode);
* printf("The my_electrode last status change has been before: %d ms .", offset);
*

```

Here is the call graph for this function:

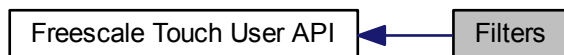


Filters

7.4 Filters

7.4.1 Overview

The filters data structure that is used in the Freescale Touch library. Collaboration diagram for Filters:



Data Structures

- struct [ft_filter_fbutt](#)
- struct [ft_filter_iir](#)
- struct [ft_filter_dctracker](#)
- struct [ft_filter_moving_average](#)

Macros

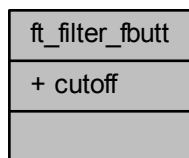
- #define [FT_FILTER_MOVING_AVERAGE_MAX_ORDER](#)

7.4.2 Data Structure Documentation

7.4.2.1 struct ft_filter_fbutt

The butterworth filter input parameters.

Collaboration diagram for ft_filter_fbutt:



Data Fields

int32_t	cutoff	The coefficient for the implemented butterworth filter polynomial.
---------	--------	--

7.4.2.2 struct ft_filter_iir

The IIR filter input parameters.

Collaboration diagram for ft_filter_iir:



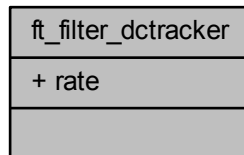
Data Fields

uint8_t	coef1	Scale of the current and previous signals. When the coef is higher, the current signal has less strength.
---------	-------	---

7.4.2.3 struct ft_filter_dctracker

The DC tracker filter input parameters.

Collaboration diagram for ft_filter_dctracker:



Filters

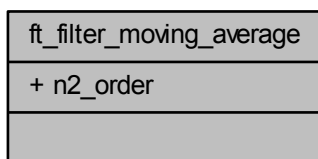
Data Fields

uint8_t	rate	Rate of how fast the baseline is updated. The rate should be defined as a modulo of the system period.
---------	------	--

7.4.2.4 struct ft_filter_moving_average

The moving average filter input parameters.

Collaboration diagram for ft_filter_moving_average:



Data Fields

int32_t	n2_order	The order 2^n moving average filter
---------	----------	---------------------------------------

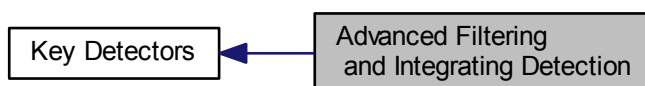
7.4.3 Macro Definition Documentation

7.4.3.1 #define FT_FILTER_MOVING_AVERAGE_MAX_ORDER

7.4.4 Advanced Filtering and Integrating Detection

7.4.4.1 Overview

The AFID (Advanced Filtering and Integrating Detection) key detector is based on using two IIR filters with different depths (one short / fast, the other long / slow) and on integrating the difference between the two filtered signals. The algorithm uses two thresholds: the touch threshold and the release threshold. The touch threshold is defined in the sensitivity register. The release threshold has a twice lower level than the touch threshold. If the integrated signal is higher than the touch threshold, or lower than the release threshold, then the integrated signal is reset. The touch state is reported for the electrode when the first touch reset is detected. The release state is reported when as many release resets are detected as the touch resets were detected during the previous touch state. Collaboration diagram for Advanced Filtering and Integrating Detection:



Data Structures

- struct [ft_keydetector_afid_asc](#)
- struct [ft_keydetector_afid](#)

Macros

- #define [FT_KEYDETECTOR_AFID_ASC_DEFAULT](#)

Variables

- struct [ft_keydetector_interface](#) [ft_keydetector_afid_interface](#)

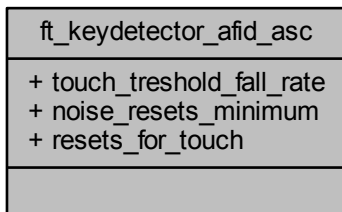
7.4.4.2 Data Structure Documentation

7.4.4.2.1 struct [ft_keydetector_afid_asc](#)

AFID Automatic Sensitive Calibration structure; This structure is used to define the parameters of evaluating the AFID process flow. You can manage your own setup of parameters, or use the default setting in the [FT_KEYDETECTOR_AFID_ASC_DEFAULT](#). This structure must be filled in.

Filters

Collaboration diagram for ft_keydetector_afid_asc:



Data Fields

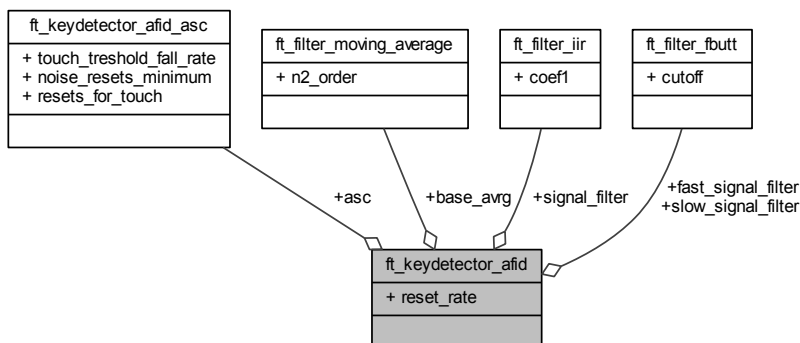
uint32_t	noise_resets_minimum	Noise Resets Minimum
int16_t	resets_for_touch	Number of resets required for touch
int16_t	touch_treshold_fall_rate	Rate of how often the touch threshold can fall

7.4.4.2.2 struct ft_keydetector_afid

The main structure representing the AFID key detector. An instance of this data type represents the AFID key detector. Consisting of parameters of filters, the AFID automatic sensitive calibration, and update rate.

You're responsible to initialize all the members before registering the AFID in the module. This structure can be allocated in ROM.

Collaboration diagram for ft_keydetector_afid:



Data Fields

struct ft_keydetector_afid_asc	asc	ASC structure for the AFID detector.
struct ft_filter_moving_average	base_avg	Settings of the moving average filter for the baseline in release state of electrode.
struct ft_filter_fbtt	fast_signal_filter	Signal butterworth signal (fast).
uint16_t	reset_rate	
struct ft_filter_iir	signal_filter	Coefficient of the input IIR signal filter, used to suppress high frequency noise.
struct ft_filter_fbtt	slow_signal_filter	Baseline butterworth signal (slow).

7.4.4.3 Macro Definition Documentation

7.4.4.3.1 #define FT_KEYDETECTOR_AFID_ASC_DEFAULT

AFID Automatic Sensitive Calibration default ASC settings:

- touch_treshold_fall_rate (default 255)
- noise_resets_minimum (default 128)
- resets_for_touch (default 6) This default values for AFID ASC definition for example:

```
const struct ft_keydetector_afid keydec =
```

Filters

```

{
    .signal_filter = {
        .cutoff = 8
    },
    .baseline_filter = {
        .cutoff = 4
    },
    .reset_rate = 1000,
    .asc = FT_KEYDETECTOR_AFID_ASC_DEFAULT,
};

```

7.4.4.4 Variable Documentation

7.4.4.4.1 struct ft_keydetector_interface ft_keydetector_afid_interface

AFID key detector interface structure.

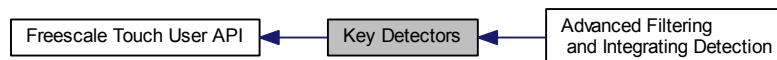
7.5 Key Detectors

7.5.1 Overview

Key Detectors represent different types of signal-processing algorithms; the primary purpose of a key detector algorithm is to determine, whether an electrode has been touched or not, calculate the normalized signal, and provide all these information to the [Controls](#) layer. The Controls layer is then able to detect much more complex finger gestures, such as a slider swing or a key press within a multiplexed keypad.

As an input, the Key Detector gets the raw electrode signal value obtained from the data-acquisition algorithm, wrapped by one of the [Modules](#) instance. The output values and intermediate calculated parameters needed by the Key Detector layer are contained within a structure type, derived from the [ft_electrode](#) type. See more information in the [Electrodes](#) chapter.

In addition to signal processing, the Key Detector also detects, reports, and acts on fault conditions during the scanning process. Two main fault conditions are reported as electrode short-circuit to supply voltage (capacitance too small), or short-circuit to ground (capacitance too high). Collaboration diagram for Key Detectors:



Modules

- [Advanced Filtering and Integrating Detection](#)

Key Detectors

7.5.2 GPIO module

7.5.2.1 Overview

GPIO module uses the MCU's General Purpose pins and Timer. Collaboration diagram for GPIO module:



Data Structures

- struct [ft_module_gpio_user_interface](#)
- struct [ft_module_gpio_params](#)

Variables

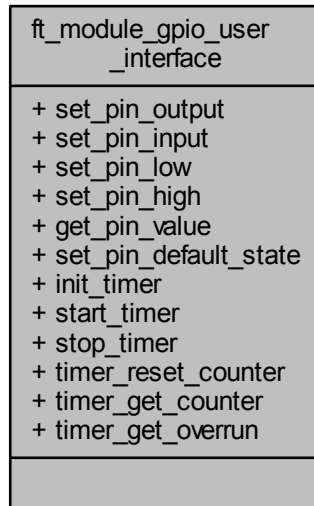
- struct [ft_module_interface](#) [ft_module_gpio_interface](#)

7.5.2.2 Data Structure Documentation

7.5.2.2.1 struct [ft_module_gpio_user_interface](#)

Gpio user's interface, which is used by the GPIO modules. All of these functions must be implemented in the application.

Collaboration diagram for ft_module_gpio_user_interface:



Data Fields

- void(* [set_pin_output](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_input](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_low](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_high](#))(uint32_t port, uint32_t pin)
- uint32_t(* [get_pin_value](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_default_state](#))(uint32_t port, uint32_t pin)
- void(* [init_timer](#))(void)
- void(* [start_timer](#))(void)
- void(* [stop_timer](#))(void)
- void(* [timer_reset_counter](#))(void)
- uint32_t(* [timer_get_counter](#))(void)
- uint32_t(* [timer_get_overrun](#))(void)

7.5.2.2.1.1 Field Documentation

7.5.2.2.1.1.1 `uint32_t(* ft_module_gpio_user_interface::get_pin_value)(uint32_t port, uint32_t pin)`

Get pin value

7.5.2.2.1.1.2 `void(* ft_module_gpio_user_interface::init_timer)(void)`

Init timer

Key Detectors

7.5.2.2.1.1.3 void(* ft_module_gpio_user_interface::set_pin_default_state)(uint32_t port, uint32_t pin)

Set pin to default state when it's not being measured

7.5.2.2.1.1.4 void(* ft_module_gpio_user_interface::set_pin_high)(uint32_t port, uint32_t pin)

Set pin to logic high

7.5.2.2.1.1.5 void(* ft_module_gpio_user_interface::set_pin_input)(uint32_t port, uint32_t pin)

Set pin direction to input

7.5.2.2.1.1.6 void(* ft_module_gpio_user_interface::set_pin_low)(uint32_t port, uint32_t pin)

Set pin to logic low

7.5.2.2.1.1.7 void(* ft_module_gpio_user_interface::set_pin_output)(uint32_t port, uint32_t pin)

Set pin direction to output

7.5.2.2.1.1.8 void(* ft_module_gpio_user_interface::start_timer)(void)

Start timer

7.5.2.2.1.1.9 void(* ft_module_gpio_user_interface::stop_timer)(void)

Stop timer

7.5.2.2.1.1.10 uint32_t(* ft_module_gpio_user_interface::timer_get_counter)(void)

Get timer counter

7.5.2.2.1.1.11 uint32_t(* ft_module_gpio_user_interface::timer_get_overrun)(void)

Get timer overrun

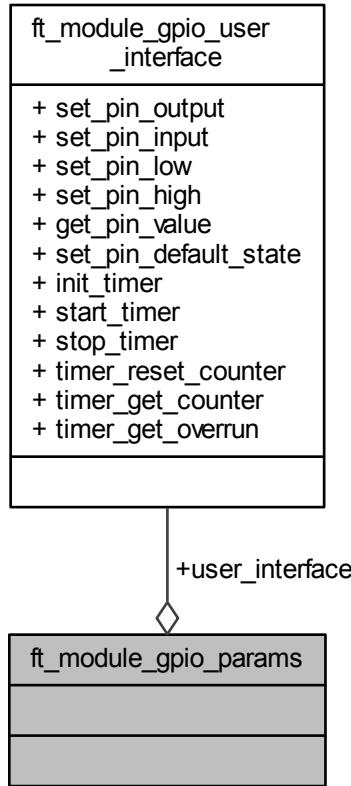
7.5.2.2.1.1.12 void(* ft_module_gpio_user_interface::timer_reset_counter)(void)

Reset timer counter

7.5.2.2.2 struct ft_module_gpio_params

GPIO module, which uses the `??interrupts?` port to sample a signal from the running timer counter.

Collaboration diagram for ft_module_gpio_params:



Data Fields

struct ft_module_ gpio_user_ interface *	user_interface	
--	----------------	--

7.5.2.3 Variable Documentation

7.5.2.3.1 struct ft_module_interface ft_module_gpio_interface

Can't be NULL.

interface gpio module

Key Detectors

7.5.3 GPIO interrupt module

7.5.3.1 Overview

The GPIO module uses the General Purpose pins and Timer of the MCU. Works on GPIO pins with interrupt. Collaboration diagram for GPIO interrupt module:



Data Structures

- struct [ft_module_gpioint_user_interface](#)
- struct [ft_module_gpioint_params](#)

Functions

- void [ft_module_gpioint_isr](#) (const struct [ft_module](#) *module)

This interrupt handler must be invoked from the user's port interrupt ISR. There can be other pins on the same port which can invoke the interrupt; therefore, it is up to the application to decode which pin caused the interrupt. For example, if there's a button on the PTA3 and an electrode on the PTA4, the PORTA ISR handler must decode, whether the interrupt was caused by the PTA3 or PTA4. Invoke the [ft_module_gpioint_isr\(\)](#) only if any of the GPIO modules' electrodes caused an interrupt.
- void [ft_module_gpioint_overflow_isr](#) (const struct [ft_module](#) *module)

This interrupt handler should be invoked from the user's timer interrupt ISR. It is not mandatory to call this function, but it's designed to avoid, stuck to the Freescale Touch GPIO Interrupt module. It should be called after the user-defined maximal timeout for one measurement.

Variables

- struct [ft_module_interface](#) [ft_module_gpioint_interface](#)

7.5.3.2 Data Structure Documentation

7.5.3.2.1 struct [ft_module_gpioint_user_interface](#)

GPIO user's interface, which is used by the GPIO modules. All of these functions must be implemented in the application.

Collaboration diagram for ft_module_gpioint_user_interface:



Data Fields

- void(* [set_pin_output](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_input](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_low](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_high](#))(uint32_t port, uint32_t pin)
- void(* [init_pin](#))(uint32_t port, uint32_t pin)
- void(* [set_pin_interrupt](#))(uint32_t port, uint32_t pin)
- void(* [clear_pin_interrupt](#))(uint32_t port, uint32_t pin)
- void(* [init_timer](#))(void)
- void(* [start_timer](#))(void)
- void(* [stop_timer](#))(void)
- void(* [timer_reset_counter](#))(void)
- uint32_t(* [timer_get_counter](#))(void)

7.5.3.2.1.1 Field Documentation

7.5.3.2.1.1.1 void(* ft_module_gpioint_user_interface::clear_pin_interrupt)(uint32_t port, uint32_t pin)

Disable the pin to generate an interrupt

7.5.3.2.1.1.2 void(* ft_module_gpioint_user_interface::init_pin)(uint32_t port, uint32_t pin)

Initialize the pin to a state ready for measurement

Key Detectors

7.5.3.2.1.1.3 void(* ft_module_gpoint_user_interface::init_timer)(void)

Init timer

7.5.3.2.1.1.4 void(* ft_module_gpoint_user_interface::set_pin_high)(uint32_t port, uint32_t pin)

Set the pin to logic high

7.5.3.2.1.1.5 void(* ft_module_gpoint_user_interface::set_pin_input)(uint32_t port, uint32_t pin)

Set the pin direction to input

7.5.3.2.1.1.6 void(* ft_module_gpoint_user_interface::set_pin_interrupt)(uint32_t port, uint32_t pin)

Enable the pin to generate an interrupt

7.5.3.2.1.1.7 void(* ft_module_gpoint_user_interface::set_pin_low)(uint32_t port, uint32_t pin)

Set the pin to logic low

7.5.3.2.1.1.8 void(* ft_module_gpoint_user_interface::set_pin_output)(uint32_t port, uint32_t pin)

Set the pin direction to output

7.5.3.2.1.1.9 void(* ft_module_gpoint_user_interface::start_timer)(void)

Start timer

7.5.3.2.1.1.10 void(* ft_module_gpoint_user_interface::stop_timer)(void)

Stop timer

7.5.3.2.1.1.11 uint32_t(* ft_module_gpoint_user_interface::timer_get_counter)(void)

Get timer counter

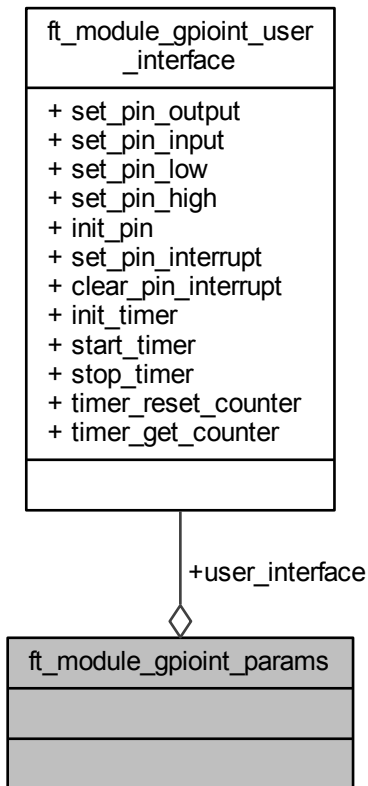
7.5.3.2.1.1.12 void(* ft_module_gpoint_user_interface::timer_reset_counter)(void)

Reset timer counter

7.5.3.2.2 struct ft_module_gpoint_params

GPIO interrupt module, which uses the port interrupts to sample a signal from the running timer counter.

Collaboration diagram for ft_module_gpoint_params:



Data Fields

struct ft_module_ gpoint_user_ interface *	user_interface	
--	----------------	--

7.5.3.3 Function Documentation

7.5.3.3.1 void ft_module_gpoint_isr (const struct ft_module * *module*)

Key Detectors

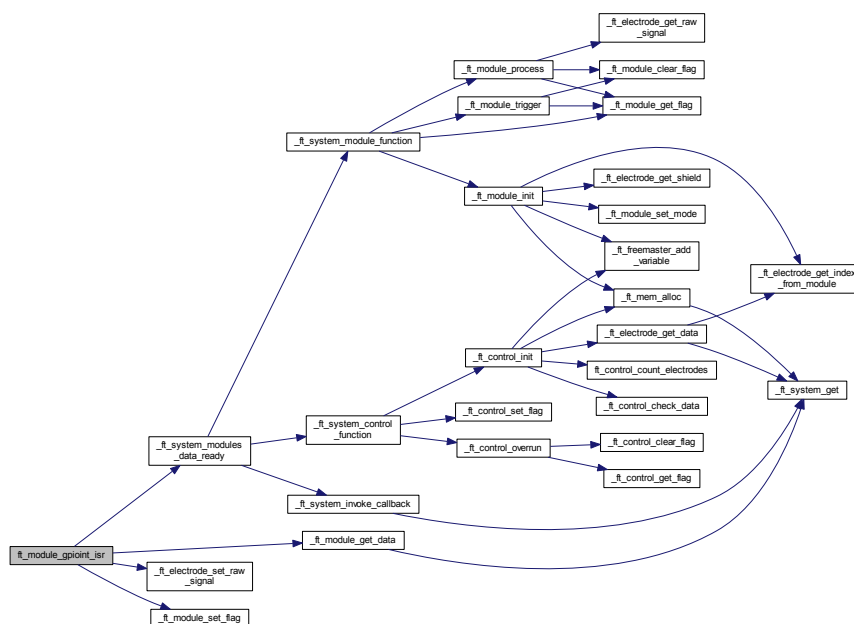
Parameters

<i>module</i>	Pointer to the module that invokes the interrupt; it depends on the user application to handle the right value.
---------------	---

Returns

None.

Here is the call graph for this function:



7.5.3.3.2 void ft_module_gpio_irq_overflow_isr (const struct ft_module * *module*)

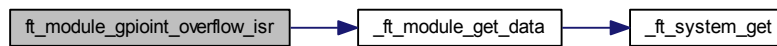
Parameters

<i>module</i>	Pointer to the module that invokes the interrupt; it depends on the user application to handle the right value.
---------------	---

Returns

None.

Here is the call graph for this function:



7.5.3.4 Variable Documentation

7.5.3.4.1 struct ft_module_interface ft_module_gpoint_interface

Can't be NULL.

interface gpio module

Key Detectors

7.5.4 TSI module

7.5.4.1 Overview

The TSI module describes the hardware configuration and control of the elementary functionality of the TSI peripheral; it covers all versions of the TSI peripheral by a generic low-level driver API.

The TSI Basic module is designed for processors that have the hardware TSI module version 1, 2, or 4 (for example Kinetis L).

The module also handles the NOISE mode supported by the TSI v4 (Kinetis L). Collaboration diagram for TSI module:



Data Structures

- struct [ft_module_tsi_noise](#)
- struct [ft_module_tsi_params](#)

Variables

- struct [ft_module_interface](#) [ft_module_tsi_interface](#)

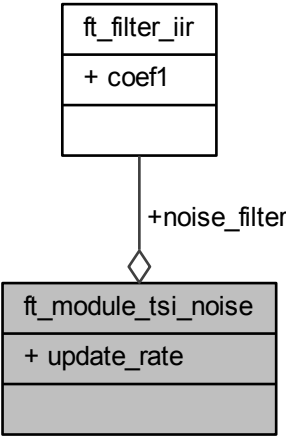
7.5.4.2 Data Structure Documentation

7.5.4.2.1 struct [ft_module_tsi_noise](#)

The structure represents the Noise detection of the TSI v4 module. An instance of this data type represents the Noise detection of the TSI v4 module. It contains the parameters of Noise filters automatic sensitive calibration.

You must initialize all the members before registering the noise in the module. This structure can be allocated in ROM.

Collaboration diagram for ft_module_tsi_noise:



Data Fields

struct ft_filter_iir	noise_filter	
uint8_t	update_rate	

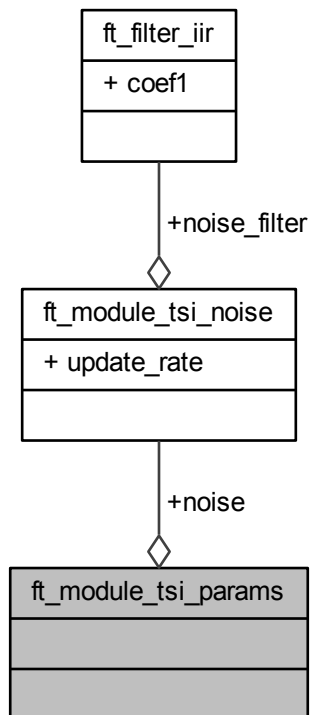
7.5.4.2.2 struct ft_module_tsi_params

The main structure representing the Noise detection of the TSI v4 module. An instance of this data type represents the Noise detection of the TSI v4 module. It contains the parameters of the Noise filters automatic sensitive calibration.

You must initialize all the members before registering the Noise in the module. This structure can be allocated in ROM.

Key Detectors

Collaboration diagram for ft_module_tsi_params:



Data Fields

struct	noise	
ft_module_tsi-noise		

7.5.4.3 Variable Documentation

7.5.4.3.1 struct ft_module_interface ft_module_tsi_interface

The TSI module interface structure. Can't be NULL.

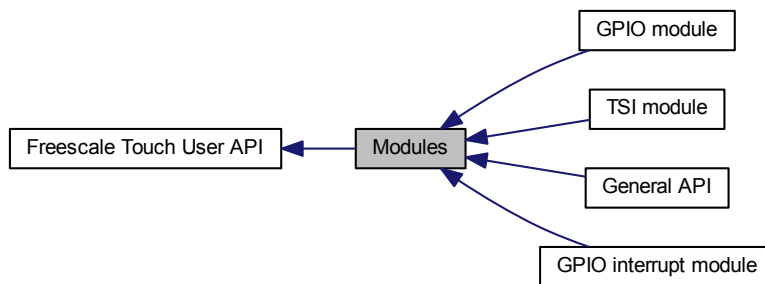
interface tsi module

7.6 Modules

7.6.1 Overview

Modules represent the data-acquisition layer in the Freescale Touch system; it is the layer that is tightly coupled with the hardware module available on the Freescale MCU device.

Each Module implements a set of functions contained in the [ft_module_interface](#) structure. This interface is used by the system to process all modules in a generic way during the data-acquisition or data-processing phases. Collaboration diagram for Modules:



Modules

- [GPIO module](#)
- [GPIO interrupt module](#)
- [TSI module](#)
- [General API](#)

Modules

7.6.2 General API

7.6.2.1 Overview

General Function definition of the modules. Collaboration diagram for General API:



Modules

- [API Functions](#)

Data Structures

- union [ft_module_params](#)
- struct [ft_module](#)

Enumerations

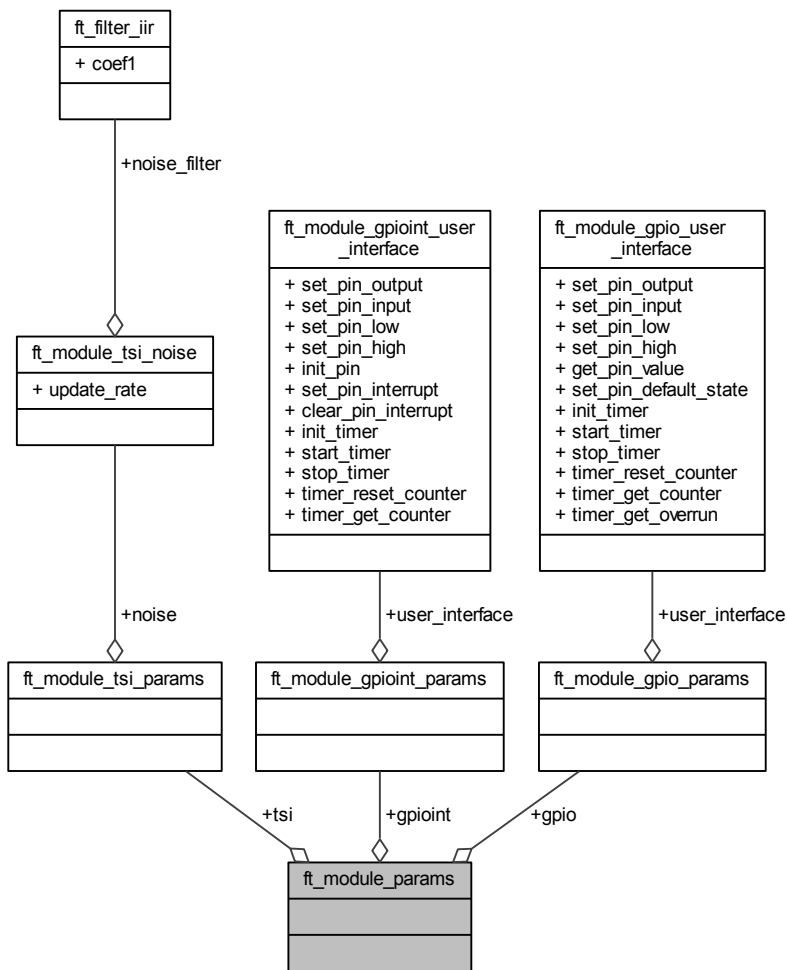
- enum [ft_module_mode](#) {
[FT_MODULE_MODE_NORMAL](#),
[FT_MODULE_MODE_PROXIMITY](#),
[FT_MODULE_MODE_LOW_POWER](#) }
- enum [ft_module_flags](#) {
[FT_MODULE_NEW_DATA_FLAG](#),
[FT_MODULE_TRIGGER_DISABLED_FLAG](#),
[FT_MODULE_DIGITAL_RESULTS_FLAG](#) }

7.6.2.2 Data Structure Documentation

7.6.2.2.1 union [ft_module_params](#)

Container that covers all possible variants of the module parameters.

Collaboration diagram for ft_module_params:



Data Fields

struct ft_module_- gpio_params *	gpio	Pointer to the GPIO module specific parameters.
---	------	---

Modules

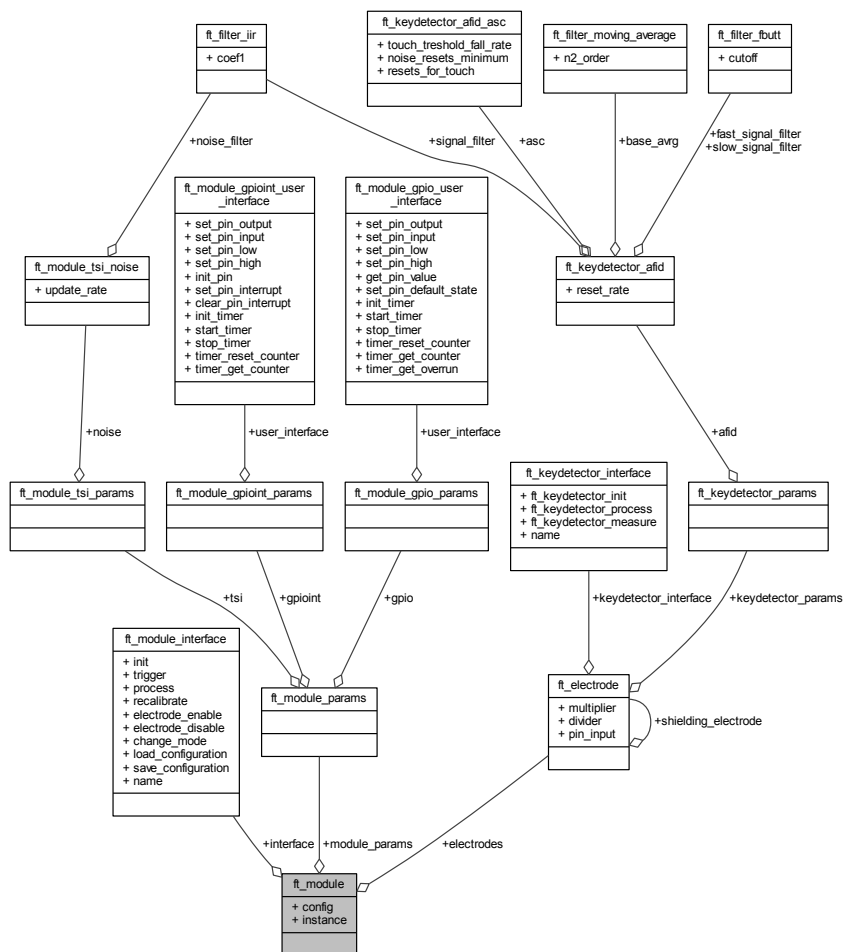
<pre> struct ft_module_ gpoint_params *</pre>	gpoint	Pointer to the GPIO interrupt module specific parameters.
<pre> struct ft_module_tsi- _params *</pre>	tsi	Pointer to the TSI module specific parameters.

7.6.2.2.2 struct ft_module

The main structure representing the Module instance; this structure is used as a base for all module implementations. The type of the module is specified by selecting the right module interface.

This structure can be allocated in ROM.

Collaboration diagram for ft_module:



Data Fields

void *	config	A pointer to the hardware configuration. Can't be NULL.
struct ft_electrode *const *	electrodes	A pointer to the list of electrodes. Can't be NULL.

Modules

<code>uint8_t</code>	<code>instance</code>	An instance of the module.
<code>struct ft_module_ interface *</code>	<code>interface</code>	Module interface. Can't be NULL.
<code>union ft_module_ params</code>	<code>module_ params</code>	An instance module params. Can't be NULL.

7.6.2.3 Enumeration Type Documentation

7.6.2.3.1 `enum ft_module_flags`

Generic flags for Module processing.

Enumerator

FT_MODULE_NEW_DATA_FLAG The new data is ready to be processed.

FT_MODULE_TRIGGER_DISABLED_FLAG Disables the trigger for the current module (in fact, the module is disabled).

FT_MODULE_DIGITAL_RESULTS_FLAG The digital data only flag (only touch / release information - no analog value).

7.6.2.3.2 `enum ft_module_mode`

Module's modes.

Enumerator

FT_MODULE_MODE_NORMAL The module is in a standard touch measure mode.

FT_MODULE_MODE_PROXIMITY The module is in a proximity mode.

FT_MODULE_MODE_LOW_POWER The module is in a low-power mode.

7.6.2.4 API Functions

7.6.2.4.1 Overview

General API functions of the modules. Collaboration diagram for API Functions:



Functions

- `uint32_t ft_module_recalibrate` (const struct `ft_module` *module, void *configuration)
Recalibrate the module. The function forces the recalibration process of the module to get optimized parameters.
- `int32_t ft_module_change_mode` (struct `ft_module` *module, const enum `ft_module_mode` mode, const struct `ft_electrode` *electrode)
Changes the module mode of the operation.
- `int32_t ft_module_load_configuration` (struct `ft_module` *module, const enum `ft_module_mode` mode, const void *config)
Load module configuration for the selected mode. The function loads the new configuration to the module for the selected mode of operation.
- `int32_t ft_module_save_configuration` (struct `ft_module` *module, const enum `ft_module_mode` mode, void *config)
Saves the module configuration for the selected mode. The function saves the configuration from the module for the selected mode of operation into the user storage place.

7.6.2.4.2 Function Documentation

7.6.2.4.2.1 `int32_t ft_module_change_mode (struct ft_module * module, const enum ft_module_mode mode, const struct ft_electrode * electrode)`

Parameters

<i>module</i>	Pointer to the module.
---------------	------------------------

Modules

<i>mode</i>	New requested mode of the module.
<i>electrode</i>	Pointer to the electrode used in special modes (low-power & proximity); only one electrode is enabled in these modes.

Returns

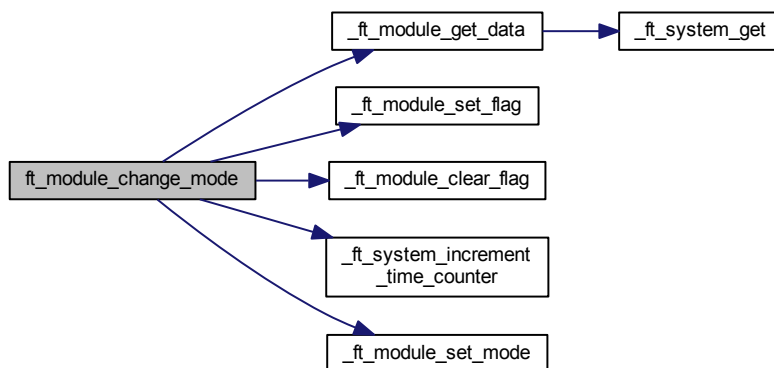
- FT_SUCCESS if the mode was properly changed
- FT_FAILURE if the mode cannot be changed This is an example of changing the mode of the module operation in the FT library:

