NXP Touch Library Reference Manual

NXP Semiconductors, Inc.

Document Number: NT20RM
Rev 2.0
Apr 2020

Contents

Chapter 1
Introduction

Chapter 2
Release Notes

2.1 What is new **3**

2.2 Development tools **3**

2.3 Supported development systems **4**

2.4 NXP Touch release overview **4**

2.5 Known issues **6**

2.6 Revision history **6**

Chapter 3
Key Library Elements

3.1 System **11**

3.2 Modules **11**
3.2.1 Module Types 11

3.3 Key Detectors **11**
3.3.1 Types 12

3.4 Controls **14**

3.5 Electrodes **16**

3.6 Drivers **17**
3.6.1 TSI v4 17
3.6.2 TSI v5 18

3.6.3 CS 19

Chapter 4
Touch GUI Tool

4.1 Touch GUI Tool 21

Chapter 5
Configuring the Library

5.1 Configuration Example 27
5.1.1 Key Detectors 27
5.1.2 Electrodes 28
5.1.3 Modules 31
5.1.3.1 TSI module 31
5.1.3.2 TSI module 33
5.1.3.3 CS module 33
5.1.4 Controls 34
5.1.4.1 Keypad 34
5.1.4.2 Analog slider 34
5.1.4.3 Analog rotary 35
5.1.5 System 35
5.1.6 Low-Power mode 36
5.1.7 Proximity mode 36
5.1.8 System Callbacks 36

Chapter 6
NXP Touch library integration.

6.1 Creating Application with NXP Touch Library 37
6.1.1 Adding Library Files into the Project 37
6.1.2 Setting 'include' Search Paths 38
6.1.3 Setting the 'linker' path 42
6.1.4 Application Setup 45
6.1.5 The main() Function 50

Chapter 7
Example

7.1 FRDM touch-sensing Demo for FRDM-KE15Z and FRDM-KE16Z 59

7.2 FRDM touch-sensing Demo for FRDM-K32L2A4S 60

7.3 DK touch sensing Demo for QN908xCDK 60

Chapter 8
NXP Touch User API

8.1 Overview 63
8.1.1 Analog Rotary Control 66
8.1.1.1 Overview 66
8.1.1.2 Data Structure Documentation 67
8.1.1.2.1 struct nt_control_arotary 67
8.1.1.3 Macro Definition Documentation 68
8.1.1.3.1 NT_CONTROL_AROTARY_NAME 68
8.1.1.4 Typedef Documentation 68
8.1.1.4.1 nt_control_arotary_callback 68
8.1.1.5 Enumeration Type Documentation 68
8.1.1.5.1 nt_control_arotary_event 68
8.1.1.6 Variable Documentation 68
8.1.1.6.1 nt_control_arotary_interface 68
8.1.1.7 Analog Rotary Control API 69
8.1.1.7.1 Overview 69
8.1.1.7.2 Function Documentation 70
8.1.1.7.2.1 nt_control_arotary_get_direction 70
8.1.1.7.2.2 nt_control_arotary_get_invalid_position 71
8.1.1.7.2.3 nt_control_arotary_get_position 72
8.1.1.7.2.4 nt_control_arotary_is_touched 72
8.1.1.7.2.5 nt_control_arotary_movement_detected 73
8.1.1.7.2.6 nt_control_arotary_register_callback 74
8.1.2 Analog Slider Control 76
8.1.2.1 Overview 76
8.1.2.2 Data Structure Documentation 77
8.1.2.2.1 struct nt_control_aslider 77
8.1.2.3 Macro Definition Documentation 78
8.1.2.3.1 NT_CONTROL_ASLIDER_NAME 78
8.1.2.4 Typedef Documentation 78
8.1.2.4.1 nt_control_aslider_callback 78
8.1.2.5 Enumeration Type Documentation 78