```

if(ft_module_change_mode(&my_ft_module,
    FT_MODULE_MODE_PROXIMITY, &my_proximity_electrode) ==
    FT_FAILURE)
{
    printf("The change of mode for my_ft_module failed.");
}
// The FT successfully changed mode of my_ft_module

```

Here is the call graph for this function:



7.6.2.4.2.2 `int32_t ft_module_load_configuration (struct ft_module * module, const enum ft_module_mode mode, const void * config)`

Parameters

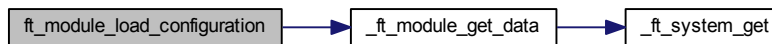
<i>module</i>	Pointer to the module.
<i>mode</i>	Mode of the module.
<i>config</i>	Pointer to the configuration data of the module, the type is dependent on the target module.

Returns

- FT_SUCCESS if the load operation was properly done
- FT_FAILURE if the load operation cannot be finished This is an example of loading the configuration data of the module in the FT library:

```
// I want to load new configuration for the TSI module and the proximity mode
if(ft_module_load_configuration(&my_ft_module,
    FT_MODULE_MODE_PROXIMITY, &my_module_tsi_proximity_configuration) ==
    FT_FAILURE)
{
    printf("Loading of new configuration for the my_ft_module failed.");
}
// The FT successfully loaded the new configuration of the my_ft_module.
```

Here is the call graph for this function:


7.6.2.4.2.3 uint32_t ft_module_recalibrate (const struct ft_module * *module*, void * *configuration*)
Parameters

<i>module</i>	Pointer to the module to be recalibrated.
<i>configuration</i>	Pointer to the module configuration that must be used as a startup configuration for the recalibration.

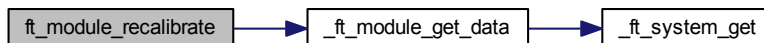
Returns

The lowest signal measured within the module. // todo is this a good return value? This is an example of recalibrating the module settings of the FT library:

```
// to do
if(ft_module_recalibrate(&my_ft_module, &my_ft_module) ==
    FT_FAILURE)
{
    printf("The change of mode for my_ft_module failed.");
}
// The FT successfully change mode of my_ft_module
```

Modules

Here is the call graph for this function:



7.6.2.4.2.4 int32_t ft_module_save_configuration (struct ft_module * *module*, const enum ft_module_mode *mode*, void * *config*)

Parameters

<i>module</i>	Pointer to the module.
<i>mode</i>	Mode of the module.
<i>config</i>	Pointer to the configuration data variable of the module, where the current configuration should be stored. The type is dependent on the target module.

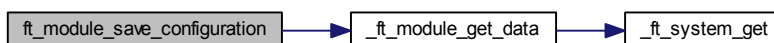
Returns

- FT_SUCCESS if the save operation was properly done
- FT_FAILURE if the save operation cannot be finished This is an example of saving the configuration data of the module in the FT library:

```

// I want to save the configuration of the TSI module and the proximity mode into my variable
// tsi_config_t my_module_tsi_proximity_configuration;
if(ft_module_save_configuration(&my_ft_module,
    FT_MODULE_MODE_PROXIMITY, &my_module_tsi_proximity_configuration) ==
    FT_FAILURE)
{
    printf("Saving of the current configuration for the my_ft_module failed.");
}
// The FT successfully saved the current configuration of the my_ft_module
  
```

Here is the call graph for this function:



7.7 System

7.7.1 Overview

The System structure represents the Freescale Touch Library in the user application; it is represented by the `ft_system` structure, which contains further references to all other application objects like: [Electrodes](#), [Key Detectors](#), [Modules](#), and [Controls](#).

The `ft_system` structure is allocated and initialized by the user, in order to define library configuration, including low-level electrode hardware channels and high-level control parameters. Just like all other structure types, it is up to the user, whether an instance of this structure is allocated statically in compile-time or dynamically. The examples provided with the Freescale Touch library show the static allocation and initialization of `ft_system` along with all other related data structures. Collaboration diagram for System:



Modules

- [API Functions](#)

Data Structures

- struct [ft_system](#)

Typedefs

- typedef void(* [ft_system_callback](#))(uint32_t event)
- typedef void(* [ft_error_callback](#))(char *file_name, uint32_t line)

Enumerations

- enum [ft_system_event](#) {
[FT_SYSTEM_EVENT_OVERRUN](#),
[FT_SYSTEM_EVENT_DATA_READY](#) }

System

7.7.2 Data Structure Documentation

7.7.2.1 struct ft_system

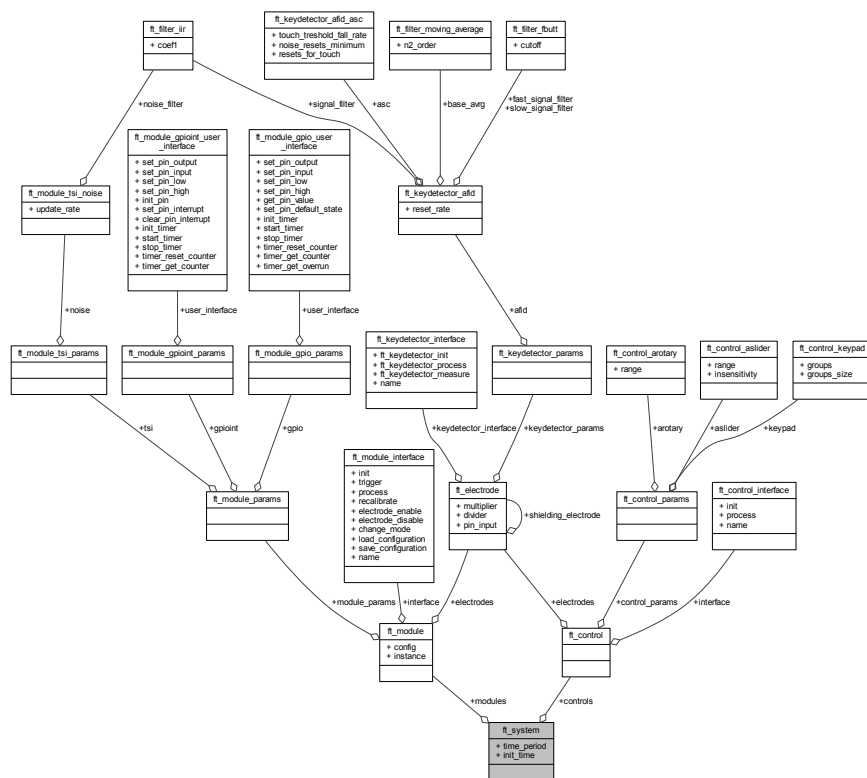
The main structure representing the Freescale Touch library; The structure contains pointer lists referring to all other objects used in the application, such as [Electrodes](#), [Key Detectors](#), [Modules](#), and [Controls](#).

The [ft_system](#) structure and all referred structures are allocated and initialized by the user code, in order to define the library configuration. This configuration affects all library layers from the low-level electrode parameters (for example hardware pins and channels) up to the high-level control parameters (for example slider range or keypad multiplexing).

Just like with all other structure types, it is up to the user, whether the instance of this structure is allocated statically in the compile-time, or dynamically. Examples provided with the Freescale Touch library show the static allocation and initialization of the [ft_system](#), along with all other related data structures.

This structure can be allocated in ROM.

Collaboration diagram for [ft_system](#):



Data Fields

struct ft_control *const *	controls	A pointer to the list of controls. Can't be NULL.
uint16_t	init_time	Initialization time for the system.
struct ft_module *const *	modules	A pointer to the list of modules. Can't be NULL.
uint16_t	time_period	Defined time period (triggering period). Can't be 0.

7.7.3 Typedef Documentation

7.7.3.1 typedef void(* ft_error_callback)(char *file_name, uint32_t line)

Error callback function pointer type.

Parameters

<i>file</i>	The name of the file where the error occurs.
<i>line</i>	The line index in the file where the error occurs.

Returns

None.

7.7.3.2 typedef void(* ft_system_callback)(uint32_t event)

System event callback function pointer type.

Parameters

<i>event</i>	Event type ft_system_event that caused the callback function call.
--------------	--

System

Returns

None.

7.7.4 Enumeration Type Documentation

7.7.4.1 enum ft_system_event

System callbacks events.

Enumerator

FT_SYSTEM_EVENT_OVERRUN Data has been overrun.

FT_SYSTEM_EVENT_DATA_READY New data are available.

7.7.5 API Functions

7.7.5.1 Overview

General Function definition of the system. Collaboration diagram for API Functions:



Functions

- void [ft_system_register_callback](#) ([ft_system_callback](#) callback)
Register the system callback function.
- void [ft_error_register_callback](#) ([ft_error_callback](#) callback)
Register the system error callback function.
- [uint32_t ft_system_get_time_counter](#) (void)
Returns the system time counter.
- [uint32_t ft_mem_get_free_size](#) (void)
Returns the free memory size in the FT memory pool.
- [int32_t ft_init](#) (const struct [ft_system](#) *system, [uint8_t](#) *pool, const [uint32_t](#) size)
Freescale Touch Library initialization.
- [int32_t ft_task](#) (void)
Freescale Touch Main processing entry point.
- [int32_t ft_trigger](#) (void)
Main Trigger function to acquire the touch-sensing data.

7.7.5.2 Function Documentation

7.7.5.2.1 void [ft_error_register_callback](#) ([ft_error_callback](#) *callback*)

Parameters

<i>callback</i>	Pointer to the callback function, which will receive the error event notifications.
-----------------	---

Returns

none After this callback finishes, the driver falls to a never ending loop. This is an example of installing and using the parameters of the FT library error handler:

System

```

* static void my_ft_error_callback(char *file_name, uint32_t line);
*
* // For library debugging only, install the error handler
* ft_error_register_callback(my_ft_error_callback)
*
* // The FT error-handling routine
* static void my_ft_error_callback(char *file_name, uint32_t line)
* {
*     printf("\nError occured in the FT library. File: %s, Line: %d.\n", file_name, line);
* }
*
*

```

7.7.5.2.2 int32_t ft_init (const struct ft_system * system, uint8_t * pool, const uint32_t size)

Parameters

<i>system</i>	Pointer to the FT system parameters structure.
<i>pool</i>	Pointer to the memory pool what will be used for internal FT data.
<i>size</i>	Size of the memory pool handled by the parameter pool (needed size depends on the number of components used in the FT - electrodes, modules, controls, and so on).

Returns

- FT_SUCCESS if library was properly initialized,
- FT_FAILURE if initialization failed (one of the reasons is not enough memory in the memory pool).

This function validates the Freescale Touch configuration passed within the [ft_system](#) structure. After this call, the system structure becomes the main data provider for the application. There are also created and filled-up internal volatile data structures used by the driver. It is the user's responsibility to prepare the configuration of all electrodes, modules, and controls in the system structure before calling this function. The application should not execute any other FT library calls if this function returns FT_FAILURE. This is an example of the FT library initialization:

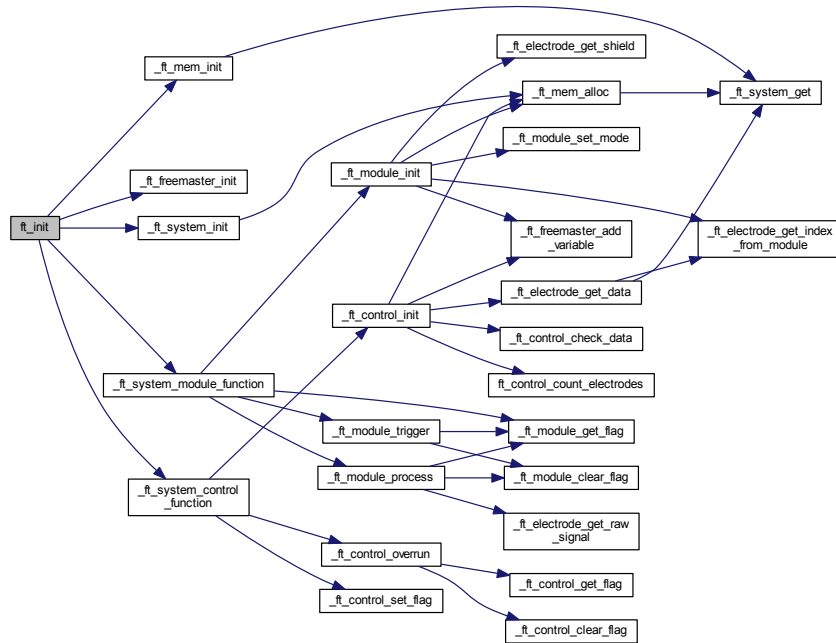
```

uint8_t ft_memory_pool[512];

if(ft_init(&my_ft_system_params, ft_memory_pool, sizeof(ft_memory_pool)) ==
    FT_FAILURE)
{
    printf("Initialization of FT failed. There can be a problem with the memory size
    or invalid parameters in component parameter structures.");
}
// The FT is successfully initialized

```


Here is the call graph for this function:



7.7.5.2.3 uint32_t ft_mem_get_free_size (void)

Returns

size of unused memory in the FT memory pool

This can be used in debugging of the driver to specify the exact size of the Freescale Touch memory pool needed. This is an example of initializing the FT library and checking the final size:

```

* uint8_t ft_memory_pool[512];
*
* if(ft_init(&my_ft_system_params, ft_memory_pool, sizeof(ft_memory_pool)) ==
*   FT_FAILURE)
* {
*   printf("Initialization of the FT failed. There may be a problem with the memory size,
*   or invalid parameters in the component parameter structures.");
* }
* // The FT is successfully initialized
*
* printf("The unused memory size is: %d Bytes. The memory pool can be reduced
* by this size.", ft_mem_get_free_size());
*
*

```

System

Here is the call graph for this function:



7.7.5.2.4 uint32_t ft_system_get_time_counter (void)

Returns

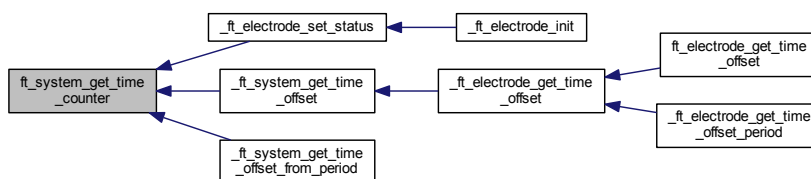
Time counter value. This is an example of getting the current time of the FT library:

```

* // Printing the current Freescale Touch library time
* printf("The current FT library time is: %d ms since start.\n",
*       ft_system_get_time_counter());
*
*

```

Here is the caller graph for this function:



7.7.5.2.5 void ft_system_register_callback (ft_system_callback callback)

Parameters

<i>callback</i>	Pointer to the callback function, which will receive the system event notifications.
-----------------	--

Returns

none This is an example of installing and using the parameters of the FT library system events handler:

```

* static void my_ft_system_callback(uint32_t event);
*
* // To catch the system events, install the system handler
* ft_system_register_callback(my_ft_system_callback)
*
* // The FT system events handling routine
* static void my_ft_system_callback(uint32_t event)
* {
*     if(event == FT_SYSTEM_EVENT_OVERRUN)
*     {
*         printf("\nThe measured data has been overrun. Call more frequently ft_task();");
*     }
*     else if(event == FT_SYSTEM_EVENT_DATA_READY)
*     {
*         printf("\nThere is new data in the FT library.");
*     }
* }
*
*
*

```

7.7.5.2.6 int32_t ft_task (void)

Returns

- FT_SUCCESS when data acquired during the last trigger are now processed
- FT_FAILURE when no new data are ready

This function should be called by the application as often as possible, in order to process the data acquired during the data trigger. This function should be called at least once per trigger time.

Internally, this function passes the FT_SYSTEM_MODULE_PROCESS and FT_SYSTEM_CONTROL_PROCESS command calls to each object configured in [Modules](#) and [Controls](#). This is an example of running a task of the FT library:

```

uint8_t ft_memory_pool[512];

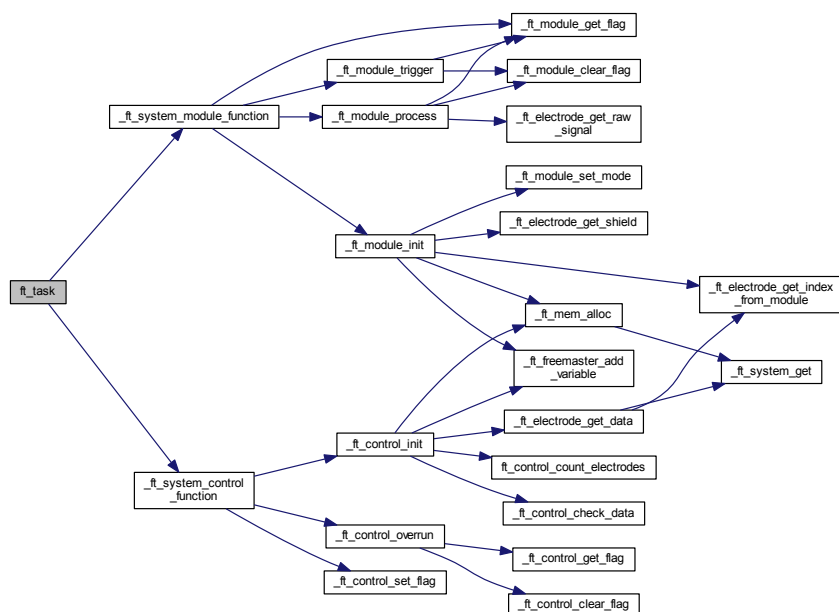
if(ft_init(&my_ft_system_params, ft_memory_pool, sizeof(ft_memory_pool)) ==
    FT_FAILURE)
{
    printf("Initialization of FT failed. There can be problem with memory size
    or invalid parameters in component parameter structures.");
}
// The FT is successfully initialized

// Main never-ending loop of the application
while(1)
{
    if(ft_task() == FT_SUCCESS)
    {
        // New data has been processed
    }
}

```

System

Here is the call graph for this function:



7.7.5.2.7 int32_t ft_trigger (void)

Returns

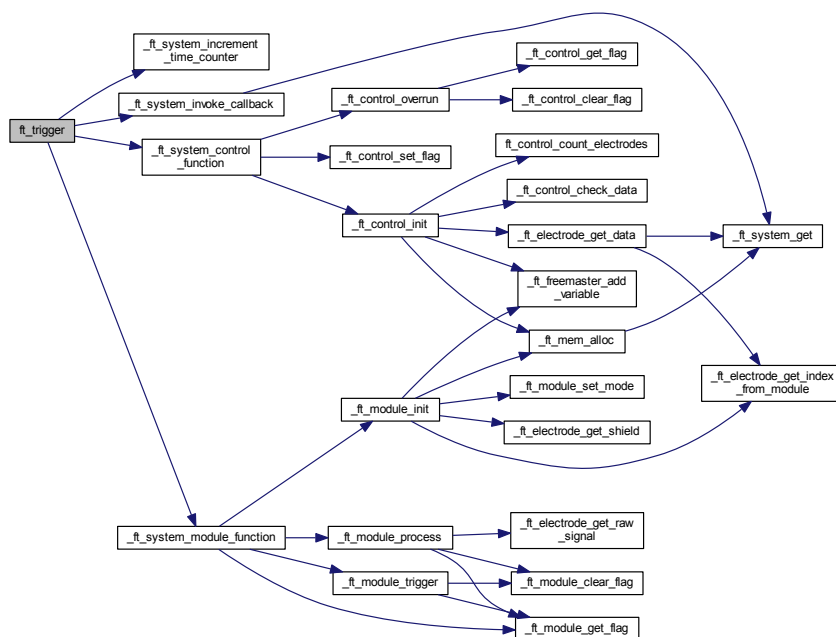
- FT_SUCCESS when the trigger was performed without any errors or warnings.
- FT_FAILURE when a problem is detected, such as module not ready, overrun (data loss) error, and so on. Regardless of the error, the trigger is always initiated.

This function should be called by the application periodically in a timer interrupt, or in a task to trigger new data measurement. Depending on the [Modules](#) implementation, this function may take the data immediately, or may only start the hardware sampling with interrupt enabled. This is an example of the FT library triggering:

```

//For example, there is a callback routine from any priodical source (for example 5 ms)
static void Timer_5msCallback(void)
{
    if(ft_trigger() != FT_SUCCESS)
    {
        // Trigger error
    }
}
  
```

Here is the call graph for this function:



General Types

7.8 General Types

7.8.1 Overview

The standard types used in the whole Freescale Touch software library. The code is built on the standard library types, such as `uint32_t`, `int8_t`, and so on, loaded from "stdint.h", and it defines just few additional types needed to run the FT. Collaboration diagram for General Types:



Macros

- #define `NULL`
Standard NULL pointer. There is a definition in the FT, in case that NULL is not defined in the project previously.
- #define `FT_FLAGS_SYSTEM_SHIFT(x)`
- #define `FT_FLAGS_SPECIFIC_SHIFT(x)`
- #define `FT_FREEMASTER_SUPPORT`
FT_FREEMASTER_SUPPORT enables the support of FreeMASTER for the Freescale Touch project. When it is enabled, the FT starts using / including the freemaster.h file. FT_DEBUG is enabled by default.
- #define `FT_DEBUG`
FT_DEBUG enables the debugging code that caused the assert states in the FT. For the release compilation, this option should be disabled. FT_DEBUG is enabled by default, which enables the FT ASSERTS.
- #define `FT_ASSERT(expr)`

Enumerations

- enum `ft_result` {
`FT_SUCCESS,`
`FT_FAILURE,`
`FT_OUT_OF_MEMORY,`
`FT_SCAN_IN_PROGRESS,`
`FT_NOT_SUPPORTED,`
`FT_INVALID_RESULT` }
The Freescale Touch result of most operations / API. The standard API function returns the result of the finished operation if needed, and it can have the following values.

7.8.2 Macro Definition Documentation

7.8.2.1 **#define FT_ASSERT(*expr*)**

7.8.2.2 **#define FT_DEBUG**

7.8.2.3 **#define FT_FLAGS_SPECIFIC_SHIFT(*x*)**

7.8.2.4 **#define FT_FLAGS_SYSTEM_SHIFT(*x*)**

Generic flags for FT processing.

7.8.2.5 **#define FT_FREEMASTER_SUPPORT**

7.8.2.6 **#define NULL**

7.8.3 Enumeration Type Documentation

7.8.3.1 **enum ft_result**

Enumerator

FT_SUCCESS Successful result - Everything is alright.

FT_FAILURE Something is wrong, unspecified error.

FT_OUT_OF_MEMORY The FT does not have enough memory.

FT_SCAN_IN_PROGRESS The scan is currently in progress.

FT_NOT_SUPPORTED The feature is not supported.

FT_INVALID_RESULT The function ends with an invalid result.

General Types

7.8.4 Analog Rotary Control

7.8.4.1 Overview

Analog Rotary enables the detection of jog-dial-like finger movement using three or more electrodes; it is represented by the `ft_control` structure.

An Analog Rotary Control uses three or more specially-shaped electrodes to enable the calculation of finger position within a circular area. The position algorithm uses a ratio of sibling electrode signals to estimate the finger position with required precision.

The Analog Rotary works similarly to the "standard" Rotary, but requires less number of electrodes, while achieving a higher resolution of the calculated position. For example, a four-electrode analog rotary can provide finger position detection in the range of 0-64. The shape of the electrodes needs to be designed specifically to achieve stable signal with a linear dependence on the finger movement.

The Analog Rotary Control provides Position, Direction, and Displacement values. It is able to generate event callbacks when finger Movement, Initial-touch, or Release is detected.

The image below shows a typical four-electrode Analog Rotary electrode placement.

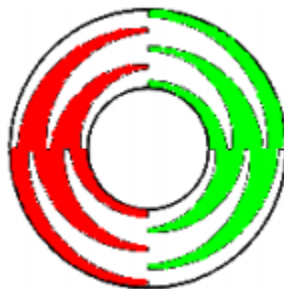


Figure 7.8.1: Analog Rotary Electrodes

Collaboration diagram for Analog Rotary Control:



Data Structures

- struct `ft_control_arotary_temp_data`
- struct `ft_control_arotary_data`

Macros

- #define `FT_AROTARY_INVALID_POSITION_VALUE`

Enumerations

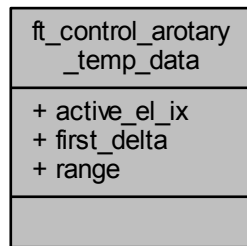
- enum `ft_control_arotary_flags` {
`FT_AROTARY_INVALID_POSITION_FLAG,`
`FT_AROTARY_DIRECTION_FLAG,`
`FT_AROTARY_MOVEMENT_FLAG,`
`FT_AROTARY_TOUCH_FLAG` }

7.8.4.2 Data Structure Documentation

7.8.4.2.1 struct `ft_control_arotary_temp_data`

Analog Rotary help structure to handle temporary values

Collaboration diagram for `ft_control_arotary_temp_data`:



Data Fields

<code>uint32_t</code>	<code>active_el_ix</code>	Index of electrode of first active electrode.
<code>uint32_t</code>	<code>first_delta</code>	Value of first delta (signal - baseline).

General Types

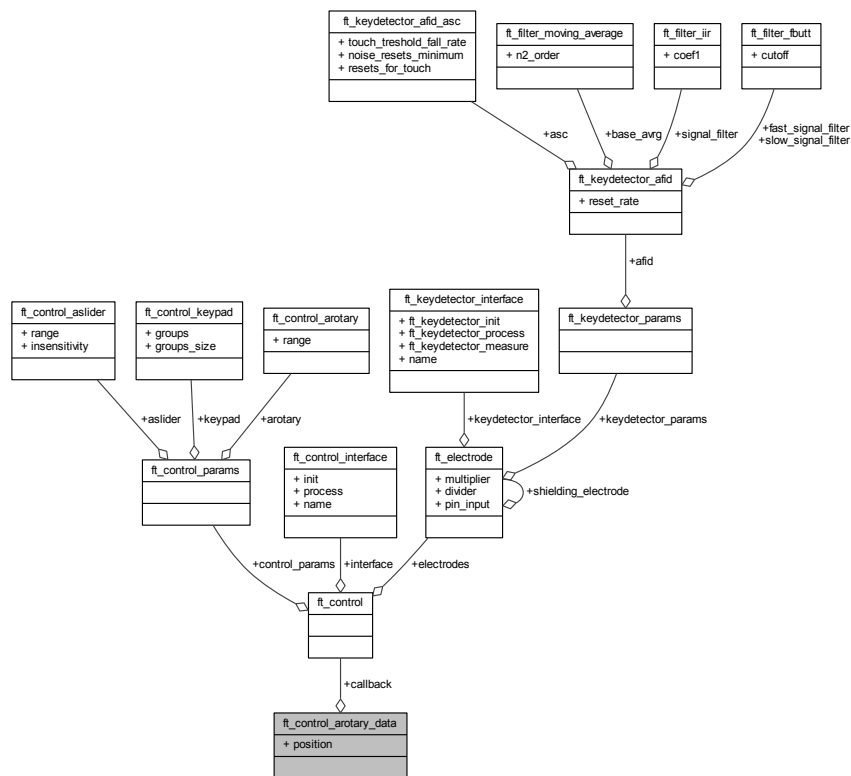
uint32_t	range	Value of first delta (signal - baseline).
----------	-------	---

7.8.4.2.2 struct ft_control_arotary_data

Analog Rotary RAM structure used to store volatile parameters of the control.

You need to allocate this structure and put a pointer into the [ft_control](#) structure when it is being registered in the system.

Collaboration diagram for `ft_control_arotary_data`:



Data Fields

ft_control_arotary_callback	callback	Analog Rotary callback handler.
uint8_t	position	Position.

7.8.4.3 Macro Definition Documentation

7.8.4.3.1 #define FT_AROTARY_INVALID_POSITION_VALUE

Value that is used to mark an invalid position of the Analog Rotary.

7.8.4.4 Enumeration Type Documentation

7.8.4.4.1 enum ft_control_arotary_flags

Analog Rotary flags.

Enumerator

FT_AROTARY_INVALID_POSITION_FLAG Analog Rotary invalid position flag.

FT_AROTARY_DIRECTION_FLAG Analog Rotary direction flag.

FT_AROTARY_MOVEMENT_FLAG Analog Rotary movement flag.

FT_AROTARY_TOUCH_FLAG Analog Rotary touch flag.

General Types

7.8.5 Analog Slider Control

7.8.5.1 Overview

Analog Slider enables the detection of linear finger movement using two or more electrodes; it is represented by the `ft_control_aslider` structure.

An Analog Slider Control uses two or more specially-shaped electrodes to enable the calculation of finger position within a linear area. The position algorithm uses a ratio of electrode signals to estimate the finger position with required precision.

The Analog Slider works similarly to the "standard" Slider, but requires less electrodes, while achieving a higher resolution of the calculated position. For example, a two-electrode analog slider can provide finger position detection in the range of 0-127. The shape of the electrodes needs to be designed specifically to achieve stable signal with a linear dependance on finger movement.

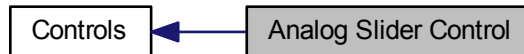
The Analog Slider Control provides Position, Direction, and Displacement values. It is able to generate event callbacks when finger Movement, Initial-touch, or Release is detected.

The image below shows a typical two-electrode Analog Slider electrode placement.



Figure 7.8.2: Analog Slider Electrodes

Collaboration diagram for Analog Slider Control:



Data Structures

- struct `ft_control_aslider_data`
- struct `ft_control_aslider_temp_data`

Macros

- `#define FT_ASILIDER_INVALID_POSITION_VALUE`

Enumerations

- enum `ft_control_aslider_flags` {
`FT_ASLIDER_INVALID_POSITION_FLAG`,
`FT_ASLIDER_DIRECTION_FLAG`,
`FT_ASLIDER_MOVEMENT_FLAG`,
`FT_ASLIDER_TOUCH_FLAG` }

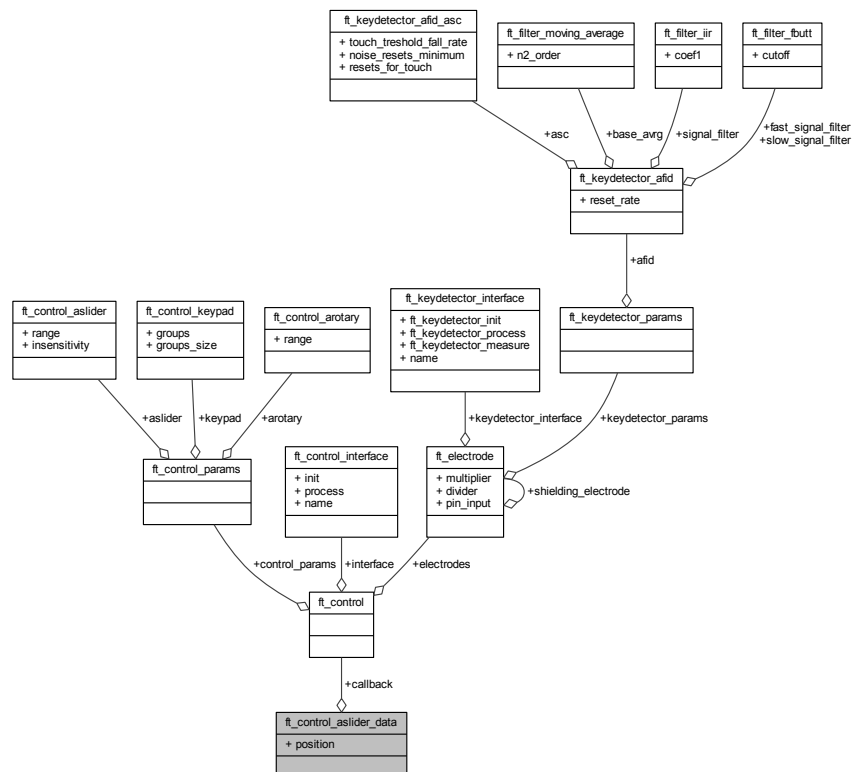
7.8.5.2 Data Structure Documentation

7.8.5.2.1 struct `ft_control_aslider_data`

Analog Slider RAM structure used to store volatile parameters of the control.

You need to allocate this structure and put a pointer into the `ft_control_aslider` structure when it is being registered in the system.

Collaboration diagram for `ft_control_aslider_data`:



General Types

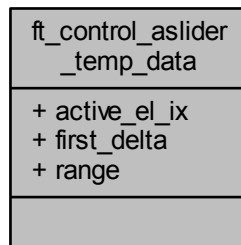
Data Fields

<code>ft_control_aslider_callback</code>	callback	Analog Slider callback handler.
<code>uint8_t</code>	position	Position.

7.8.5.2 struct ft_control_aslider_temp_data

Analog Slider help structure to handle temporary values

Collaboration diagram for `ft_control_aslider_temp_data`:



Data Fields

<code>uint32_t</code>	<code>active_el_ix</code>	Index of electrode of first active electrode.
<code>uint32_t</code>	<code>first_delta</code>	Value of first delta (signal - baseline).
<code>uint32_t</code>	<code>range</code>	Value of first delta (signal - baseline).

7.8.5.3 Macro Definition Documentation

7.8.5.3.1 #define FT_ASIDER_INVALID_POSITION_VALUE

Value that is used to mark an invalid position of analog slider.

7.8.5.4 Enumeration Type Documentation

7.8.5.4.1 enum ft_control_aslider_flags

Analog Slider flags.

Enumerator

FT_ASLIDER_INVALID_POSITION_FLAG Analog Slider invalid position flag.

FT_ASLIDER_DIRECTION_FLAG Analog Slider direction flag.

FT_ASLIDER_MOVEMENT_FLAG Analog Slider movement flag.

FT_ASLIDER_TOUCH_FLAG Analog Slider touch flag.

General Types

7.8.6 Keypad Control

7.8.6.1 Overview

Keypad implements the keyboard-like functionality on top of an array of electrodes; it is represented by the `ft_control_keypad` structure.

An application may use the Electrode API to determine the touch or release states of individual electrodes. The Keypad simplifies this task and it extends this simple scenario by introducing a concept of a "key". The "key" is represented by one or more physical electrodes, so the Keypad control enables one electrode to be shared by several keys. Each key is defined by a set of electrodes that all need to be touched in order to report the "key press" event.

The Keypad Control provides Key status values and is able to generate the Key Touch, Auto-repeat, and Release events.

The images below show simple and grouped keypad electrode layouts.

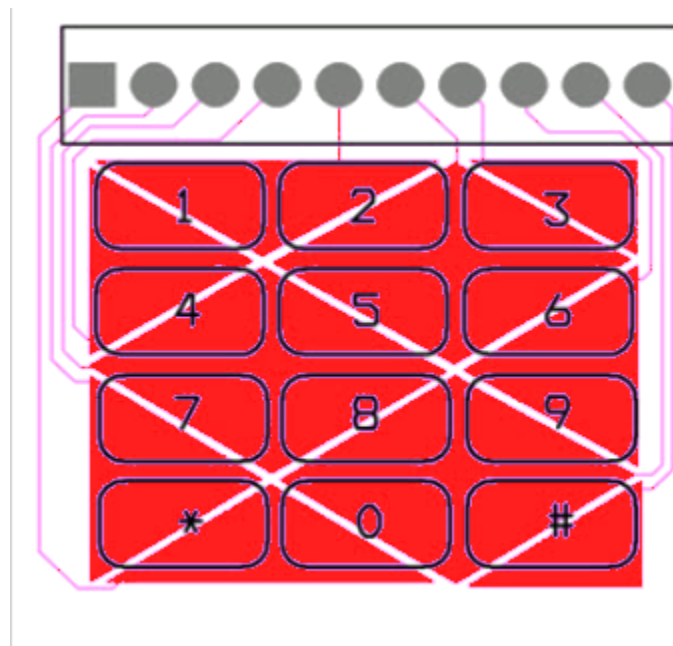


Figure 7.8.3: Keypad Electrodes

Collaboration diagram for Keypad Control:



Data Structures

- struct [ft_control_keypad_data](#)

Enumerations

- enum [ft_control_keypad_flags](#) { [FT_KEYPAD_ONLY_ONE_KEY_FLAG](#) }

7.8.6.2 Data Structure Documentation

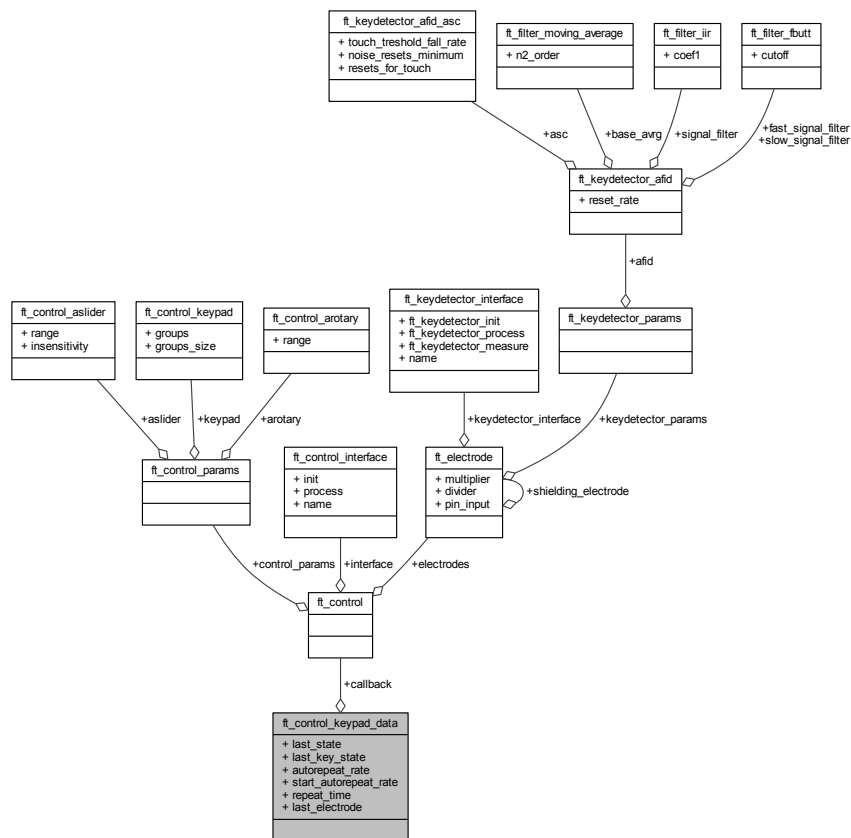
7.8.6.2.1 struct [ft_control_keypad_data](#)

The Keypad RAM structure used to store volatile parameters of the control.

You must allocate this structure and put a pointer into the [ft_control_keypad](#) structure when it is being registered in the system.

General Types

Collaboration diagram for ft_control_keypad_data:



Data Fields

uint16_t	autorepeat_rate	Autorepeat rate.
ft_control_keypad_callback	callback	Keypad callback handler.
int32_t	last_electrode	Last touched electrode index.
uint32_t	last_key_state	Last state of keypad keys.
uint32_t	last_state	Last state of keypad electrodes.

uint32_t	repeat_time	Time of next autorepeat event.
uint32_t	start_ autorepeat_rate	Start Autorepeat rate.

7.8.6.3 Enumeration Type Documentation

7.8.6.3.1 enum ft_control_keypad_flags

Keypad flags.

Enumerator

FT_KEYPAD_ONLY_ONE_KEY_FLAG Keypad only one key is valid flag.

General Types

7.8.7 Rotary Control

7.8.7.1 Overview

Rotary enables the detection of jog-dial-like finger movement using discrete electrodes; it is represented by the `ft_control_rotary_control` structure.

The Rotary Control uses a set of discrete electrodes to enable the calculation of finger position within a circular area. The position algorithm localizes the touched electrode and its sibling electrodes to estimate the finger position. The Rotary consisting of N electrodes enables the rotary position to be calculated in $2N$ steps.

The Rotary Control provides Position, Direction, and Displacement values. It is able to generate event callbacks when finger Movement, Initial-touch, or Release is detected.

The image below shows a typical Rotary electrode placement.

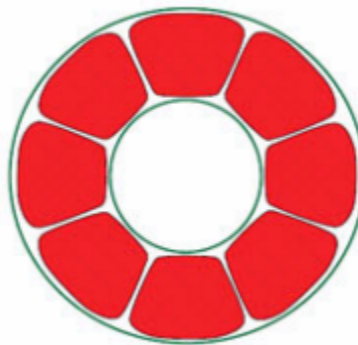


Figure 7.8.4: Rotary Electrodes

Collaboration diagram for Rotary Control:



Data Structures

- struct [ft_control_rotary_data](#)

Enumerations

- enum ft_control_rotary_flags {
 - FT_ROTARY_INVALID_POSITION_FLAG,
 - FT_ROTARY_DIRECTION_FLAG,
 - FT_ROTARY_MOVEMENT_FLAG,
 - FT_ROTARY_TOUCH_FLAG }

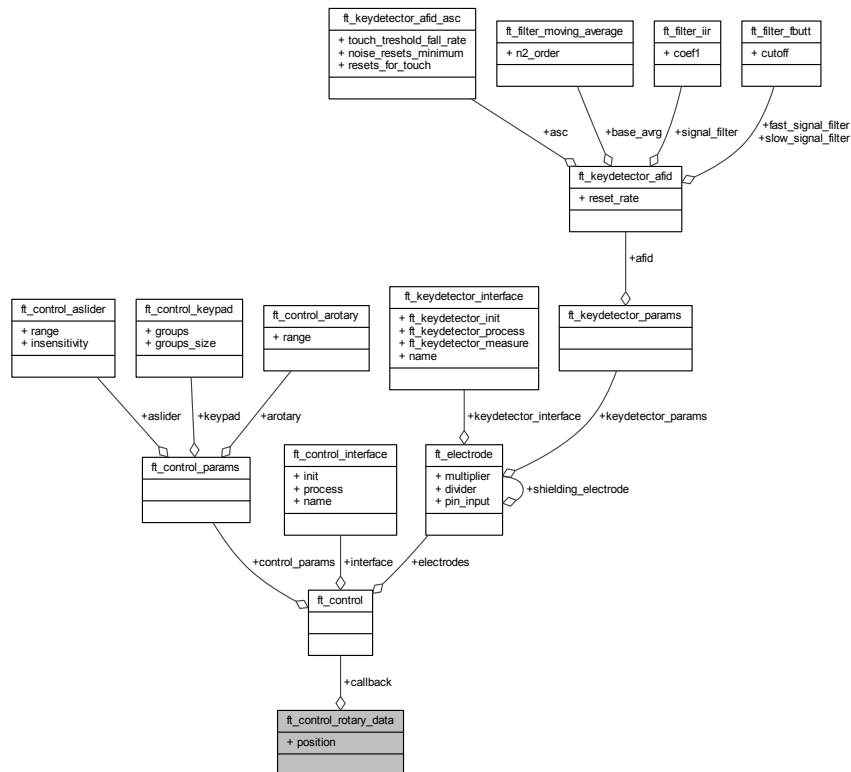
7.8.7.2 Data Structure Documentation

7.8.7.2.1 struct ft_control_rotary_data

Rotary RAM structure used to store volatile parameters of the control.

You must allocate this structure and put a pointer into the ft_control_rotary_control structure when it is being registered in the system.

Collaboration diagram for ft_control_rotary_data:



General Types

Data Fields

ft_control_rotary_callback	callback	Callback handler.
uint8_t	position	Position.

7.8.7.3 Enumeration Type Documentation

7.8.7.3.1 enum ft_control_rotary_flags

Rotary flags.

Enumerator

FT_ROTARY_INVALID_POSITION_FLAG Rotary invalid position flag.

FT_ROTARY_DIRECTION_FLAG Rotary direction flag.

FT_ROTARY_MOVEMENT_FLAG Rotary movement flag.

FT_ROTARY_TOUCH_FLAG Rotary touch flag.

7.8.8 Slider Control

7.8.8.1 Overview

Slider control enables the detection of a linear finger movement using discrete electrodes; it is represented by the [ft_control](#) structure.

The Slider Control uses a set of discrete electrodes to enable the calculation of finger position within a linear area. The position algorithm localizes the touched electrode and its sibling electrodes to estimate the finger position. The Slider consisting of N electrodes enables the position to be calculated in 2N-1 steps.

The Slider Control provides Position, Direction, and Displacement values. It is able to generate event callbacks when finger Movement, Initial-touch, or Release is detected.

The image below shows a typical Slider electrode placement.



Figure 7.8.5: Slider Electrodes

Collaboration diagram for Slider Control:



Data Structures

- struct [ft_control_slider_data](#)

General Types

Enumerations

- enum `ft_control_slider_flags` {
`FT_SLIDER_INVALID_POSITION_FLAG`,
`FT_SLIDER_DIRECTION_FLAG`,
`FT_SLIDER_MOVEMENT_FLAG`,
`FT_SLIDER_TOUCH_FLAG` }

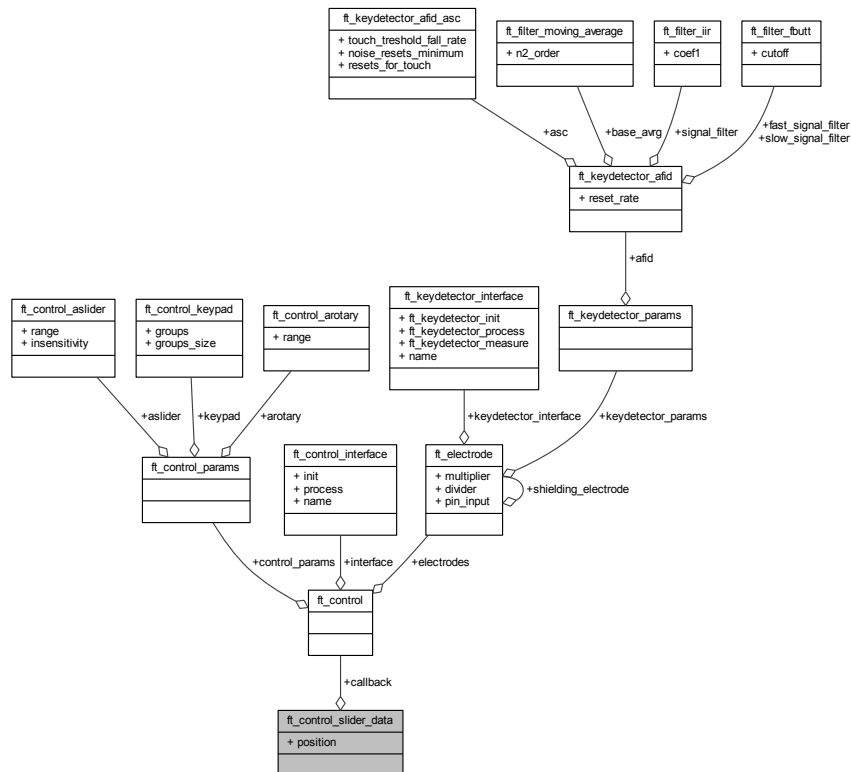
7.8.8.2 Data Structure Documentation

7.8.8.2.1 struct `ft_control_slider_data`

Slider RAM structure used to store volatile parameters of the control.

You must allocate this structure and put a pointer into the `ft_control_slider_control` structure when it is being registered in the system.

Collaboration diagram for `ft_control_slider_data`:



Data Fields

ft_control_slider_callback	callback	Slider Callback handler.
uint8_t	position	Position.

7.8.8.3 Enumeration Type Documentation

7.8.8.3.1 enum ft_control_slider_flags

Slider flags.

Enumerator

FT_SLIDER_INVALID_POSITION_FLAG Slider invalid position flag.

FT_SLIDER_DIRECTION_FLAG Slider direction flag.

FT_SLIDER_MOVEMENT_FLAG Slider movement flag.

FT_SLIDER_TOUCH_FLAG Slider touch flag.

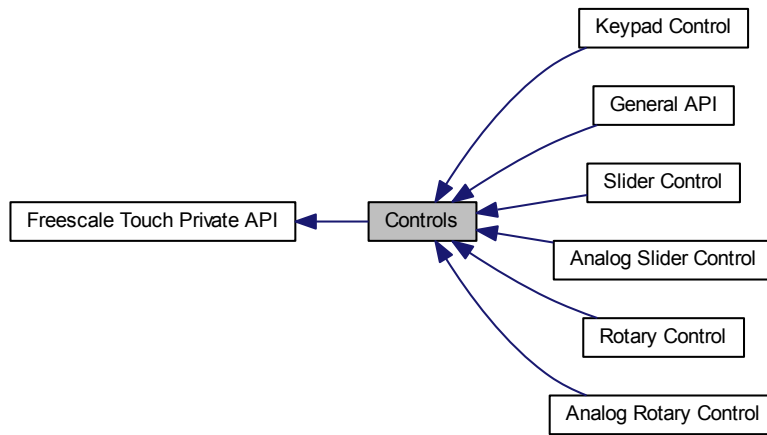
Controls

7.9 Controls

7.9.1 Overview

Controls represent the highest level of abstraction in the finger touch evaluation; the generic control object used as a base for other "derived" control types is represented by the `ft_control` structure.

Based on the signal and status information coming from the Electrode layer, the controls calculate finger actions like movement, keyboard touch, hold and so on. Collaboration diagram for Controls:



Modules

- [Analog Rotary Control](#)
- [Analog Slider Control](#)
- [Keypad Control](#)
- [Rotary Control](#)
- [Slider Control](#)
- [General API](#)

7.9.2 General API

7.9.2.1 Overview

General Function definition of controls. Collaboration diagram for General API:



Modules

- [API Functions](#)

Data Structures

- union [ft_control_special_data](#)
- struct [ft_control_data](#)
- struct [ft_control_interface](#)

Enumerations

- enum [ft_control_flags](#) {
 [FT_CONTROL_NEW_DATA_FLAG](#),
 [FT_CONTROL_EN_FLAG](#) }

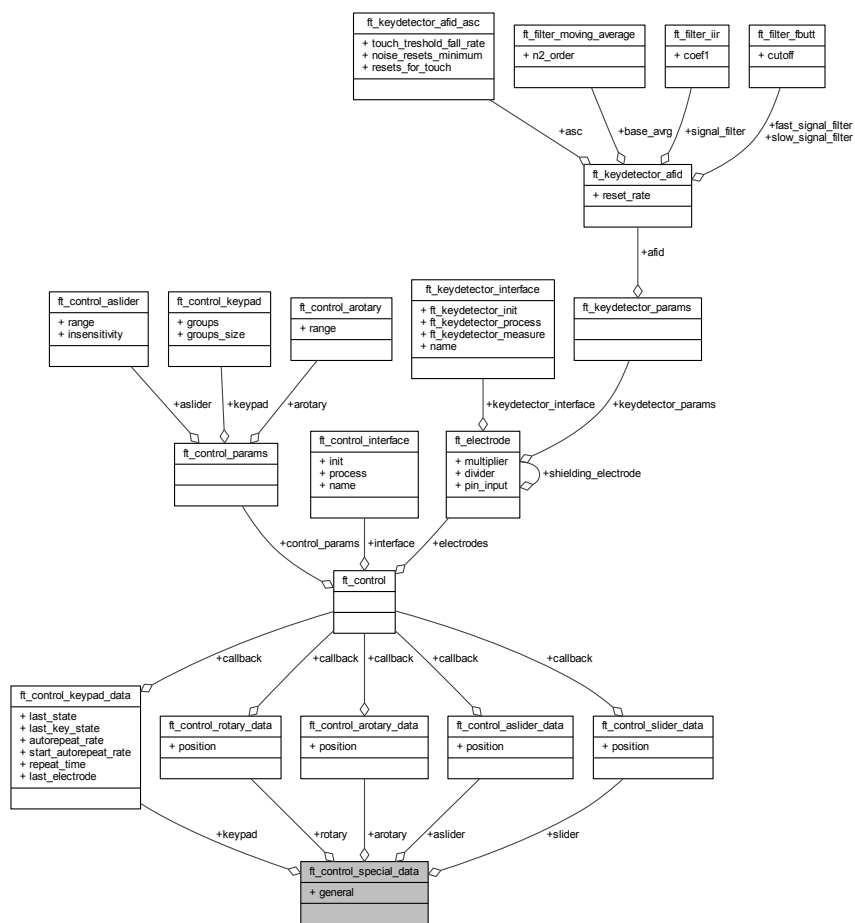
7.9.2.2 Data Structure Documentation

7.9.2.2.1 union ft_control_special_data

The pointer to the special data of the control. Each control type has its own type of the data structure, and the pointers to these special data structures are handled by this union, to keep the clearance of the types.

Controls

Collaboration diagram for `ft_control_special_data`:



Data Fields

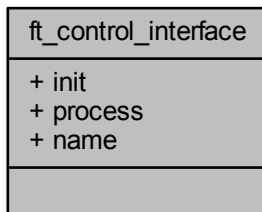
struct <code>ft_control_- arotary_data</code> *	<code>arotary</code>	Pointer to the Analog Rotary control special data.
---	----------------------	--

struct ft_control_- aslider_data *	aslider	Pointer to the Analog Slider control special data.
void *	general	Just one point of view on this union for a general sanity check.
struct ft_control_- keypad_data *	keypad	Pointer to the Keypad control special data.
struct ft_control_- rotary_data *	rotary	Pointer to the Rotary control special data.
struct ft_control_- slider_data *	slider	Pointer to the Slider control special data.

7.9.2.2.2 struct ft_control_data

The Control RAM structure used to store volatile parameters, flags, and other data to enable a generic behavior of the Control. You must allocate this structure and put a pointer into the [ft_control](#) structure when the control is being registered in the system.

Collaboration diagram for ft_control_interface:



Data Fields

- int32_t(* [init](#))(struct [ft_control_data](#) *control)
- int32_t(* [process](#))(struct [ft_control_data](#) *control)
- const char * [name](#)

7.9.2.2.3.1 Field Documentation

7.9.2.2.3.1.1 int32_t(* ft_control_interface::init)(struct ft_control_data *control)

The adress of init function.

7.9.2.2.3.1.2 const char* ft_control_interface::name

The name of the variable of this type, used for FreeMASTER support purposes.

7.9.2.2.3.1.3 int32_t(* ft_control_interface::process)(struct ft_control_data *control)

The adress of process function.

7.9.2.3 Enumeration Type Documentation

7.9.2.3.1 enum ft_control_flags

Controls flags which can be set / cleared.

Enumerator

FT_CONTROL_NEW_DATA_FLAG Indication flag that the control has new data.

FT_CONTROL_EN_FLAG Control Enable flag.

Controls

7.9.2.4 API Functions

7.9.2.4.1 Overview

The functions in this category can be used to manipulate the Control objects. Collaboration diagram for API Functions:



Functions

- struct `ft_control_data` * `_ft_control_get_data` (const struct `ft_control` *control)
Get control data structure pointer.
- struct `ft_control_data` * `_ft_control_init` (const struct `ft_control` *control)
Initialize the control object.
- int32_t `_ft_control_check_data` (const struct `ft_control_data` *control)
Validate control data.
- uint32_t `_ft_control_get_electrodes_state` (struct `ft_control_data` *control)
Get the state of all control electrodes.
- void `_ft_control_set_flag_all_elec` (struct `ft_control_data` *control, uint32_t flags)
- void `_ft_control_clear_flag_all_elec` (struct `ft_control_data` *control, uint32_t flag)
- uint32_t `_ft_control_get_first_elec_touched` (uint32_t current_state)
- uint32_t `_ft_control_get_last_elec_touched` (uint32_t current_state)
- uint32_t `_ft_control_get_touch_count` (uint32_t current_state)
- int32_t `_ft_control_check_neighbours_electrodes` (struct `ft_control_data` *control, uint32_t first, uint32_t second, uint32_t overrun)
- int32_t `_ft_control_check_edge_electrodes` (struct `ft_control_data` *control, uint32_t electrode_ix)
- static void `_ft_control_set_flag` (struct `ft_control_data` *control, uint32_t flags)
- static void `_ft_control_clear_flag` (struct `ft_control_data` *control, uint32_t flags)
- static uint32_t `_ft_control_get_flag` (const struct `ft_control_data` *control, uint32_t flags)
- static int32_t `_ft_control_overrun` (struct `ft_control_data` *control)
- static struct `ft_electrode` * `_ft_control_get_electrode` (const struct `ft_control_data` *control, uint32_t index)

7.9.2.4.2 Function Documentation

7.9.2.4.2.1 int32_t `_ft_control_check_data` (const struct `ft_control_data` * *control*)

Parameters

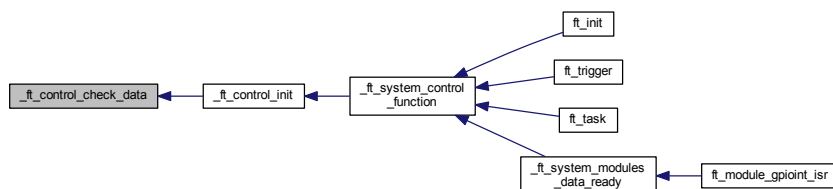
<i>control</i>	Pointer to the control data.
----------------	------------------------------

Returns

Status code.

Checking, whether the control data structure is sane; the interface, ram, and electrodes array should not be NULL.

Here is the caller graph for this function:

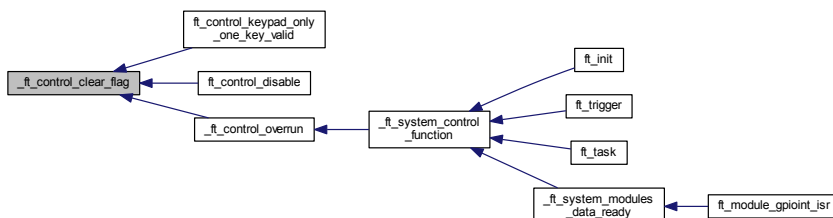


7.9.2.4.2.2 `int32_t ft_control_check_edge_electrodes (struct ft_control_data * control, uint32_t electrode_ix)`

7.9.2.4.2.3 `int32_t ft_control_check_neighbours_electrodes (struct ft_control_data * control, uint32_t first, uint32_t second, uint32_t overrun)`

7.9.2.4.2.4 `static void ft_control_clear_flag (struct ft_control_data * control, uint32_t flags) [inline], [static]`

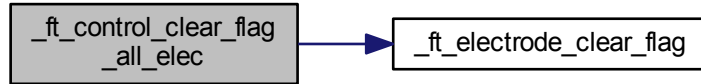
Here is the caller graph for this function:



Controls

7.9.2.4.2.5 `void _ft_control_clear_flag_all_elec (struct ft_control_data * control, uint32_t flag)`

Here is the call graph for this function:



7.9.2.4.2.6 `struct ft_control_data* _ft_control_get_data (const struct ft_control * control)`

Parameters

<i>control</i>	Pointer to the control user parameter structure.
----------------	--

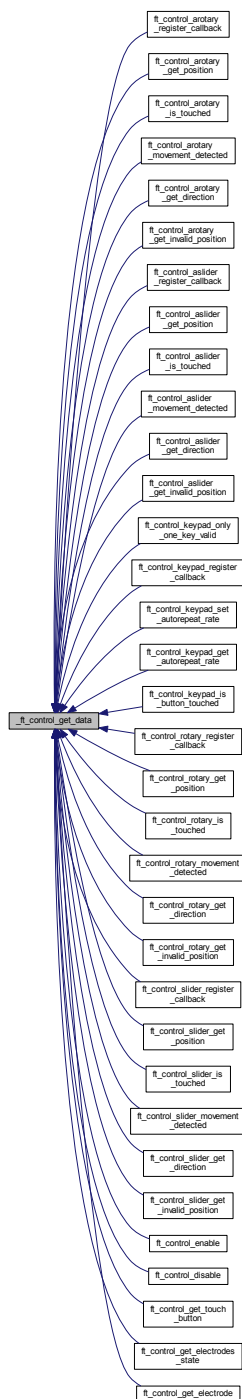
Returns

Pointer to the data control structure that is represented by the handled user parameter structure pointer.

Here is the call graph for this function:



Here is the caller graph for this function:



Controls

7.9.2.4.2.7 `static struct ft_electrode* _ft_control_get_electrode (const struct ft_control_data * control, uint32_t index) [static]`

7.9.2.4.2.8 `uint32_t _ft_control_get_electrodes_state (struct ft_control_data * control)`

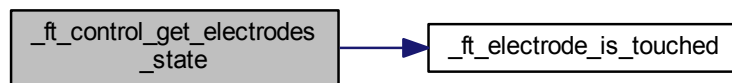
Parameters

<i>control</i>	Pointer to the control data.
----------------	------------------------------

Returns

This function returns a bit-mask value where each bit represents one control electrode. Logic 1 in the returned value represents a touched electrode.

Here is the call graph for this function:



Here is the caller graph for this function:

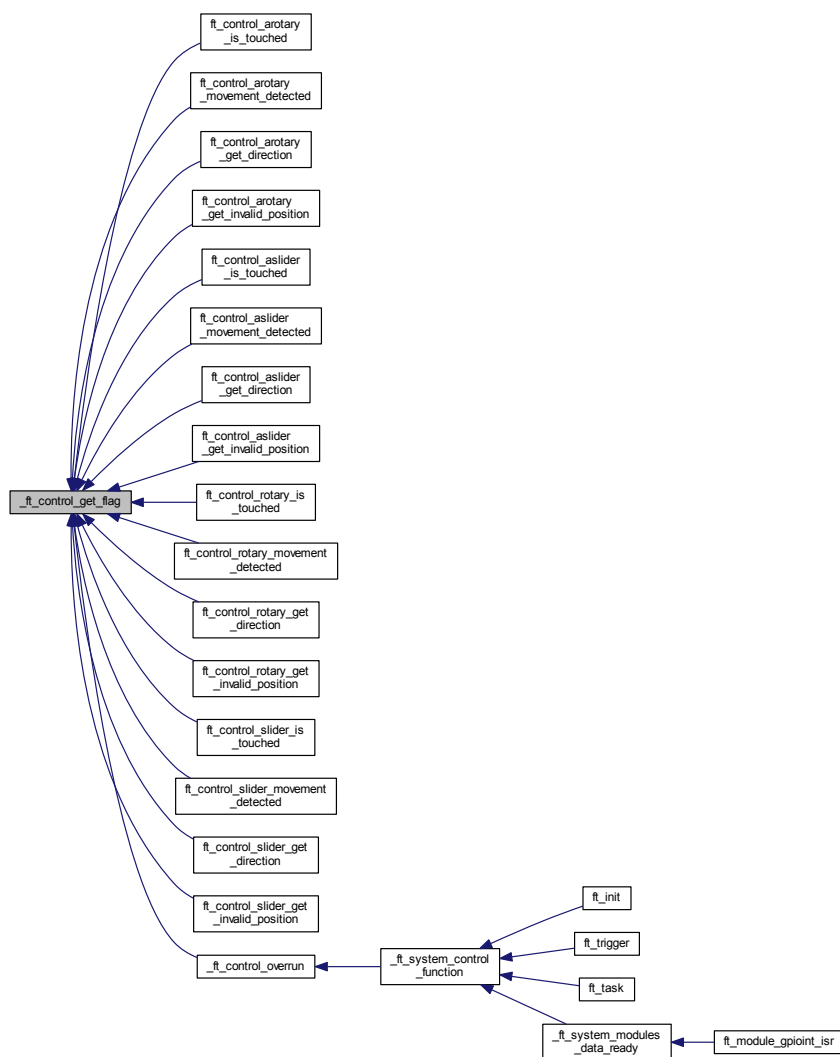


Controls

7.9.2.4.2.9 `uint32_t ft_control_get_first_elec_touched (uint32_t current_state)`

7.9.2.4.2.10 `static uint32_t ft_control_get_flag (const struct ft_control_data * control, uint32_t flags) [inline], [static]`

Here is the caller graph for this function:



7.9.2.4.2.11 `uint32_t ft_control_get_last_elec_touched (uint32_t current_state)`

7.9.2.4.2.12 `uint32_t ft_control_get_touch_count (uint32_t current_state)`

7.9.2.4.2.13 `struct ft_control_data* ft_control_init (const struct ft_control * control)`

Parameters

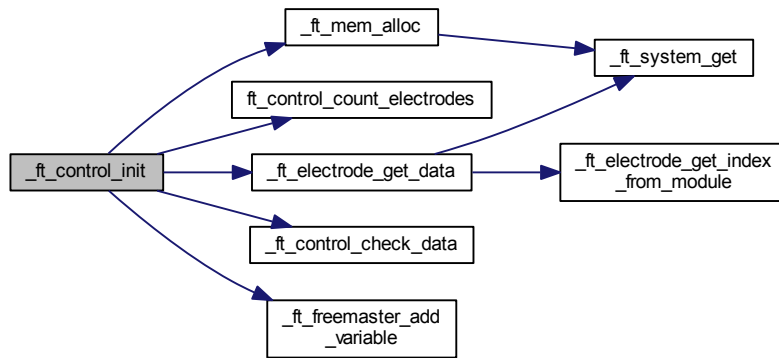
<i>control</i>	Pointer to the control.
----------------	-------------------------

Returns

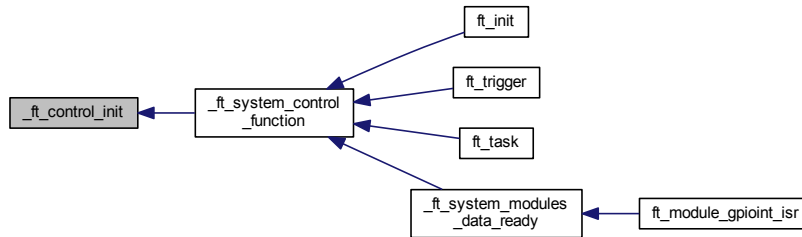
Pointer to create the control data structure. In case of a fail it returns NULL.

The function creates and initializes the control data structure, including the special data of the selected control (keypad, rotary, and so on).

Here is the call graph for this function:



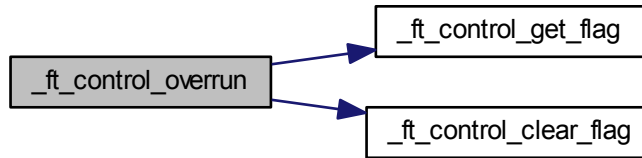
Here is the caller graph for this function:



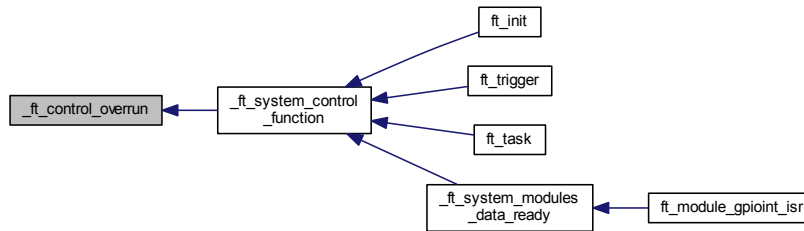
Controls

7.9.2.4.2.14 `static int32_t _ft_control_overrun (struct ft_control_data * control) [inline], [static]`

Here is the call graph for this function:

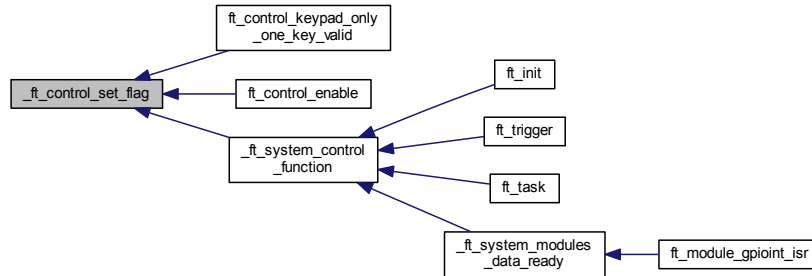


Here is the caller graph for this function:



**7.9.2.4.2.15 static void _ft_control_set_flag (struct ft_control_data * control, uint32_t flags)
[inline], [static]**

Here is the caller graph for this function:



7.9.2.4.2.16 void _ft_control_set_flag_all_elec (struct ft_control_data * control, uint32_t flags)

Here is the call graph for this function:



Electrodes

7.10 Electrodes

7.10.1 Overview

Electrodes are data objects which are used by data acquisition algorithms to store per-electrode data as well as resulting signal and touch / time stamp information.

Each Electrode provides at minimum the processed and normalized signal value, the baseline value, and touch / timestamp buffer containing the time of last few touch and release events. All such common information are contained in the [ft_electrode](#) structure type. Also, the electrode contains information about the key detector used to detect touches for this physical electrode (this is mandatory). This brings the advantage that each electrode has its own setting of the key detector independent on the module used. It contains information about hardware pin, immediate touch status, and time stamps of the last few touch or release events.

The private electrodes API provides all the functionality needed to handle the private needs of the Freescale Touch library. Collaboration diagram for Electrodes:



Modules

- [API Functions](#)

Data Structures

- union [ft_electrode_special_data](#)
- struct [ft_electrode_data](#)

Enumerations

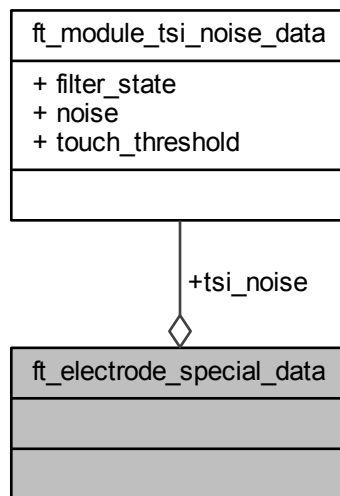
- enum [ft_electrode_flags](#) {
[FT_ELECTRODE_LOCK_BASELINE_REQ_FLAG](#),
[FT_ELECTRODE_LOCK_BASELINE_FLAG](#),
[FT_ELECTRODE_DIGITAL_RESULT_ONLY_FLAG](#) }

7.10.2 Data Structure Documentation

7.10.2.1 union ft_electrode_special_data

The pointer to the special data of the electrode. Each module has its own types handled by this union to keep clearance of the types.

Collaboration diagram for ft_electrode_special_data:



Data Fields

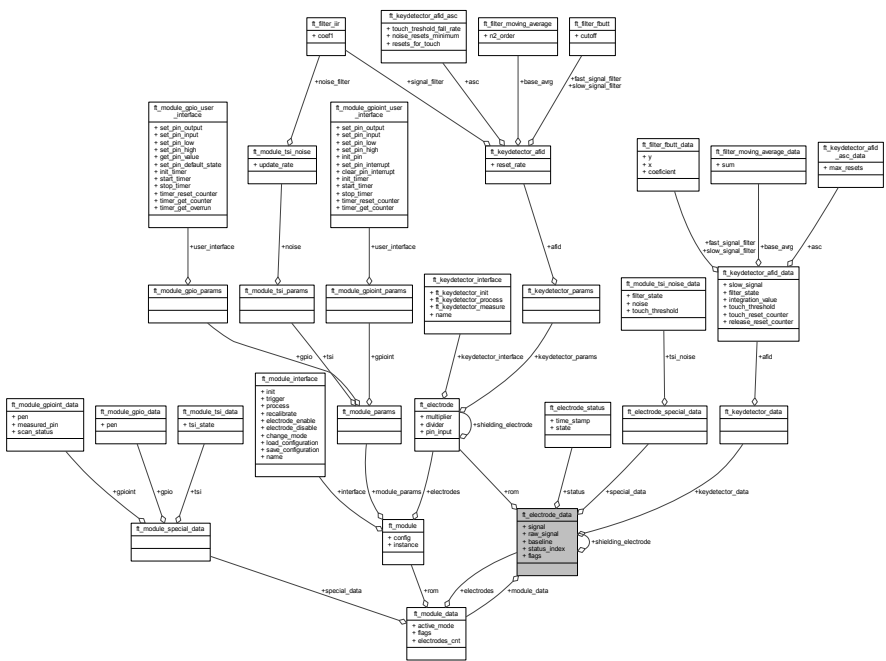
struct ft_module_tsi- _noise_data *	tsi_noise	Pointer to the TSI noise mode data for the TSI module.
---	-----------	--

7.10.2.2 struct ft_electrode_data

Electrode RAM structure used to store volatile parameters, flags, and other data, to enable a generic behavior of the Electrode. You must allocate this structure and put a pointer into the [ft_electrode](#) structure, when the electrode is being configured and registered in the module and control.

Electrodes

Collaboration diagram for ft_electrode_data:



Data Fields

uint16_t	baseline	Baseline.
uint32_t	flags	Flags.
union ft_keydetector_data	keydetector_data	Pointer to the key detector data structure.
struct ft_module_data *	module_data	Pointer to the owner module data.
uint16_t	raw_signal	Raw data to be handled in the task process.
struct ft_electrode *	rom	Pointer to the electrode user parameters.

struct <code>ft_electrode_data</code> *	shielding_electrode	Pointer to a shielding electrode (if it is used).
uint16_t	signal	Processed signal.
union <code>ft_electrode_special_data</code>	special_data	Pointer to the special data (for example noise mode data for the TSI).
struct <code>ft_electrode_status</code>	status[FT_ELECTRODE_STATUS_HISTORY_COUNT]	Statuses.
uint8_t	status_index	Status index.

7.10.3 Enumeration Type Documentation

7.10.3.1 enum `ft_electrode_flags`

Electrodes flags which can be set/cleared.

Enumerator

FT_ELECTRODE_LOCK_BASELINE_REQ_FLAG This flag signals that the electrode's baseline should be locked (can't be updated).

FT_ELECTRODE_LOCK_BASELINE_FLAG This flag signals that the electrode's baseline is locked (cannot be updated).

FT_ELECTRODE_DIGITAL_RESULT_ONLY_FLAG This flag signals that the electrode's event does not have analog information (cannot be used for analog controls).

Electrodes

7.10.4 API Functions

7.10.4.1 Overview

The functions in this category can be used to manipulate the Electrode objects. Collaboration diagram for API Functions:



Functions

- `struct ft_electrode_data * _ft_electrode_get_data` (const struct `ft_electrode` *electrode)
Get electrode data structure pointer.
- `int32_t _ft_electrode_get_index_from_module` (const struct `ft_module` *module, const struct `ft_electrode` *electrode)
Get the electrode index in the module electrode array structure pointer.
- `struct ft_electrode_data * _ft_electrode_init` (struct `ft_module_data` *module, const struct `ft_electrode` *electrode)
Initialize an electrode object.
- `uint32_t _ft_electrode_shielding_process` (struct `ft_electrode_data` *electrode, uint32_t signal)
Process shielding if it is enabled, otherwise it returns the same value.
- `uint32_t _ft_electrode_normalization_process` (const struct `ft_electrode_data` *electrode, uint32_t signal)
Scale signal.
- `void _ft_electrode_set_signal` (struct `ft_electrode_data` *electrode, uint32_t signal)
Set the signal for the electrode.
- `void _ft_electrode_set_raw_signal` (struct `ft_electrode_data` *electrode, uint32_t signal)
Set the raw signal for the electrode.
- `void _ft_electrode_set_status` (struct `ft_electrode_data` *electrode, int32_t state)
Set the status of the electrode.
- `static void _ft_electrode_set_flag` (struct `ft_electrode_data` *electrode, uint32_t flags)
- `static void _ft_electrode_clear_flag` (struct `ft_electrode_data` *electrode, uint32_t flags)
- `static uint32_t _ft_electrode_get_flag` (struct `ft_electrode_data` *electrode, uint32_t flags)
- `uint32_t _ft_electrode_get_time_offset_period` (const struct `ft_electrode_data` *electrode, uint32_t event_period)
Determine, whether the specified time (or its multiples) has elapsed since the last electrode event.
- `int32_t _ft_electrode_get_last_status` (const struct `ft_electrode_data` *electrode)
Get the last known electrode status.
- `uint32_t _ft_electrode_get_time_offset` (const struct `ft_electrode_data` *electrode)
Get the time since the last electrode event.
- `uint32_t _ft_electrode_get_signal` (const struct `ft_electrode_data` *electrode)

- Get the normalized and processed electrode signal.*

 - `int32_t _ft_electrode_get_status` (const struct `ft_electrode_data` *electrode, uint32_t index)

Get the electrode status by specifying the history buffer index.
- `uint32_t _ft_electrode_get_last_time_stamp` (const struct `ft_electrode_data` *electrode)

Get the last known electrode time-stamp.
- `uint32_t _ft_electrode_get_time_stamp` (const struct `ft_electrode_data` *electrode, uint32_t index)

Get the electrode status time-stamp by specifying the history buffer index.
- `uint32_t _ft_electrode_get_raw_signal` (const struct `ft_electrode_data` *electrode)

Get the raw electrode signal.
- `int32_t _ft_electrode_get_delta` (const struct `ft_electrode_data` *electrode)

Return difference between the signal and its baseline.
- `uint32_t _ft_electrode_is_touched` (const struct `ft_electrode_data` *electrode)

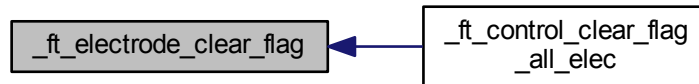
Get the state of the electrode.
- struct `ft_electrode` * `_ft_electrode_get_shield` (const struct `ft_electrode` *electrode)

Get the shielding electrode.

7.10.4.2 Function Documentation

7.10.4.2.1 `static void _ft_electrode_clear_flag (struct ft_electrode_data * electrode, uint32_t flags) [inline], [static]`

Here is the caller graph for this function:



7.10.4.2.2 `struct ft_electrode_data* _ft_electrode_get_data (const struct ft_electrode * electrode)`

Parameters

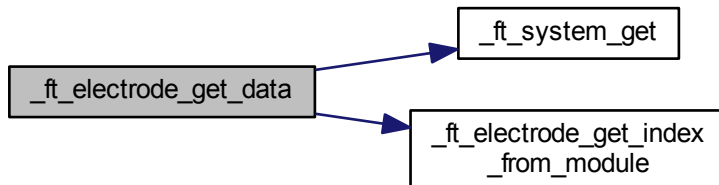
<i>electrode</i>	Pointer to the electrode user parameter structure.
------------------	--

Returns

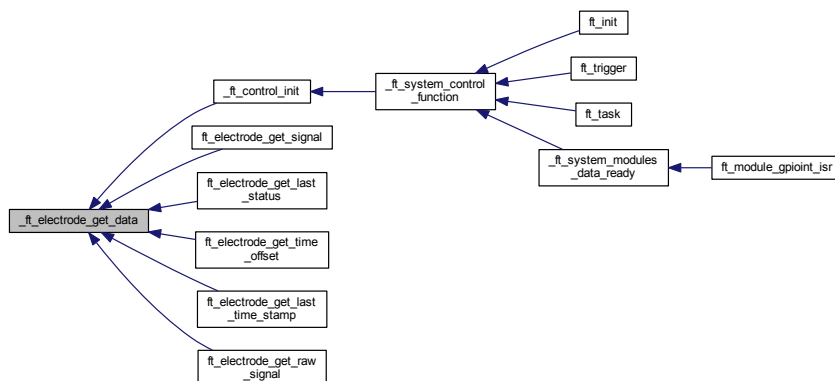
The pointer to the data electrode structure that is represented by the handled user parameter structure pointer.

Electrodes

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.4.2.3 int32_t _ft_electrode_get_delta (const struct ft_electrode_data * *electrode*)

Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

Immediate delta value between the processed signal and its baseline (idle) value.

Here is the call graph for this function:



7.10.4.2.4 static uint32_t ft_electrode_get_flag (struct ft_electrode_data * *electrode*, uint32_t *flags*) [inline], [static]

7.10.4.2.5 int32_t ft_electrode_get_index_from_module (const struct ft_module * *module*, const struct ft_electrode * *electrode*)

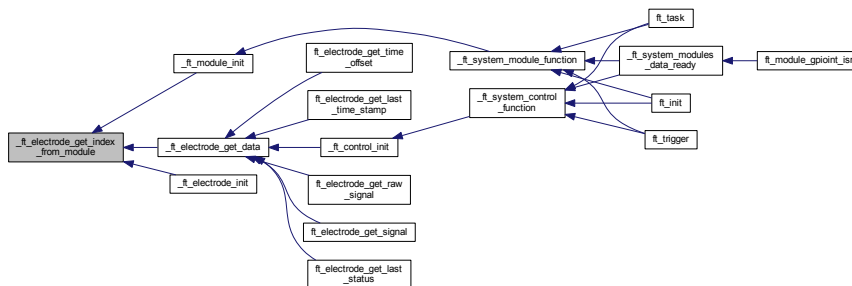
Parameters

<i>electrode</i>	Pointer to the electrode user parameter structure.
------------------	--

Returns

The index to the electrode structure array, in case that the electrode is not available it returns -1.

Here is the caller graph for this function:



7.10.4.2.6 int32_t ft_electrode_get_last_status (const struct ft_electrode_data * *electrode*)

Electrodes

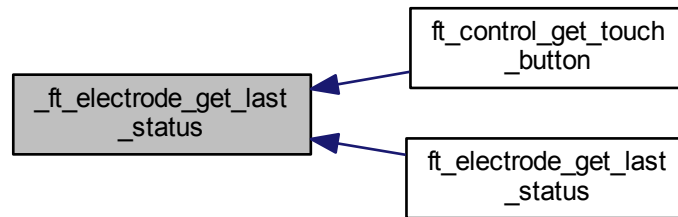
Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

Current electrode status.

Here is the caller graph for this function:



7.10.4.2.7 `uint32_t _ft_electrode_get_last_time_stamp (const struct ft_electrode_data * electrode)`

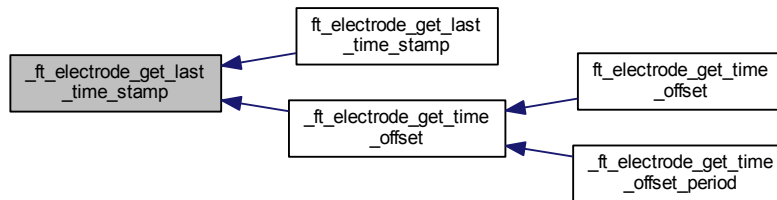
Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

Current electrode status.

Here is the caller graph for this function:



7.10.4.2.8 uint32_t ft_electrode_get_raw_signal (const struct ft_electrode_data * *electrode*)

Parameters

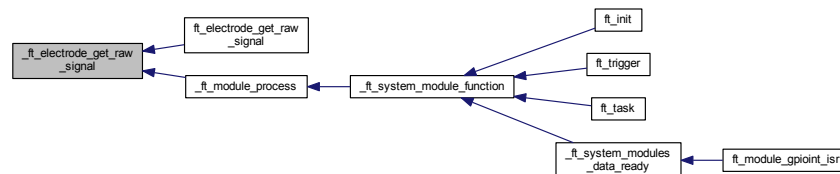
<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

Electrode signal, as it is measured by the physical module.

The raw signal is used internally by the filtering and normalization algorithms to calculate the real electrode signal value, which is good to be compared with the signals coming from other electrodes.

Here is the caller graph for this function:



7.10.4.2.9 struct ft_electrode* _ft_electrode_get_shield (const struct ft_electrode * *electrode*)

Electrodes

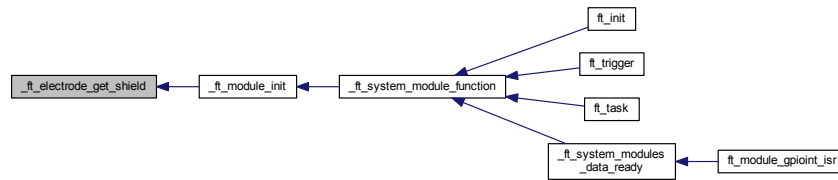
Parameters

<i>electrode</i>	Pointer to the electrode.
------------------	---------------------------

Returns

Pointer to the shielding electrode, if available.

Here is the caller graph for this function:



7.10.4.2.10 uint32_t _ft_electrode_get_signal (const struct ft_electrode_data * *electrode*)

Parameters

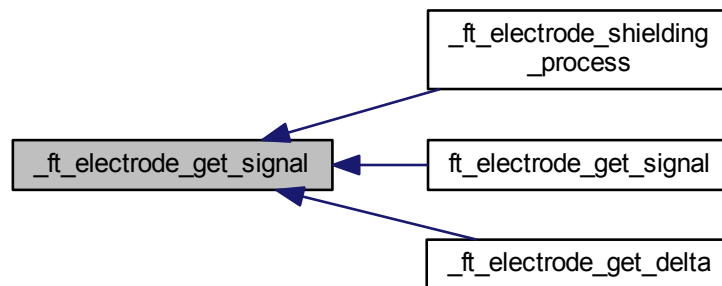
<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

Signal calculated from the last raw value measured.

The signal value is calculated from the raw electrode capacitance or other physical signal by applying the filtering and normalization algorithms. This signal is used by "analog" [Controls](#), which estimate the finger position based on the signal value, rather than on a simple touch / release status.

Here is the caller graph for this function:



7.10.4.2.11 int32_t _ft_electrode_get_status (const struct ft_electrode_data * *electrode*, uint32_t *index*)

Parameters

<i>electrode</i>	Pointer to the electrode data.
<i>index</i>	Index of the required status.

Returns

- status within the ft_electrode_state, if the index is within the range
- FT_FAILURE if the index is out of range.

7.10.4.2.12 uint32_t _ft_electrode_get_time_offset (const struct ft_electrode_data * *electrode*)

Electrodes

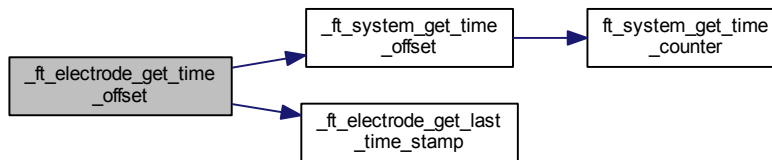
Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

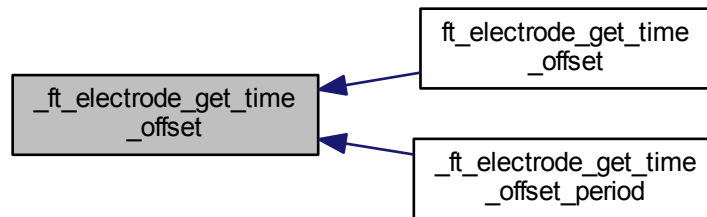
Returns

Time elapsed since the last electrode event.

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.4.2.13 `uint32_t ft_electrode_get_time_offset_period (const struct ft_electrode_data * electrode, uint32_t event_period)`

Parameters

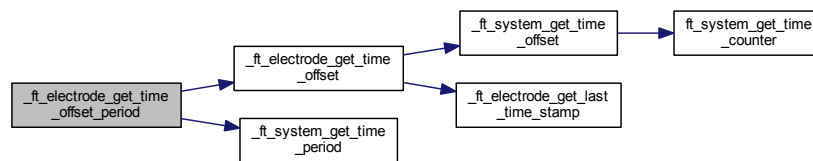
<i>electrode</i>	Pointer to the electrode data.
<i>event_period</i>	Number of time periods that should elapse since the last electrode event.

Returns

zero if the specified number of time periods has elapsed, or any whole multiple of this number has elapsed since the last electrode event.

This function can be used to determine the multiples of specified time interval since the electrode event has been detected.

Here is the call graph for this function:



7.10.4.2.14 `uint32_t ft_electrode_get_time_stamp (const struct ft_electrode_data * electrode, uint32_t index)`

Parameters

<i>electrode</i>	Pointer to the electrode data.
<i>index</i>	Index of the required status.

Returns

- non-zero value (valid time stamp)
- 0 - index out of range

7.10.4.2.15 `struct ft_electrode_data* ft_electrode_init (struct ft_module_data * module, const struct ft_electrode * electrode)`

Electrodes

Parameters

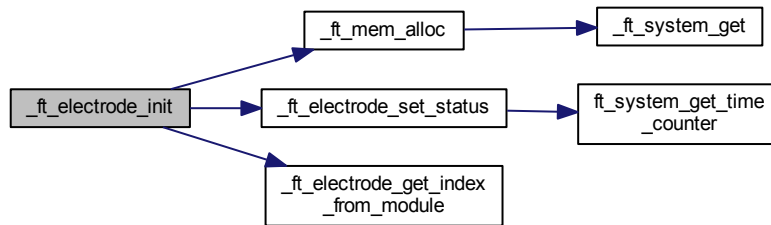
<i>module</i>	Pointer to the module data.
<i>electrode</i>	Pointer to the electrode.

Returns

Pointer to the newly-created electrode data structure. (In case of a fail, it returns NULL).

This function creates the electrode data, and resets the electrode's status and status index.

Here is the call graph for this function:



7.10.4.2.16 uint32_t _ft_electrode_is_touched (const struct ft_electrode_data * *electrode*)

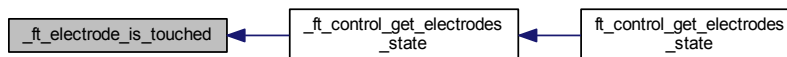
Parameters

<i>electrode</i>	Pointer to the electrode data.
------------------	--------------------------------

Returns

Non-zero if the current electrode status is "touched"; zero otherwise.

Here is the caller graph for this function:



7.10.4.2.17 `uint32_t ft_electrode_normalization_process (const struct ft_electrode_data *
electrode, uint32_t signal)`

Electrodes

Parameters

<i>electrode</i>	A pointer to the electrode data.
<i>signal</i>	

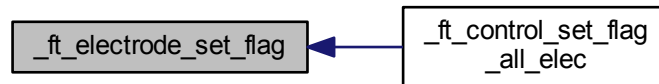
Returns

signal

Normalize the signal to working values. Values from [ft_electrode](#) divider or multiplier normalize the measured signal.

7.10.4.2.18 `static void _ft_electrode_set_flag (struct ft_electrode_data * electrode, uint32_t flags) [inline], [static]`