8.1.2.5.1 nt_control_aslider_event . . . 78
8.1.2.6 Variable Documentation . . . 78
8.1.2.6.1 nt_control_aslider_interface . . . 78
8.1.2.7 Analog Slider Control API . . . 79
8.1.2.7.1 Overview . . . 79
8.1.2.7.2 Function Documentation . . . 80
8.1.2.7.2.1 nt_control_aslider_get_direction . . . 80
8.1.2.7.2.2 nt_control_aslider_get_invalid_position . . . 81
8.1.2.7.2.3 nt_control_aslider_get_position . . . 82
8.1.2.7.2.4 nt_control_aslider_is_touched . . . 82
8.1.2.7.2.5 nt_control_aslider_movement_detected . . . 83
8.1.2.7.2.6 nt_control_aslider_register_callback . . . 84
8.1.3 Keypad Control . . . 86
8.1.3.1 Overview . . . 86
8.1.3.2 Data Structure Documentation . . . 88
8.1.3.2.1 struct nt_control_keypad . . . 88
8.1.3.3 Macro Definition Documentation . . . 89
8.1.3.3.1 NT_CONTROL_KEYPAD_NAME . . . 89
8.1.3.4 Typedef Documentation . . . 89
8.1.3.4.1 nt_control_keypad_callback . . . 89
8.1.3.5 Enumeration Type Documentation . . . 89
8.1.3.5.1 nt_control_keypad_event . . . 89
8.1.3.6 Variable Documentation . . . 89
8.1.3.6.1 nt_control_keypad_interface . . . 89
8.1.3.7 Keypad Control API . . . 90
8.1.3.7.1 Overview . . . 90
8.1.3.7.2 Function Documentation . . . 91
8.1.3.7.2.1 nt_control_keypad_get_autorepeat_rate . . . 91
8.1.3.7.2.2 nt_control_keypad_is_button_touched . . . 92
8.1.3.7.2.3 nt_control_keypad_only_one_key_valid . . . 93
8.1.3.7.2.4 nt_control_keypad_register_callback . . . 93
8.1.3.7.2.5 nt_control_keypad_set_autorepeat_rate . . . 94
8.1.4 Matrix Control . . . 96
8.1.4.1 Overview . . . 96
8.1.4.2 Data Structure Documentation . . . 97
8.1.4.2.1 struct nt_coordinates8_xy . . . 97
8.1.4.2.2 struct nt_coordinates64_xy . . . 98
8.1.4.2.3 struct nt_control_matrix . . . 98
8.1.4.3 Macro Definition Documentation . . . 100
8.1.4.3.1 NT_CONTROL_MATRIX_NAME . . . 100
8.1.4.4 Typedef Documentation . . . 100
8.1.4.4.1 nt_control_matrix_callback . . . 100
8.1.4.4.2 nt_points64 . . . 100
8.1.4.4.3 nt_points8 . . . 100
8.1.4.5 Enumeration Type Documentation . . . 100

Section number	Title	Page

8.1.4.5.1 nt_control_matrix_control_gestures . . . 100
8.1.4.5.2 nt_control_matrix_event . . . 101
8.1.4.6 Variable Documentation . . . 101
8.1.4.6.1 nt_control_matrix_interface . . . 101
8.1.4.7 Matrix Control API . . . 102
8.1.4.7.1 Overview . . . 102
8.1.4.7.2 Function Documentation . . . 103
8.1.4.7.2.1 nt_control_matrix_register_callback . . . 103
8.1.4.7.2.2 nt_control_matrix_set_gesture . . . 104
8.1.5 Proxi Control . . . 105
8.1.5.1 Overview . . . 105
8.1.5.2 Data Structure Documentation . . . 106
8.1.5.2.1 struct nt_control_proxi . . . 106
8.1.5.3 Macro Definition Documentation . . . 107
8.1.5.3.1 NT_CONTROL_PROXI_NAME . . . 107
8.1.5.4 Typedef Documentation . . . 107
8.1.5.4.1 nt_control_proxi_callback . . . 107
8.1.5.5 Enumeration Type Documentation . . . 108
8.1.5.5.1 nt_control_proxi_event . . . 108
8.1.5.6 Variable Documentation . . . 108
8.1.5.6.1 nt_control_proxi_interface . . . 108
8.1.5.7 Proxi Control API . . . 109
8.1.5.7.1 Overview . . . 109
8.1.5.7.2 Function Documentation . . . 110
8.1.5.7.2.1 nt_control_proxi_register_callback . . . 110
8.1.5.7.2.2 nt_control_proxi_set_gesture . . . 111
8.1.6 Rotary Control . . . 113
8.1.6.1 Overview . . . 113
8.1.6.2 Macro Definition Documentation . . . 114
8.1.6.2.1 NT_CONTROL_ROTARY_NAME . . . 114
8.1.6.3 Typedef Documentation . . . 114
8.1.6.3.1 nt_control_rotary_callback . . . 114
8.1.6.4 Enumeration Type Documentation . . . 114
8.1.6.4.1 nt_control_rotary_event . . . 114
8.1.6.5 Variable Documentation . . . 115
8.1.6.5.1 nt_control_rotary_interface . . . 115
8.1.6.6 Rotary Control API . . . 116
8.1.6.6.1 Overview . . . 116
8.1.6.6.2 Function Documentation . . . 116
8.1.6.6.2.1 nt_control_rotary_get_direction . . . 116
8.1.6.6.2.2 nt_control_rotary_get_invalid_position . . . 117
8.1.6.6.2.3 nt_control_rotary_get_position . . . 118
8.1.6.6.2.4 nt_control_rotary_is_touched . . . 118
8.1.6.6.2.5 nt_control_rotary_movement_detected . . . 119
8.1.6.6.2.6 nt_control_rotary_register_callback . . . 120