Here is the caller graph for this function:



7.10.4.2.19 `void _ft_electrode_set_raw_signal (struct ft_electrode_data * electrode, uint32_t signal)`

Parameters

<i>electrode</i>	Pointer to the electrode data.
<i>signal</i>	

Returns

none

Here is the caller graph for this function:



7.10.4.2.20 void _ft_electrode_set_signal (struct ft_electrode_data * *electrode*, uint32_t *signal*)

Parameters

<i>electrode</i>	A pointer to the electrode.
<i>signal</i>	

Returns

none

7.10.4.2.21 void _ft_electrode_set_status (struct ft_electrode_data * *electrode*, int32_t *state*)

Parameters

<i>electrode</i>	A pointer to the electrode data.
<i>state</i>	ft_electrode_state

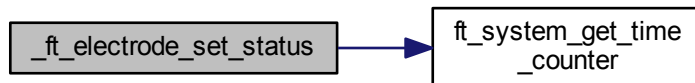
Returns

none

This function sets the state of the electrode, and assigns a time stamp from the system to the electrode.

Electrodes

Here is the call graph for this function:



Here is the caller graph for this function:



7.10.4.2.22 `uint32_t _ft_electrode_shielding_process (struct ft_electrode_data * electrode, uint32_t signal)`

Parameters

<i>electrode</i>	A pointer to the electrode data.
<i>signal</i>	Current signal value.

Returns

signal value.

The signal is subtracted by the baseline, and incremented by the signal. If the signal is greater than 0, it returns the signal value other than 0.

Here is the call graph for this function:



Filters

7.11 Filters

7.11.1 Overview

The filters data structure that is used in the Freescale Touch library. Collaboration diagram for Filters:



Modules

- [API Functions](#)

Data Structures

- struct [ft_filter_fbutt_data](#)
- struct [ft_filter_moving_average_data](#)

Enumerations

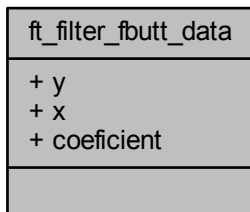
- enum [ft_filter_state](#) {
 [FT_FILTER_STATE_INIT](#),
 [FT_FILTER_STATE_RUN](#) }

7.11.2 Data Structure Documentation

7.11.2.1 struct ft_filter_fbutt_data

The butterworth filter context data.

Collaboration diagram for ft_filter_fbutt_data:



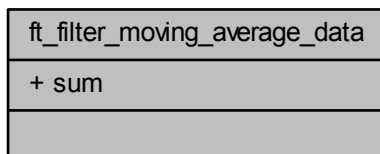
Data Fields

int32_t	coefficient	
int16_t	x	The previous input value.
int32_t	y	The last result of the filter.

7.11.2.2 struct ft_filter_moving_average_data

The moving average filter context data.

Collaboration diagram for ft_filter_moving_average_data:



Filters

Data Fields

int32_t	sum	The sum of the filter data
---------	-----	----------------------------

7.11.3 Enumeration Type Documentation

7.11.3.1 enum ft_filter_state

The filter state definition.

Enumerator

FT_FILTER_STATE_INIT The filter is initialized.

FT_FILTER_STATE_RUN The filter is running correctly.

7.11.4 API Functions

7.11.4.1 Overview

General Private Function definition of filters. Collaboration diagram for API Functions:



Functions

- `uint32_t _ft_abs_int32 (int32_t lsrc)`
Gets the absolute value.
- `void _ft_filter_fbutt_init (const struct ft_filter_fbutt *rom, struct ft_filter_fbutt_data *ram, uint32_t signal)`
Initialize the ButterWorth filter for the first use.
- `uint16_t _ft_filter_fbutt_process (struct ft_filter_fbutt_data *ram, uint16_t signal)`
Process signal fbutt filter.
- `uint32_t _ft_filter_iir_process (const struct ft_filter_iir *rom, uint32_t signal, uint32_t previous_signal)`
Process signal IIR filter.
- `int32_t _ft_filter_moving_average_init (const struct ft_filter_moving_average *rom, struct ft_filter_moving_average_data *ram, uint16_t value)`
This function initialize moving average filter.
- `uint32_t _ft_filter_moving_average_process (const struct ft_filter_moving_average *rom, struct ft_filter_moving_average_data *ram, uint16_t value)`
This function compute moving average filter.
- `uint16_t _ft_filter_abs (int16_t value)`
This function compute absolute value (16-bit version).
- `uint16_t _ft_filter_limit_u (int32_t value, uint16_t limit_l, uint16_t limit_h)`
This function limit the input value in allowed range (16-bit version).
- `uint16_t _ft_filter_deadrange_u (uint16_t value, uint16_t base, uint16_t range)`
This function make dead range for input value out of the allowed range (16-bit version).
- `int32_t _ft_filter_is_deadrange_u (uint16_t value, uint16_t base, uint16_t range)`
This function checks if input value is inside of the deadband range (16-bit version).

7.11.4.2 Function Documentation

7.11.4.2.1 `uint32_t _ft_abs_int32 (int32_t lsrc)`

Filters

Parameters

<i>lsrc</i>	Input signed 32-bit number.
-------------	-----------------------------

Returns

Unsigned 32-bit absolute value of the input number.

7.11.4.2.2 uint16_t _ft_filter_abs (int16_t value)

Parameters

<i>value</i>	Input signed value.
--------------	---------------------

Returns

Absolute unsigned value of input.

7.11.4.2.3 uint16_t _ft_filter_deadrange_u (uint16_t value, uint16_t base, uint16_t range)

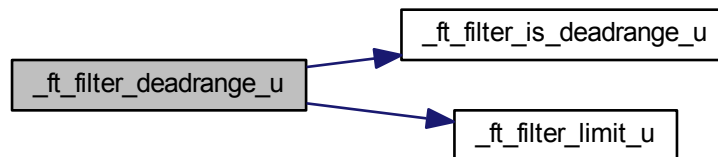
Parameters

<i>value</i>	Input value.
<i>base</i>	Base value of deadband range.
<i>range</i>	Range of the deadband range (one half).

Returns

Result value out of the deadband range.

Here is the call graph for this function:



7.11.4.2.4 `void _ft_filter_fbutt_init (const struct ft_filter_fbutt * rom, struct ft_filter_fbutt_data * ram, uint32_t signal)`

Filters

Parameters

<i>rom</i>	Pointer to the ft_filter_fbutt structure.
<i>ram</i>	Pointer to the ft_filter_fbutt_data .
<i>signal</i>	Input signal.

Returns

none

Here is the call graph for this function:



7.11.4.2.5 uint16_t ft_filter_fbutt_process (struct ft_filter_fbutt_data * ram, uint16_t signal)

Parameters

<i>ram</i>	Pointer to the ft_filter_fbutt_data structure.
<i>signal</i>	Input signal.

Returns

Filtered signal.

Returns signal equal

7.11.4.2.6 uint32_t ft_filter_iir_process (const struct ft_filter_iir * rom, uint32_t signal, uint32_t previous_signal)

Parameters

<i>rom</i>	Pointer to ft_filter_iir
<i>signal</i>	Current signal.
<i>previous_- signal</i>	Previous signal

Returns

signal

Process the signal, using the following equation: $y(n) = (1 / (coef + 1)) * \text{current signal} + (coef / (coef + 1)) * \text{previous_signal}$

7.11.4.2.7 int32_t _ft_filter_is_deadrange_u (uint16_t value, uint16_t base, uint16_t range)

Parameters

<i>value</i>	Input value.
<i>base</i>	Base value of deadband range.
<i>range</i>	Range of the deadband range (one half).

Returns

Result TRUE - value is in deadband range. FALSE - value is out of deadband range.

Here is the caller graph for this function:


7.11.4.2.8 uint16_t _ft_filter_limit_u (int32_t value, uint16_t limit_l, uint16_t limit_h)

Filters

Parameters

<i>value</i>	Input value.
<i>limit_l</i>	Limitation low range border.
<i>limit_H</i>	Limitation high range border.

Returns

Result value.

Here is the caller graph for this function:



7.11.4.2.9 `int32_t ft_filter_moving_average_init (const struct ft_filter_moving_average * rom, struct ft_filter_moving_average_data * ram, uint16_t value)`

Parameters

<i>rom</i>	Pointer to ft_filter_moving_average structure.
<i>ram</i>	Pointer to ft_filter_moving_average_data structure.
<i>value</i>	Input initial value.

Returns

result of operation (0 - OK, otherwise - FALSE).

7.11.4.2.10 `uint32_t ft_filter_moving_average_process (const struct ft_filter_moving_average * rom, struct ft_filter_moving_average_data * ram, uint16_t value)`

Parameters

<i>rom</i>	Pointer to ft_filter_moving_average structure.
<i>ram</i>	Pointer to ft_filter_moving_average_data structure.
<i>value</i>	Input new value.

Returns

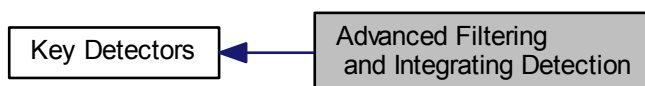
Current value of the moving average filter.

Filters

7.11.5 Advanced Filtering and Integrating Detection

7.11.5.1 Overview

The AFID (Advanced Filtering and Integrating Detection) key detector is based on using two IIR filters with different depths (one short / fast, the other long / slow) and on integrating the difference between the two filtered signals. The algorithm uses two thresholds; the touch threshold and the release threshold. The touch threshold is defined in the sensitivity register. The release threshold has twice the lower level than the touch threshold. If the integrated signal is higher than the touch threshold, or lower than the release threshold, then the integrated signal is reset. The Touch state is reported for the electrode when the first touch reset is detected. The Release state is reported, when as many release resets as the touch resets were detected during the previous touch state. Collaboration diagram for Advanced Filtering and Integrating Detection:



Data Structures

- struct [ft_keydetector_afid_asc_data](#)
- struct [ft_keydetector_afid_data](#)

Macros

- #define [FT_KEYDETECTOR_AFID_INITIAL_INTEGRATOR_VALUE](#)
- #define [FT_KEYDETECTOR_AFID_INITIAL_RESET_TOUCH_COUNTER_VALUE](#)
- #define [FT_KEYDETECTOR_AFID_INITIAL_RESET_RELEASE_COUNTER_VALUE](#)

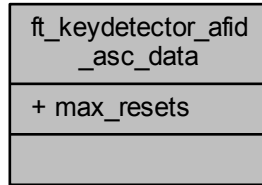
7.11.5.2 Data Structure Documentation

7.11.5.2.1 struct [ft_keydetector_afid_asc_data](#)

AFID Automatic Sensitive Calibration RAM structure; This structure is used for internal algorithms to store the data while evaluating the AFID. Contains data of the calculating result and auxiliary variables.

This structure only manages and uses the internal methods.

Collaboration diagram for ft_keydetector_afid_asc_data:



Data Fields

int32_t	max_resets	
---------	------------	--

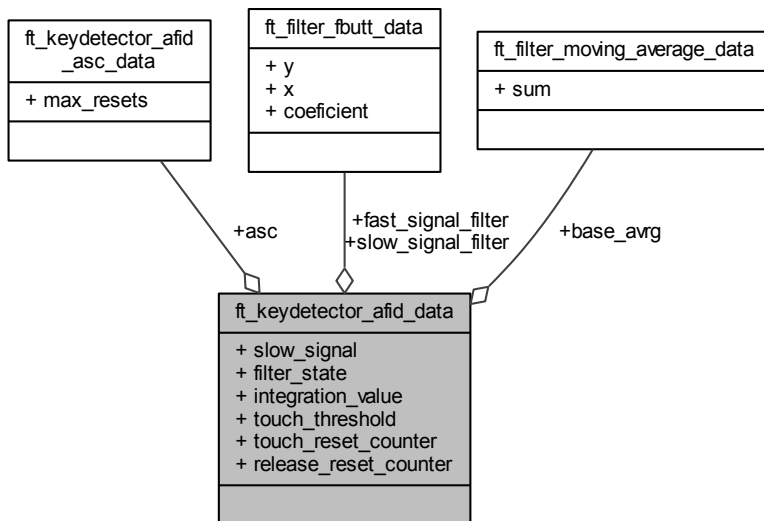
7.11.5.2.2 struct ft_keydetector_afid_data

AFID Ram structure; This structure is used for internal algorithms to store the data while evaluating the AFID. Contains the data of the calculating result and auxiliary variables.

This structure only manages and uses the internal methods.

Filters

Collaboration diagram for ft_keydetector_afid_data:



Data Fields

struct ft_keydetector_afid_asc_data	asc	Storage of ASC (Automatic sensitive calibration) data for AFID
struct ft_filter_moving_average_data	base_avrg	Base line moving average filter data.
struct ft_filter_fbutt_data	fast_signal_filter	Signal fast butterworth filter data storage.
uint8_t	filter_state	State of filter.
int32_t	integration_value	Current value of internal integrator.

uint32_t	release_reset_- counter	Count of release events resets.
uint16_t	slow_signal	Slow signal value .
struct ft_filter_- fbutt_data	slow_signal_- filter	Signal slow butterworth filter data storage.
uint32_t	touch_reset_- counter	Count of touch resets.
uint32_t	touch_- threshold	Current threshold value for integrator resets.

7.11.5.3 Macro Definition Documentation

7.11.5.3.1 #define FT_KEYDETECTOR_AFID_INITIAL_INTEGRATOR_VALUE

The initial integration value of the AFID.

7.11.5.3.2 #define FT_KEYDETECTOR_AFID_INITIAL_RESET_RELEASE_COUNTER_VALUE

The reset threshold value of the AFID.

7.11.5.3.3 #define FT_KEYDETECTOR_AFID_INITIAL_RESET_TOUCH_COUNTER_VALUE

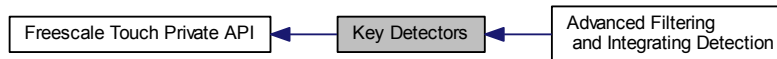
The initial reset counter value of the AFID.

Key Detectors

7.12 Key Detectors

7.12.1 Overview

The key detector module determines, whether an electrode has been touched or released, based on the values obtained by the capacitive sensing layer. Along with this detection, the key detector module uses a debounce algorithm that prevents the library from false touches. The key detector also detects, reports, and acts on fault conditions during the scanning process. Two main fault conditions are identified according to the electrode short-circuited either to the supply voltage or to the ground. The same conditions can be caused by a small capacitance (equal to a short circuit to supply voltage) or by a big capacitance (equals to a short circuit to the ground). Collaboration diagram for Key Detectors:



Modules

- [Advanced Filtering and Integrating Detection](#)

Data Structures

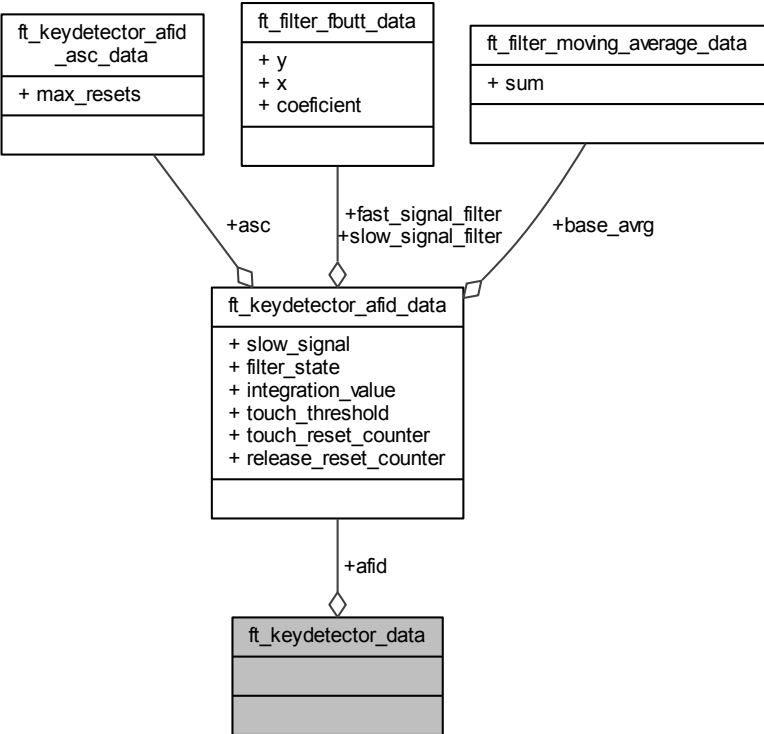
- union [ft_keydetector_data](#)

7.12.2 Data Structure Documentation

7.12.2.1 union ft_keydetector_data

The key detector optional run-time data.

Collaboration diagram for ft_keydetector_data:



Data Fields

struct ft_keydetector- _afid_data *	afid	AFID electrode run-time data
--	------	------------------------------

Key Detectors

7.12.3 GPIO module

7.12.3.1 Overview

The GPIO module describes the hardware configuration and control of the elementary functionality of the method that is using standard GPIO pins of the MCU.

The GPIO method is designed for all general processors that have a GPIO module. Collaboration diagram for GPIO module:



Data Structures

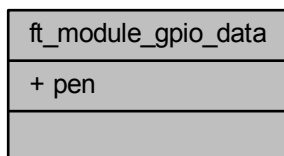
- struct [ft_module_gpio_data](#)

7.12.3.2 Data Structure Documentation

7.12.3.2.1 struct ft_module_gpio_data

GPIO module's RAM. This structure contains

Collaboration diagram for ft_module_gpio_data:



Data Fields

uint32_t	pen	PEN - enablement of all modules electrodes
----------	-----	--

Key Detectors

7.12.4 GPIO interrupt module

7.12.4.1 Overview

The GPIO interrupt module describes the hardware configuration and control of the elementary functionality of the method that is using standard GPIO pins of the MCU with the GPIO and timer interrupts.

The GPIO interrupt method is designed for all general processors that have a GPIO module with interrupt capability. Collaboration diagram for GPIO interrupt module:



Data Structures

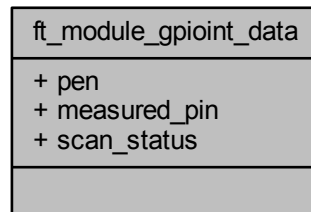
- struct [ft_module_gpoint_data](#)

7.12.4.2 Data Structure Documentation

7.12.4.2.1 struct ft_module_gpoint_data

GPIO interrupt module's RAM. This structure contains

Collaboration diagram for `ft_module_gpoint_data`:



Data Fields

uint8_t	measured_pin	The currently measured pin
uint32_t	pen	PEN - enablement of all modules' electrodes
uint8_t	scan_status	Module's scanning status - see enum ft_gpio_scan_states

Key Detectors

7.12.5 TSI module

7.12.5.1 Overview

The TSI module describes the hardware configuration and control of elementary functionality of the TSI peripheral, it covers all versions of the TSI peripheral by a generic low-level driver API.

The TSI module is designed for processors that have a hardware TSI module with version 1, 2, or 4 (for example Kinetis L).

The module also handles the NOISE mode supported by TSI v4 (Kinetis L). Collaboration diagram for TSI module:



Data Structures

- struct [ft_module_tsi_noise_data](#)
- struct [ft_module_tsi_data](#)

Macros

- #define [FT_TSI_NOISE_INITIAL_TOUCH_THRESHOLD](#)
- #define [FT_TSI_NOISE_TOUCH_RANGE](#)

Enumerations

- enum [ft_module_tsi_flags](#) {
[FT_MODULE_IN_NOISE_MODE_FLAG](#),
[FT_MODULE_HAS_NOISE_MODE_FLAG](#),
[FT_MODULE_NOISE_MODE_REQ_FLAG](#) }

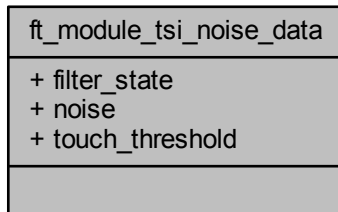
7.12.5.2 Data Structure Documentation

7.12.5.2.1 struct [ft_module_tsi_noise_data](#)

Noise data structure; This structure is used for internal algorithms to store data while evaluating the noise. Contains data of calculating the result and auxiliary variables.

This structure manages and uses internal methods only.

Collaboration diagram for ft_module_tsi_noise_data:

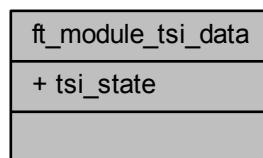


Data Fields

enum ft_filter_state	filter_state	Noise filter state.
uint8_t	noise	Noise current value.
uint8_t	touch_ threshold	Noise touch threshold run-time value.

7.12.5.2.2 struct ft_module_tsi_data

Collaboration diagram for ft_module_tsi_data:



Key Detectors

Data Fields

tsi_state_t	tsi_state	
-------------	-----------	--

7.12.5.3 Macro Definition Documentation

7.12.5.3.1 #define FT_TSI_NOISE_INITIAL_TOUCH_THRESHOLD

The TSI module noise mode initial touch threshold value.

7.12.5.3.2 #define FT_TSI_NOISE_TOUCH_RANGE

The TSI module noise mode touch range value.

7.12.5.4 Enumeration Type Documentation

7.12.5.4.1 enum ft_module_tsi_flags

The TSI module's noise mode flags definition.

Enumerator

FT_MODULE_IN_NOISE_MODE_FLAG This flag signalises that the module is currently in the noise mode.

FT_MODULE_HAS_NOISE_MODE_FLAG This flag signalises that the module can be switched to the noise mode (TSI v4).

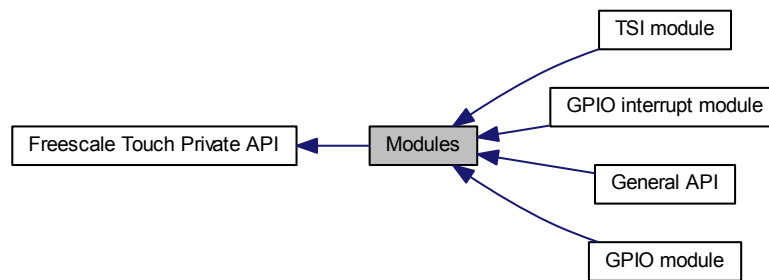
FT_MODULE_NOISE_MODE_REQ_FLAG This flag signalises that the module wants to switch to the noise mode.

7.13 Modules

7.13.1 Overview

Modules represent the data-acquisition layer in the Freescale Touch system, it is the layer that is tightly coupled to hardware module available on the Freescale MCU device.

Each Module implements a set of private functions contained in the [ft_modules_prv.h](#) file. Collaboration diagram for Modules:



Modules

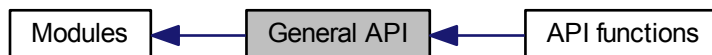
- [GPIO module](#)
- [GPIO interrupt module](#)
- [TSI module](#)
- [General API](#)

Modules

7.13.2 General API

7.13.2.1 Overview

General API and definition over all modules. Collaboration diagram for General API:



Modules

- [API functions](#)

Data Structures

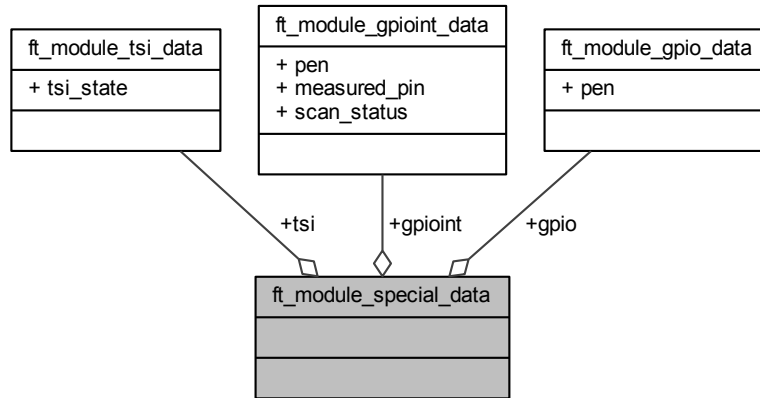
- union [ft_module_special_data](#)
- struct [ft_module_data](#)
- struct [ft_module_interface](#)

7.13.2.2 Data Structure Documentation

7.13.2.2.1 union ft_module_special_data

The module optional run-time data.

Collaboration diagram for ft_module_special_data:



Data Fields

struct ft_module_ gpio_data *	gpio	GPIO module run-time data
struct ft_module_ gpioint_data *	gpioint	GPIO interrupt module run-time data
struct ft_module_tsi- _data *	tsi	TSI module run-time data

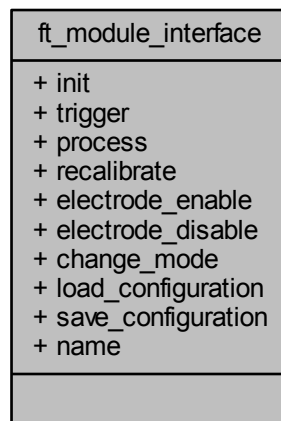
7.13.2.2.2 struct ft_module_data

Module RAM structure used to store volatile parameters, flags, and other data to enable a generic behavior of the Module. This is the main internal structure for a module in the FT library. A list of pointers to the electrode RAM data structure is created.

7.13.2.2.3 struct ft_module_interface

Module interface structure; each module uses this structure to register the entry points to its algorithms. This approach enables a kind-of polymorphism in the touch System. All modules are processed the same way from the System layer, regardless of the specific implementation. Each module type defines one static constant structure of this type to register its own initialization, triggering, processing functions, and functions for enabling or disabling of electrodes, low power, and proximity.

Collaboration diagram for ft_module_interface:



Data Fields

- int32_t(* [init](#))(struct [ft_module_data](#) *module)
- int32_t(* [trigger](#))(struct [ft_module_data](#) *module)
- int32_t(* [process](#))(struct [ft_module_data](#) *module)
- int32_t(* [recalibrate](#))(struct [ft_module_data](#) *module, void *configuration)
- int32_t(* [electrode_enable](#))(struct [ft_module_data](#) *module, const uint32_t elec_index)
- int32_t(* [electrode_disable](#))(struct [ft_module_data](#) *module, const uint32_t elec_index)
- int32_t(* [change_mode](#))(struct [ft_module_data](#) *module, const enum [ft_module_mode](#) mode, const struct [ft_electrode](#) *electrode)
- int32_t(* [load_configuration](#))(struct [ft_module_data](#) *module, const enum [ft_module_mode](#) mode, const void *config)
- int32_t(* [save_configuration](#))(struct [ft_module_data](#) *module, const enum [ft_module_mode](#) mode, void *config)
- const char * [name](#)

Modules

7.13.2.2.3.1 Field Documentation

7.13.2.2.3.1.1 `int32_t(* ft_module_interface::change_mode)(struct ft_module_data *module, const enum ft_module_mode mode, const struct ft_electrode *electrode)`

Change the the mode of the module.

7.13.2.2.3.1.2 `int32_t(* ft_module_interface::electrode_disable)(struct ft_module_data *module, const uint32_t elec_index)`

Disable the module electrode in hardware.

7.13.2.2.3.1.3 `int32_t(* ft_module_interface::electrode_enable)(struct ft_module_data *module, const uint32_t elec_index)`

Enable the module electrode in hardware.

7.13.2.2.3.1.4 `int32_t(* ft_module_interface::init)(struct ft_module_data *module)`

The initialization of the module.

7.13.2.2.3.1.5 `int32_t(* ft_module_interface::load_configuration)(struct ft_module_data *module, const enum ft_module_mode mode, const void *config)`

Load the configuration for the selected mode.

7.13.2.2.3.1.6 `const char* ft_module_interface::name`

A name of the variable of this type, used for FreeMASTER support purposes.

7.13.2.2.3.1.7 `int32_t(* ft_module_interface::process)(struct ft_module_data *module)`

Process the read data from the trigger event.

7.13.2.2.3.1.8 `int32_t(* ft_module_interface::recalibrate)(struct ft_module_data *module, void *configuration)`

Force recalibration of the module in the current mode.

7.13.2.2.3.1.9 `int32_t(* ft_module_interface::save_configuration)(struct ft_module_data *module, const enum ft_module_mode mode, void *config)`

Save the configuration of the selected mode.

7.13.2.2.3.1.10 `int32_t(* ft_module_interface::trigger)(struct ft_module_data *module)`

Send a trigger event into the module to perform hardware reading of the touches.

7.13.2.3 API functions

7.13.2.3.1 Overview

General Private Function definition of the modules. Collaboration diagram for API functions:



Functions

- struct `ft_module_data` * `_ft_module_get_data` (const struct `ft_module` *module)
Get the module data structure pointer.
- struct `ft_module_data` * `_ft_module_init` (const struct `ft_module` *module)
Init module.
- int32_t `_ft_module_trigger` (struct `ft_module_data` *module)
Trigger the start of measure event of the module.
- int32_t `_ft_module_process` (struct `ft_module_data` *module)
Process the module.
- static void `_ft_module_set_flag` (struct `ft_module_data` *module, uint32_t flags)
Set the flag of the module.
- static void `_ft_module_clear_flag` (struct `ft_module_data` *module, uint32_t flags)
Reset the flag of the module.
- static uint32_t `_ft_module_get_flag` (struct `ft_module_data` *module, uint32_t flags)
Return the flag of the module.
- static uint32_t `_ft_module_get_instance` (const struct `ft_module_data` *module)
Return the instance of the module.
- static void `_ft_module_set_mode` (struct `ft_module_data` *module, uint32_t mode)
Set the module's mode.
- static uint32_t `_ft_module_get_mode` (struct `ft_module_data` *module)
Get the module's mode.

7.13.2.3.2 Function Documentation

7.13.2.3.2.1 static void `_ft_module_clear_flag` (struct `ft_module_data` * *module*, uint32_t *flags*)
[inline], [static]

Modules

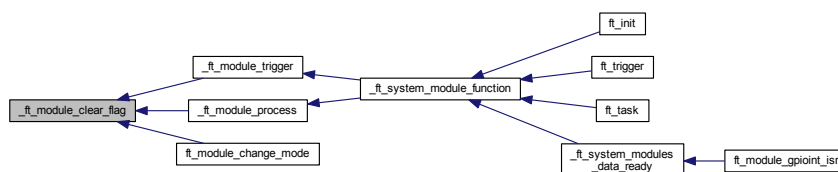
Parameters

<i>module</i>	Pointer to the FT module.
<i>flags</i>	The flags to be cleared.

Returns

void

Here is the caller graph for this function:



7.13.2.3.2 struct ft_module_data* _ft_module_get_data (const struct ft_module * module)

Parameters

<i>module</i>	Pointer to the module user parameter structure.
---------------	---

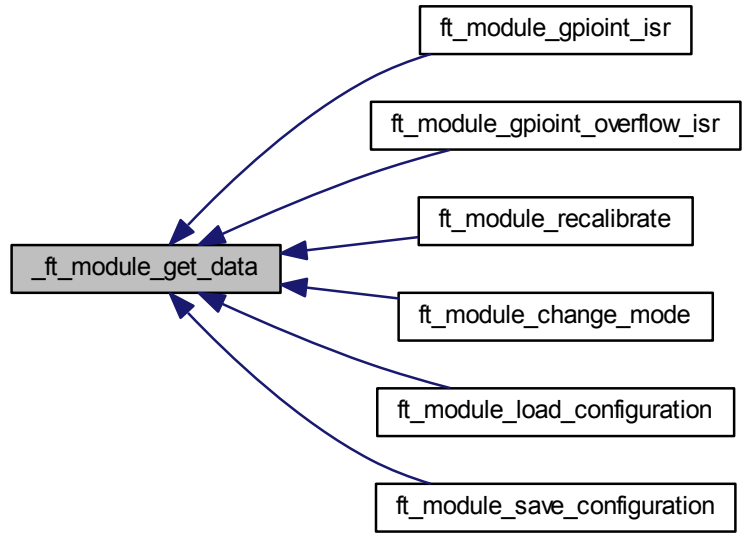
Returns

Pointer to the data module structure that is represented by the handled user parameter structure pointer.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.2.3.3 static uint32_t ft_module_get_flag (struct ft_module_data * module, uint32_t flags) [inline], [static]

Parameters

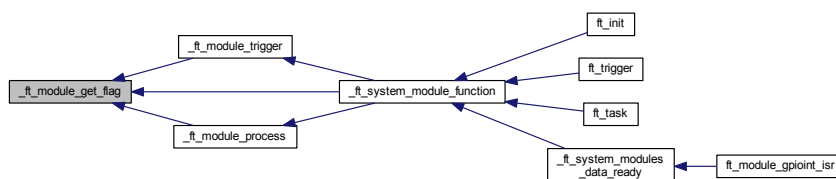
<i>module</i>	Pointer to the FT module.
<i>flags</i>	The flags to be tested

Modules

Returns

Non-zero if any of the tested flags are set. This is bit-wise AND of the control flags and the flags parameter.

Here is the caller graph for this function:



7.13.2.3.2.4 static uint32_t _ft_module_get_instance (const struct ft_module_data * *module*) [inline], [static]

Parameters

<i>module</i>	Pointer to the FT module.
---------------	---------------------------

Returns

instance

7.13.2.3.2.5 static uint32_t _ft_module_get_mode (struct ft_module_data * *module*) [inline], [static]

Parameters

<i>module</i>	Pointer to the FT module_data.
---------------	--------------------------------

Returns

mode.

7.13.2.3.2.6 struct ft_module_data* _ft_module_init (const struct ft_module * *module*)

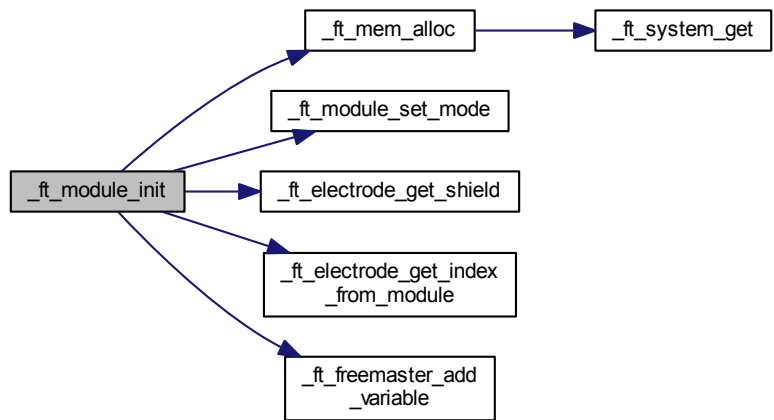
Parameters

<i>module</i>	Pointer to the module to be initialized.
---------------	--

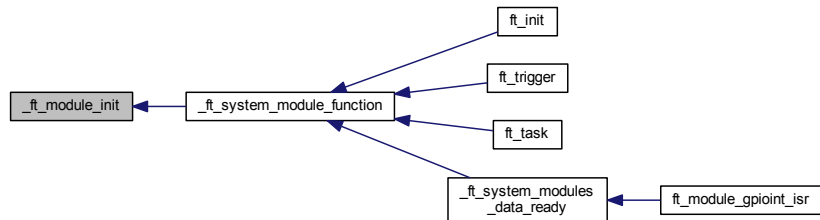
Returns

The result of the operation.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.2.3.2.7 int32_t ft_module_process (struct ft_module_data * module)

Modules

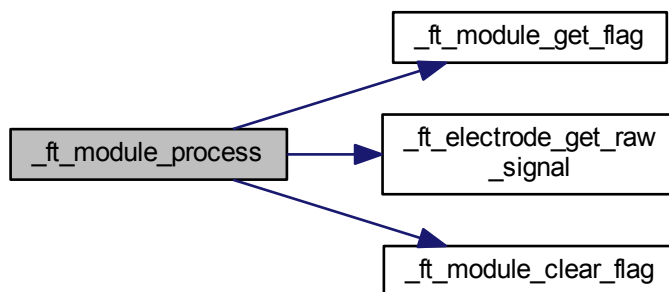
Parameters

<i>module</i>	Pointer to the module to be processed.
---------------	--

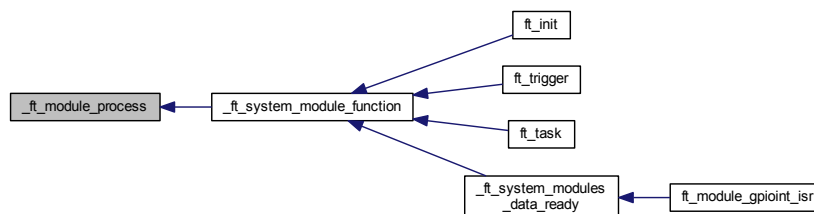
Returns

The result of the operation.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.2.3.2.8 `static void _ft_module_set_flag (struct ft_module_data * module, uint32_t flags)`
`[inline], [static]`

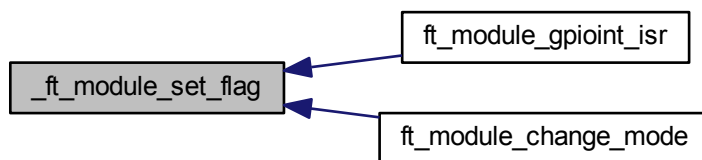
Parameters

<i>module</i>	Pointer to the FT module.
<i>flags</i>	The flags to be set.

Returns

void

Here is the caller graph for this function:



**7.13.2.3.2.9 static void _ft_module_set_mode (struct ft_module_data * *module*, uint32_t *mode*)
[inline], [static]**

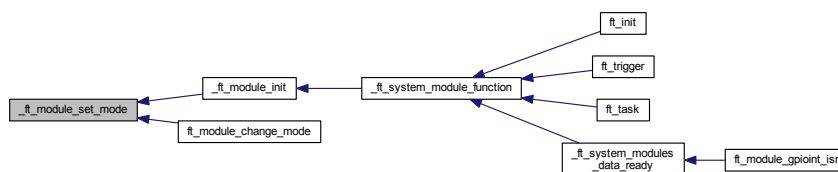
Parameters

<i>module</i>	Pointer to the FT module.
<i>mode</i>	

Returns

None.

Here is the caller graph for this function:



Modules

7.13.2.3.2.10 `int32_t ft_module_trigger (struct ft_module_data * module)`

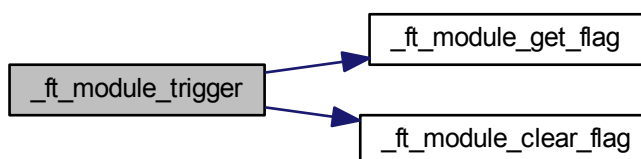
Parameters

<i>module</i>	Pointer to the module to be triggered.
---------------	--

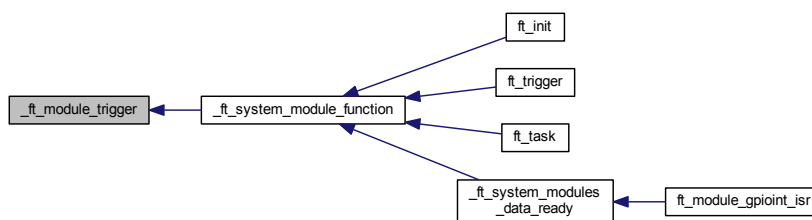
Returns

The result of the operation.

Here is the call graph for this function:



Here is the caller graph for this function:



FreeMASTER support

7.14 FreeMASTER support

7.14.1 Overview

Collaboration diagram for FreeMASTER support:



Modules

- [API functions](#)

7.14.2 API functions

7.14.2.1 Overview

General Private Function definition of the FreeMASTER support. Collaboration diagram for API functions:



Functions

- `int32_t _ft_freemaster_init` (void)
Initialized the Freescale touch FreeMASTER support system.
- `int32_t _ft_freemaster_add_variable` (const char *name, const char *type_name, void *address, uint32_t size)
This function adds a dynamic variable into the FreeMASTER TSA table.

7.14.2.2 Function Documentation

7.14.2.2.1 `int32_t _ft_freemaster_add_variable (const char * name, const char * type_name, void * address, uint32_t size)`

Parameters

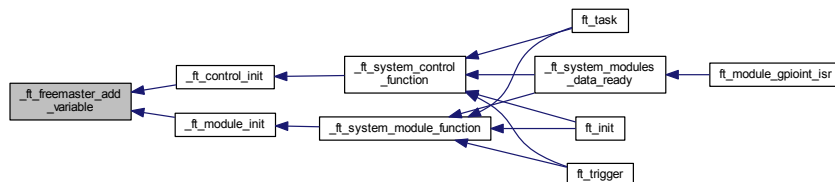
<i>name</i>	- pointer to the string with the name of the variable
<i>type_name</i>	- pointer to the string with the name of the variable type
<i>address</i>	- address of the variable
<i>size</i>	- size of the variable

FreeMASTER support

Returns

The result of the operation.

Here is the caller graph for this function:

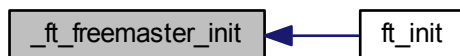


7.14.2.2.2 int32_t _ft_freemaster_init (void)

Returns

The result of operation.

Here is the caller graph for this function:



7.15 Memory Management

7.15.1 Overview

Collaboration diagram for Memory Management:



Modules

- [API functions](#)

Data Structures

- struct [ft_mem](#)

7.15.2 Data Structure Documentation

7.15.2.1 struct ft_mem

This structure contains the memory pool for all RAM data of the Freescale touch volatile data structures. This structure can be allocated in RAM.

Collaboration diagram for ft_mem:



Memory Management

Data Fields

uint8_t *	free_pointer	Pointer to the last free position in the memory pool.
uint8_t *	pool	Pointer to the allocated memory pool for the Freescale touch.
uint32_t	pool_size	Size of the allocated memory pool for the Freescale touch.

7.15.3 API functions

7.15.3.1 Overview

General Private Function definition of the memory support. Collaboration diagram for API functions:



Functions

- `int32_t _ft_mem_init (uint8_t *pool, const uint32_t size)`
Initialized the Freescale touch memory management system.
- `void * _ft_mem_alloc (const uint32_t size)`
Allocation of memory from the memory pool.
- `int32_t _ft_mem_deinit (void)`
Deinitialized the Freescale touch memory management system.

7.15.3.2 Function Documentation

7.15.3.2.1 void* _ft_mem_alloc (const uint32_t size)

Parameters

<i>size</i>	- size of the memory block to allocate.
-------------	---

Returns

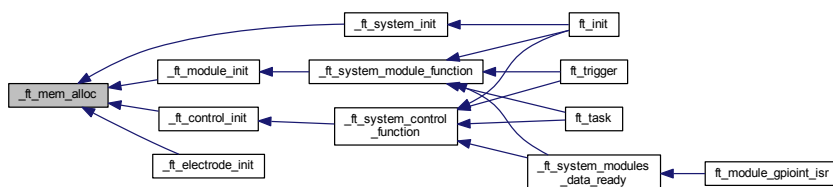
The pointer to the new allocated block, NULL in case there is not enough space in the pool.

Here is the call graph for this function:



Memory Management

Here is the caller graph for this function:



7.15.3.2.2 int32_t _ft_mem_deinit (void)

Returns

The result of the operation.

Here is the call graph for this function:



7.15.3.2.3 int32_t _ft_mem_init (uint8_t * pool, const uint32_t size)

Parameters

<i>pool</i>	- pointer to the allocated memory place to be used by the system (the size must be aligned by 4).
<i>size</i>	- size of the memory pool handled by the pool parameter.

Returns

The result of the operation.

Here is the call graph for this function:



Here is the caller graph for this function:

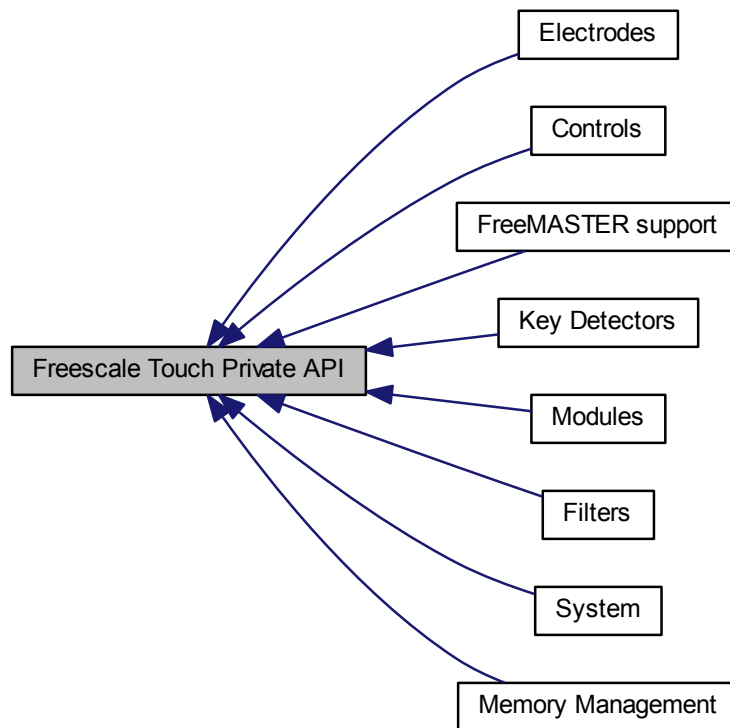


Memory Management

Chapter 8 Freescale Touch Private API

8.1 Overview

The functions documented in this module are the private ones used by the library itself. All the API here is just documented and not designed to be used by the user. Collaboration diagram for Freescale Touch Private API:



Modules

- [Controls](#)
- [Electrodes](#)
- [Filters](#)
- [Key Detectors](#)
- [Modules](#)
- [FreeMASTER support](#)
- [Memory Management](#)

Overview

- [System](#)

8.2 System

8.2.1 Overview

/

The system private API and definitions. Collaboration diagram for System:



Modules

- [API Functions](#)

Data Structures

- [struct ft_kernel](#)

Enumerations

- [enum ft_system_module_call](#) {
[FT_SYSTEM_MODULE_INIT](#),
[FT_SYSTEM_MODULE_TRIGGER](#),
[FT_SYSTEM_MODULE_PROCESS](#),
[FT_SYSTEM_MODULE_CHECK_DATA](#) }
- [enum ft_system_control_call](#) {
[FT_SYSTEM_CONTROL_INIT](#),
[FT_SYSTEM_CONTROL_PROCESS](#),
[FT_SYSTEM_CONTROL_OVERRUN](#),
[FT_SYSTEM_CONTROL_DATA_READY](#) }

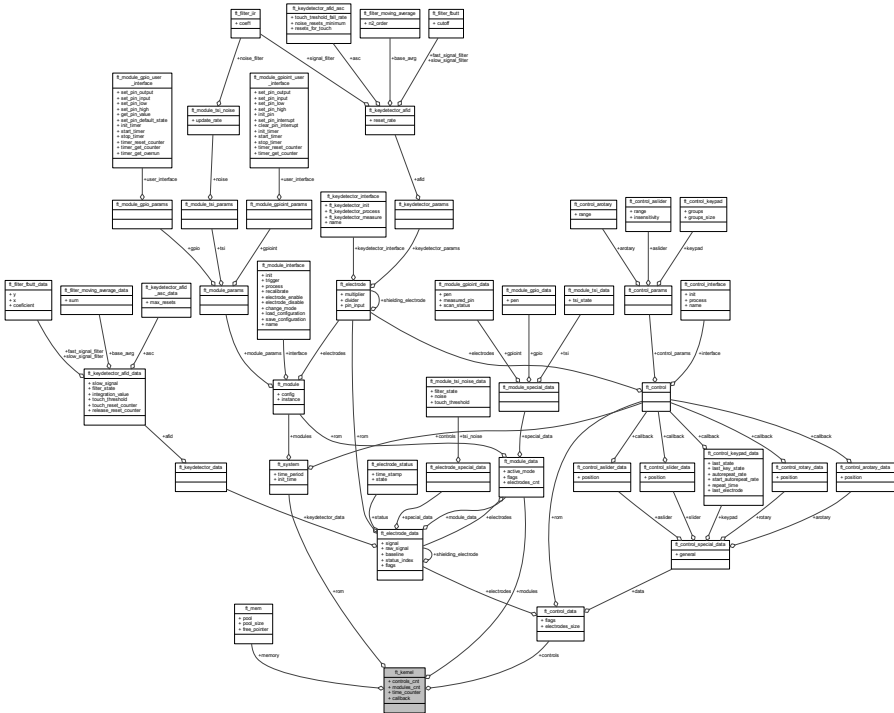
8.2.2 Data Structure Documentation

8.2.2.1 struct ft_kernel

System RAM structure used to store volatile parameters, counter, and system callback functions. This is the only statically placed RAM variable in the whole Freescale Touch library.

System

Collaboration diagram for ft_kernel:



Data Fields

<code>ft_system_callback</code>	callback	System event handler.
<code>ft_control_data</code> **	controls	Pointer to the list of controls. Can't be NULL.
<code>uint8_t</code>	controls_cnt	Count of the controls.
<code>ft_mem</code>	memory	System Memory handler
<code>ft_module_data</code> **	modules	Pointer to the list of modules. Can't be NULL.

uint8_t	modules_cnt	Count of the modules.
struct ft_system *	rom	Pointer to the system parameters.
uint32_t	time_counter	Time counter.

8.2.3 Enumeration Type Documentation

8.2.3.1 enum ft_system_control_call

Internal Controls function call identifier

Enumerator

- FT_SYSTEM_CONTROL_INIT*** Do control initialization.
- FT_SYSTEM_CONTROL_PROCESS*** Process the new data of control.
- FT_SYSTEM_CONTROL_OVERRUN*** Control data are overrun.
- FT_SYSTEM_CONTROL_DATA_READY*** Control data are ready.

8.2.3.2 enum ft_system_module_call

Internal Module function call identifier

Enumerator

- FT_SYSTEM_MODULE_INIT*** Do module initialization.
- FT_SYSTEM_MODULE_TRIGGER*** Send trigger event to module.
- FT_SYSTEM_MODULE_PROCESS*** Do process data in the module.
- FT_SYSTEM_MODULE_CHECK_DATA*** Check the module data.

System

8.2.4 API Functions

8.2.4.1 Overview

General Private Function definition of system. Collaboration diagram for API Functions:



Functions

- `struct ft_kernel * _ft_system_get (void)`
Obtain a pointer to the system.
- `void _ft_system_increment_time_counter (void)`
Increments the system counter.
- `uint32_t _ft_system_get_time_period (void)`
Get time period.
- `uint32_t _ft_system_get_time_offset (uint32_t event_stamp)`
Elapsed time based on the event stamp.
- `uint32_t _ft_system_get_time_offset_from_period (uint32_t event_period)`
Time offset by a defined period.
- `int32_t _ft_system_module_function (uint32_t option)`
Invoke the module function based on the option parameter.
- `int32_t _ft_system_control_function (uint32_t option)`
Invoke the control function based on the option parameter.
- `int32_t _ft_system_init (const struct ft_system *system)`
Initialize system.
- `void _ft_system_invoke_callback (uint32_t event)`
System callback invocation.
- `void _ft_system_modules_data_ready (void)`
*Function used internally to detect, whether new **Modules** data are available and to set the same flag for the controls. This function also invokes the control callbacks.*
- `struct ft_module * _ft_system_get_module (uint32_t interface_address, uint32_t instance)`
Find the n-th instance of a module of a specified type.
- `void ft_error (char *file_name, uint32_t line)`
The FT error function that is invoked from FT asserts.

8.2.4.2 Function Documentation

8.2.4.2.1 `int32_t _ft_system_control_function (uint32_t option)`

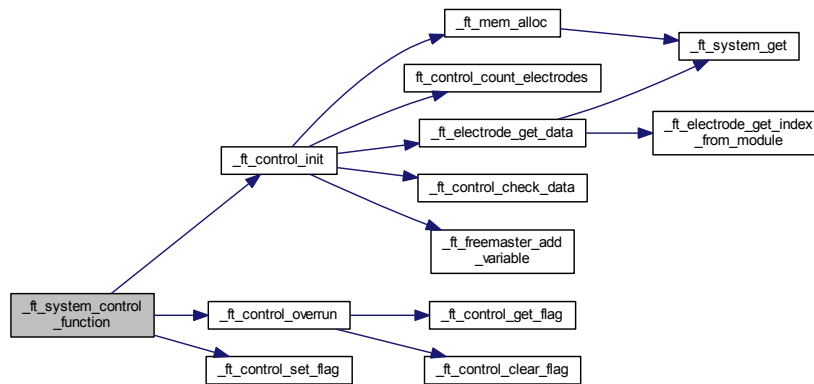
Parameters

<i>option</i>	One of the options from ft_system_control_call enum
---------------	---

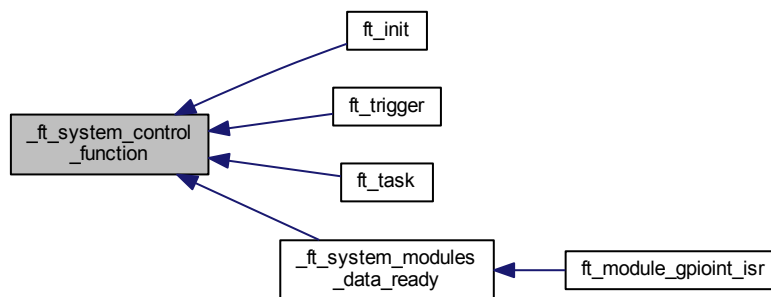
Returns

- FT_SUCCESS if the control's action was carried out successfully,
- FT_FAILURE if the control's action failed.

Here is the call graph for this function:



Here is the caller graph for this function:



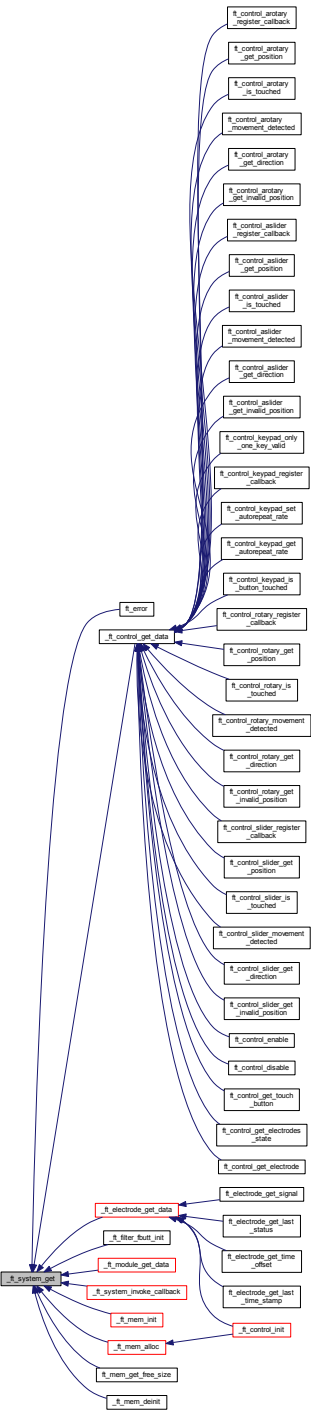
8.2.4.2.2 struct ft_kernel* _ft_system_get (void)

System

Returns

A pointer to the system kernel structure.

Here is the caller graph for this function:



8.2.4.2.3 `struct ft_module* _ft_system_get_module (uint32_t interface_address, uint32_t instance)`

System

Parameters

<i>interface_address</i>	Address to the module's interface (uniquely identifies module type).
<i>instance</i>	Zero-based module instance index.

Returns

- valid pointer to module.
- NULL if the module was not found

8.2.4.2.4 uint32_t ft_system_get_time_offset (uint32_t event_stamp)

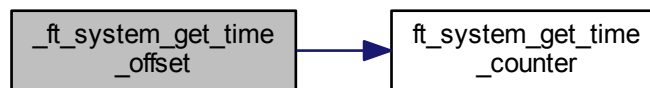
Parameters

<i>event_stamp</i>	Time stamp.
--------------------	-------------

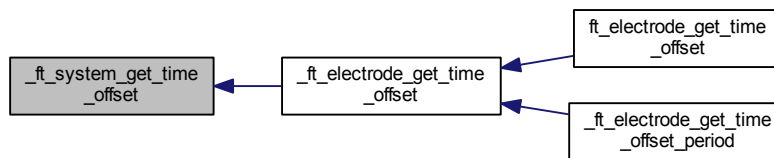
Returns

Elapsed time from the defined event stamp.

Here is the call graph for this function:



Here is the caller graph for this function:



8.2.4.2.5 `uint32_t ft_system_get_time_offset_from_period (uint32_t event_period)`

System

Parameters

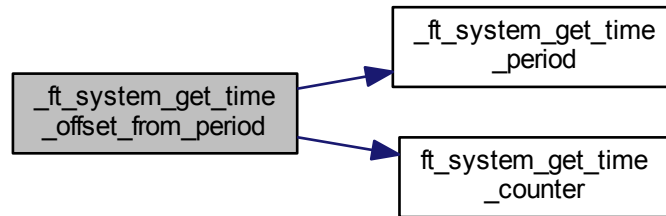
<i>event_period</i>	Defined event period.
---------------------	-----------------------

Returns

0 if event_period period modulo current counter is equal to 0 positive number otherwise.

This function is used to find out if an event can be invoked in its defined period.

Here is the call graph for this function:

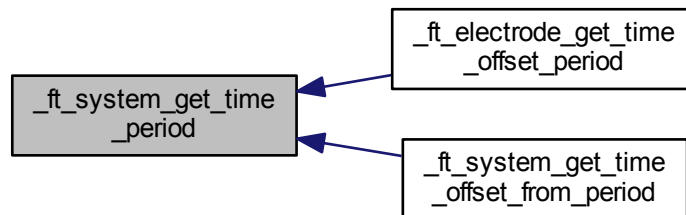


8.2.4.2.6 uint32_t _ft_system_get_time_period (void)

Returns

Time period.

Here is the caller graph for this function:

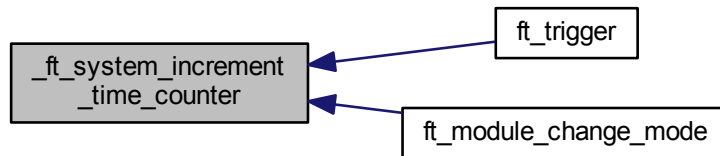


8.2.4.2.7 void _ft_system_increment_time_counter (void)

Returns

None.

Here is the caller graph for this function:



8.2.4.2.8 int32_t _ft_system_init (const struct ft_system * system)

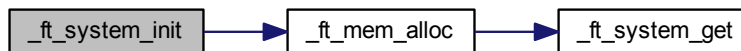
Parameters

<i>system</i>	Pointer to the user system parameters structure.
---------------	--

Returns

- FT_SUCCESS if the system data are set correctly,
- FT_FAILURE if the system data check failed.

Here is the call graph for this function:



System

Here is the caller graph for this function:



8.2.4.2.9 void _ft_system_invoke_callback (uint32_t event)

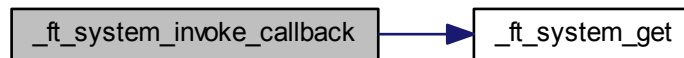
Parameters

<i>event</i>	Callback event.
--------------	-----------------

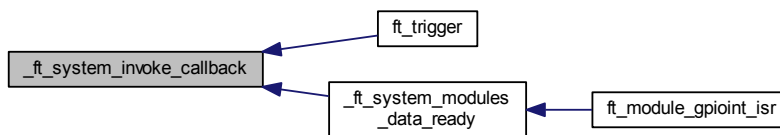
Returns

None.

Here is the call graph for this function:



Here is the caller graph for this function:



8.2.4.2.10 int32_t _ft_system_module_function (uint32_t *option*)

System

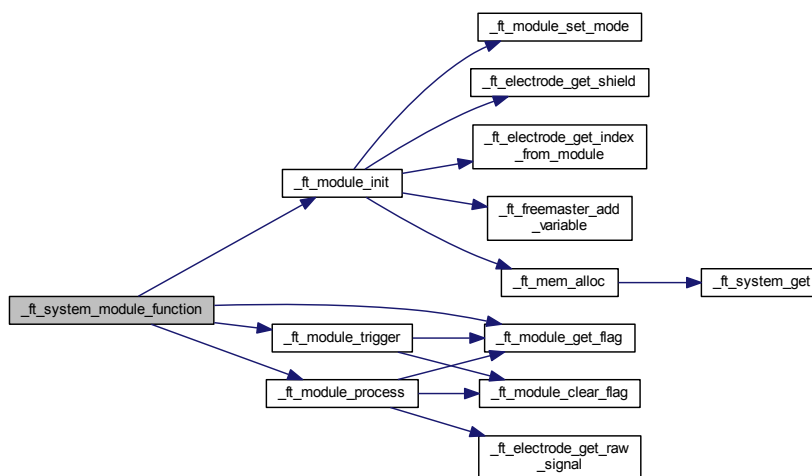
Parameters

<i>option</i>	One of the options from the <code>ft_system_module_call</code> enum
---------------	---

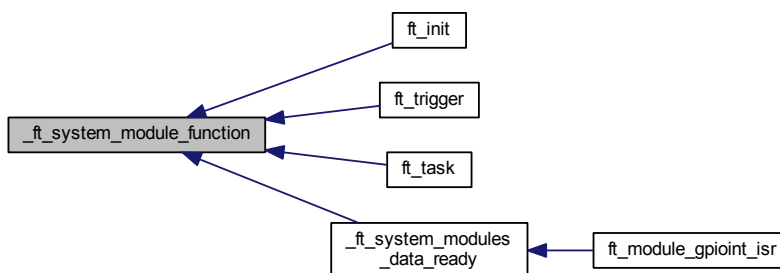
Returns

- FT_SUCCESS if the module's action was carried out successfully,
- FT_FAILURE if the module's action failed.

Here is the call graph for this function:



Here is the caller graph for this function:

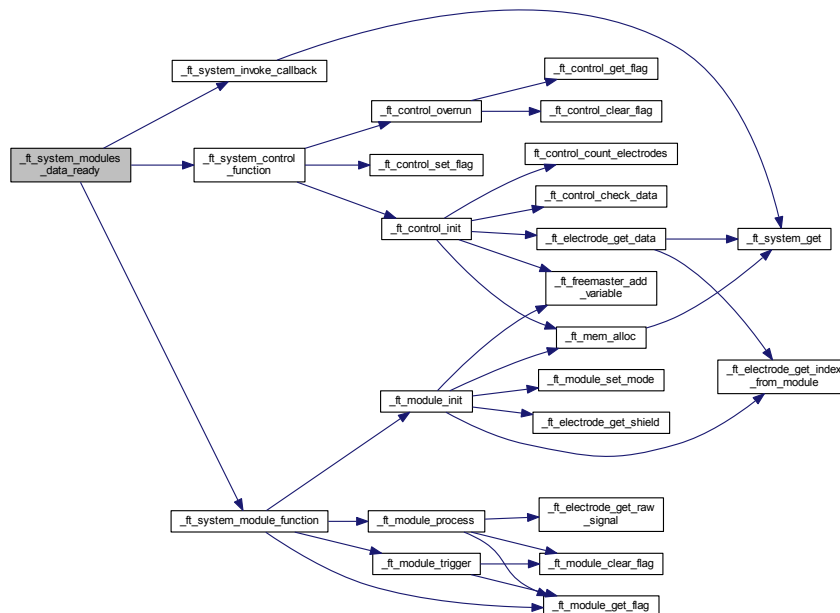


8.2.4.2.11 void _ft_system_modules_data_ready (void)

Returns

None.

Here is the call graph for this function:



Here is the caller graph for this function:



8.2.4.2.12 void ft_error (char * file_name, uint32_t line)

System

Parameters

<i>file_name</i>	Pointer to the file name.
<i>line</i>	Number of the line which was asserted.

Returns

none

Here is the call graph for this function:

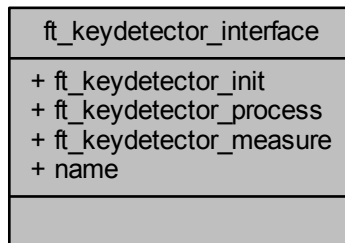


Chapter 9 Data Structure Documentation

9.0.5 ft_keydetector_interface Struct Reference