8.1.7 Slider control . . . 122
8.1.7.1 Overview . . . 122
8.1.7.2 Macro Definition Documentation . . . 123
8.1.7.2.1 NT_CONTROL_SLIDER_NAME . . . 123
8.1.7.3 Typedef Documentation . . . 123
8.1.7.3.1 nt_control_slider_callback . . . 123
8.1.7.4 Enumeration Type Documentation . . . 123
8.1.7.4.1 nt_control_slider_event . . . 123
8.1.7.5 Variable Documentation . . . 124
8.1.7.5.1 nt_control_slider_interface . . . 124
8.1.7.6 Slider Control API . . . 125
8.1.7.6.1 Overview . . . 125
8.1.7.6.2 Function Documentation . . . 125
8.1.7.6.2.1 nt_control_slider_get_direction . . . 125
8.1.7.6.2.2 nt_control_slider_get_invalid_position . . . 126
8.1.7.6.2.3 nt_control_slider_get_position . . . 127
8.1.7.6.2.4 nt_control_slider_is_touched . . . 127
8.1.7.6.2.5 nt_control_slider_movement_detected . . . 128
8.1.7.6.2.6 nt_control_slider_register_callback . . . 129

8.2 Controls . . . 131
8.2.1 Overview . . . 131
8.2.2 General API . . . 132
8.2.2.1 Overview . . . 132
8.2.2.2 Data Structure Documentation . . . 132
8.2.2.2.1 union nt_control_params . . . 132
8.2.2.2.2 struct nt_control . . . 133
8.2.2.3 API Functions . . . 135
8.2.2.3.1 Overview . . . 135
8.2.2.3.2 Function Documentation . . . 135
8.2.2.3.2.1 nt_control_count_electrodes . . . 135
8.2.2.3.2.2 nt_control_disable . . . 135
8.2.2.3.2.3 nt_control_enable . . . 136
8.2.2.3.2.4 nt_control_get_electrode . . . 137
8.2.2.3.2.5 nt_control_get_electrodes_state . . . 137
8.2.2.3.2.6 nt_control_get_touch_button . . . 138

8.3 Electrodes . . . 140
8.3.1 Overview . . . 140
8.3.2 Data Structure Documentation . . . 141
8.3.2.1 struct nt_electrode_status . . . 141
8.3.2.2 struct gpio_inputs . . . 142
8.3.2.3 struct nt_electrode . . . 142
8.3.3 Macro Definition Documentation . . . 144
8.3.3.1 NT_ELECTRODE_SHIELD_ATTN . . . 144

8.3.3.2 NT_ELECTRODE_SHIELD_GAIN . . . 144
8.3.3.3 NT_ELECTRODE_SHIELD_SENS . . . 144
8.3.3.4 NT_ELECTRODE_SHIELD_THRESHOLD . . . 144
8.3.3.5 NT_ELECTRODE_STATUS_HISTORY_COUNT . . . 144
8.3.4 Enumeration Type Documentation . . . 145
8.3.4.1 nt_electrode_state . . . 145
8.3.5 API Functions . . . 146
8.3.5.1 Overview . . . 146
8.3.5.2 Function Documentation . . . 147
8.3.5.2.1 nt_electrode_disable . . . 147
8.3.5.2.2 nt_electrode_enable . . . 148
8.3.5.2.3 nt_electrode_get_data . . . 149
8.3.5.2.4 nt_electrode_get_last_status . . . 150
8.3.5.2.5 nt_electrode_get_last_time_stamp . . . 151
8.3.5.2.6 nt_electrode_get_module_data . . . 151
8.3.5.2.7 nt_electrode_get_raw_signal . . . 152
8.3.5.2.8 nt_electrode_get_signal . . . 153
8.3.5.2.9 nt_electrode_get_time_offset . . . 154
8.3.5.2.10 nt_electrode_is_enabled . . . 154
8.3.5.2.11 nt_electrode_reset . . . 155
8.3.5.2.12 nt_electrode_resume . . . 155
8.3.5.2.13 nt_electrode_suspend . . . 156

8.4 Filters . . . 158
8.4.1 Overview . . . 158
8.4.2 Data Structure Documentation . . . 158
8.4.2.1 struct nt_filter_fbutt . . . 158
8.4.2.2 struct nt_filter_iir . . . 159
8.4.2.3 struct nt_filter_dctracker . . . 159
8.4.2.4 struct nt_filter_moving_average . . . 160
8.4.3 Macro Definition Documentation . . . 160
8.4.3.1 NT_FILTER_MOVING_AVERAGE_MAX_ORDER . . . 160
8.4.4 Advanced Filtering and Integrating Detection . . . 161
8.4.4.1 Overview . . . 161
8.4.4.2 Data Structure Documentation . . . 161
8.4.4.2.1 struct nt_keydetector_afid_asc . . . 161
8.4.4.2.2 struct nt_keydetector_afid . . . 162
8.4.4.3 Macro Definition Documentation . . . 163
8.4.4.3.1 NT_KEYDETECTOR_AFID_ASC_DEFAULT . . . 163
8.4.4.4 Variable Documentation . . . 164
8.4.4.4.1 nt_keydetector_afid_interface . . . 164
8.4.5 Safa Detector . . . 165
8.4.5.1 Overview . . . 165
8.4.5.2 Data Structure Documentation . . . 165
8.4.5.2.1 struct nt_keydetector_safa . . . 165

8.4.5.3 Variable Documentation . . . 167
8.4.5.3.1 nt_keydetector_safa_default . . . 167
8.4.5.3.2 nt_keydetector_safa_interface . . . 167
8.4.6 Usafa Detector . . . 168
8.4.6.1 Overview . . . 168
8.4.6.2 Data Structure Documentation . . . 168
8.4.6.2.1 struct nt_keydetector_usafa . . . 168
8.4.6.3 Variable Documentation . . . 170
8.4.6.3.1 nt_keydetector_usafa_default . . . 170
8.4.6.3.2 nt_keydetector_usafa_interface . . . 171

8.5 Key Detectors . . . 172
8.5.1 Overview . . . 172
8.5.2 CS module . . . 173
8.5.2.1 Overview . . . 173
8.5.2.2 Data Structure Documentation . . . 174
8.5.2.2.1 struct nt_module_cs_params . . . 174
8.5.2.3 Macro Definition Documentation . . . 174
8.5.2.3.1 CUSTOM_DELAY . . . 174
8.5.2.3.2 NT_MODULE_CS_NAME . . . 174
8.5.2.4 Function Documentation . . . 174
8.5.2.4.1 delay . . . 174
8.5.2.5 Variable Documentation . . . 174
8.5.2.5.1 nt_module_cs_interface . . . 174
8.5.3 GPIO module . . . 175
8.5.3.1 Overview . . . 175
8.5.3.2 Data Structure Documentation . . . 175
8.5.3.2.1 struct nt_module_gpio_user_interface . . . 175
8.5.3.2.1.1 Field Documentation . . . 177
8.5.3.2.1.1.1 get_pin_value . . . 177
8.5.3.2.1.1.2 init_timer . . . 177
8.5.3.2.1.1.3 set_pin_default_state . . . 177
8.5.3.2.1.1.4 set_pin_gpio_mode . . . 177
8.5.3.2.1.1.5 set_pin_high . . . 177
8.5.3.2.1.1.6 set_pin_input . . . 177
8.5.3.2.1.1.7 set_pin_low . . . 177
8.5.3.2.1.1.8 set_pin_output . . . 177
8.5.3.2.1.1.9 set_pin_pull_down . . . 177
8.5.3.2.1.1.10 set_pin_pull_up . . . 177
8.5.3.2.1.1.11 set_pin_tsi_mode . . . 178
8.5.3.2.1.1.12 start_timer . . . 178
8.5.3.2.1.1.13 stop_timer . . . 178
8.5.3.2.1.1.14 timer_get_counter . . . 178
8.5.3.2.1.1.15 timer_get_overrun . . . 178
8.5.3.2.1.1.16 timer_reset_counter . . . 178

8.5.3.2.2 struct nt_module_gpio_params . . . 178
8.5.3.3 Macro Definition Documentation . . . 180
8.5.3.3.1 NT_MODULE_GPIO_NAME . . . 180
8.5.3.4 Function Documentation . . . 180
8.5.3.4.1 nt_gpio_data_check . . . 180
8.5.3.5 Variable Documentation . . . 180
8.5.3.5.1 nt_module_gpio_interface . . . 180
8.5.4 GPIO interrupt basic module . . . 181
8.5.4.1 Overview . . . 181
8.5.4.2 Data Structure Documentation . . . 182
8.5.4.2.1 struct nt_module_gpioint_user_interface . . . 182
8.5.4.2.1.1 Field Documentation . . . 183
8.5.4.2.1.1.1 clear_pin_interrupt . . . 183
8.5.4.2.1.1.2 init_pin . . . 183
8.5.4.2.1.1.3 init_timer . . . 183
8.5.4.2.1.1.4 set_pin_high . . . 183
8.5.4.2.1.1.5 set_pin_input . . . 183
8.5.4.2.1.1.6 set_pin_interrupt . . . 183
8.5.4.2.1.1.7 set_pin_low . . . 183
8.5.4.2.1.1.8 set_pin_output . . . 183
8.5.4.2.1.1.9 start_timer . . . 183
8.5.4.2.1.1.10 stop_timer . . . 183
8.5.4.2.1.1.11 timer_get_counter . . . 183
8.5.4.2.1.1.12 timer_reset_counter . . . 184
8.5.4.2.2 struct nt_module_gpioint_params . . . 184
8.5.4.3 Macro Definition Documentation . . . 185
8.5.4.3.1 NT_MODULE_GPIOINT_NAME . . . 185
8.5.4.4 Function Documentation . . . 185
8.5.4.4.1 nt_module_gpioint_isr . . . 185
8.5.4.4.2 nt_module_gpioint_overflow_isr . . . 186
8.5.4.5 Variable Documentation . . . 187
8.5.4.5.1 nt_module_gpioint_interface . . . 187
8.5.5 TSI module . . . 188
8.5.5.1 Overview . . . 188
8.5.5.2 Data Structure Documentation . . . 189
8.5.5.2.1 struct nt_module_tsi_noise . . . 189
8.5.5.2.2 struct nt_module_tsi_params . . . 190
8.5.5.3 Macro Definition Documentation . . . 192
8.5.5.3.1 NT_MODULE_TSI_NAME . . . 192
8.5.5.4 Function Documentation . . . 192
8.5.5.4.1 delay . . . 192
8.5.5.4.2 nt_module_TSI_Release . . . 192
8.5.5.4.3 nt_module_TSI_ReleaseAll . . . 192
8.5.5.4.4 nt_module_TSI_ReleaseOneElect . . . 193
8.5.5.4.5 nt_module_TSI_Touch . . . 194

8.5.5.4.6 nt_module_TSI_TouchAll . . . 194
8.5.5.4.7 nt_module_TSI_TouchOneElect . . . 195
8.5.5.5 Variable Documentation . . . 195
8.5.5.5.1 nt_module_tsi_interface . . . 195

8.6 Modules . . . 196
8.6.1 Overview . . . 196
8.6.2 General API . . . 197
8.6.2.1 Overview . . . 197
8.6.2.2 Data Structure Documentation . . . 198
8.6.2.2.1 union nt_module_params . . . 198
8.6.2.2.2 struct nt_module_safety_params . . . 199
8.6.2.2.3 struct nt_module . . . 201
8.6.2.3 Macro Definition Documentation . . . 202
8.6.2.3.1 NT_MODULE_SIGNAL_WTRMARK_HIGH . . . 202
8.6.2.3.2 NT_MODULE_SIGNAL_WTRMARK_LOW . . . 202
8.6.2.4 Enumeration Type Documentation . . . 202
8.6.2.4.1 nt_module_flags . . . 202
8.6.2.4.2 nt_module_mode . . . 203
8.6.2.5 API Functions . . . 204
8.6.2.5.1 Overview . . . 204
8.6.2.5.2 Function Documentation . . . 204
8.6.2.5.2.1 nt_module_change_mode . . . 204
8.6.2.5.2.2 nt_module_load_configuration . . . 205
8.6.2.5.2.3 nt_module_recalibrate . . . 206
8.6.2.5.2.4 nt_module_save_configuration . . . 207

8.7 Safety . . . 208
8.7.1 Overview . . . 208
8.7.2 Data Structure Documentation . . . 208
8.7.2.1 struct nt_module_safety_gpio_params . . . 208
8.7.3 Macro Definition Documentation . . . 210
8.7.3.1 NT_MODULE_SAFETY_NAME . . . 210
8.7.4 General API . . . 211
8.7.4.1 Overview . . . 211
8.7.4.2 Function Documentation . . . 211
8.7.4.2.1 _nt_module_safety_switch_electrode_gpio . . . 211
8.7.4.2.2 _nt_module_safety_switch_electrode_tsi . . . 212
8.7.4.2.3 nt_module_safety_check_baseline_high . . . 213
8.7.4.2.4 nt_module_safety_check_baseline_low . . . 214
8.7.4.2.5 nt_module_safety_check_electrode_short_adj . . . 214
8.7.4.2.6 nt_module_safety_check_electrode_short_gnd_read . . . 215
8.7.4.2.7 nt_module_safety_check_electrode_short_gnd_set . . . 215
8.7.4.2.8 nt_module_safety_check_electrode_short_vdd . . . 215
8.7.4.2.9 nt_module_safety_check_electrodes . . . 216

8.7.4.2.10 nt_module_safety_gpio_check_data . . . 217
8.7.4.3 Variable Documentation . . . 217
8.7.4.3.1 nt_safety_interface . . . 217

8.8 System . . . 218
8.8.1 Overview . . . 218
8.8.2 Data Structure Documentation . . . 219
8.8.2.1 union nt_system_event_context . . . 219
8.8.2.2 struct nt_system_event_context.sig_out_of_limit . . . 220
8.8.2.3 struct nt_system . . . 220
8.8.3 Typedef Documentation . . . 222
8.8.3.1 nt_error_callback . . . 222
8.8.3.2 nt_system_callback . . . 222
8.8.4 Enumeration Type Documentation . . . 222
8.8.4.1 nt_system_event . . . 222
8.8.5 API Functions . . . 224
8.8.5.1 Overview . . . 224
8.8.5.2 Function Documentation . . . 224
8.8.5.2.1 _nt_system_get_safety_period . . . 224
8.8.5.2.2 nt_error_register_callback . . . 225
8.8.5.2.3 nt_init . . . 225
8.8.5.2.4 nt_mem_get_free_size . . . 226
8.8.5.2.5 nt_system_get_time_counter . . . 227
8.8.5.2.6 nt_system_get_time_offset . . . 228
8.8.5.2.7 nt_system_register_callback . . . 228
8.8.5.2.8 nt_task . . . 229
8.8.5.2.9 nt_trigger . . . 230

8.9 General Types . . . 232
8.9.1 Overview . . . 232
8.9.2 Macro Definition Documentation . . . 233
8.9.2.1 NT_ASSERT . . . 233
8.9.2.2 NT_DEBUG . . . 233
8.9.2.3 NT_FLAGS_SPECIFIC_SHIFT . . . 233
8.9.2.4 NT_FLAGS_SYSTEM_SHIFT . . . 233
8.9.2.5 NT_FREEMASTER_SUPPORT . . . 233
8.9.2.6 NT_SAFETY_SUPPORT . . . 233
8.9.2.7 NULL . . . 233
8.9.3 Enumeration Type Documentation . . . 233
8.9.3.1 nt_result . . . 233
8.9.4 Analog Rotary Control . . . 234
8.9.4.1 Overview . . . 234
8.9.4.2 Data Structure Documentation . . . 235
8.9.4.2.1 struct nt_control_arotary_temp_data . . . 235
8.9.4.2.2 struct nt_control_arotary_data . . . 236

Section number	Title	Page

8.9.4.3 Macro Definition Documentation . . . 237
8.9.4.3.1 NT_AROTARY_INVALID_POSITION_VALUE . . . 237
8.9.4.4 Enumeration Type Documentation . . . 237
8.9.4.4.1 nt_control_arotary_flags . . . 237
8.9.5 Analog Slider Control . . . 238
8.9.5.1 Overview . . . 238
8.9.5.2 Data Structure Documentation . . . 239
8.9.5.2.1 struct nt_control_aslider_data . . . 239
8.9.5.2.2 struct nt_control_aslider_temp_data . . . 240
8.9.5.3 Macro Definition Documentation . . . 240
8.9.5.3.1 NT_ASLIDER_INVALID_POSITION_VALUE . . . 240
8.9.5.4 Enumeration Type Documentation . . . 241
8.9.5.4.1 nt_control_aslider_flags . . . 241
8.9.6 Keypad Control . . . 242
8.9.6.1 Overview . . . 242
8.9.6.2 Data Structure