```
#include <ft_keydetectors.h>
```

Collaboration diagram for ft_keydetector_interface:



Data Fields

- `int32_t(* ft_keydetector_init)(struct ft_electrode_data *electrode)`
- `void(* ft_keydetector_process)(struct ft_electrode_data *electrode)`
- `void(* ft_keydetector_measure)(struct ft_electrode_data *electrode, uint32_t signal)`
- `const char * name`

9.0.5.1 Detailed Description

The key detector interface structure represents the Freescale Touch library Key Detector algorithm interface. The context data of the key detectors are stored in the [Electrodes](#) application objects.

9.0.5.2 Field Documentation

9.0.5.2.1 `int32_t(* ft_keydetector_interface::ft_keydetector_init)(struct ft_electrode_data *electrode)`

Key Detector initialization function pointer

9.0.5.2.2 void(* ft_keydetector_interface::ft_keydetector_measure)(struct ft_electrode_data *electrode, uint32_t signal)

Key Detector measure function pointer

9.0.5.2.3 void(* ft_keydetector_interface::ft_keydetector_process)(struct ft_electrode_data *electrode)

Key Detector process function pointer

9.0.5.2.4 const char* ft_keydetector_interface::name

A name of the variable of this type, used for FreeMASTER support purposes.



How to Reach Us:

Home Page:

freescale.com

Web Support:

freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address:

freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, Kinetis, Processor Expert are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Tower is a trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM, ARM Powered logo, and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2014 Freescale Semiconductor, Inc.

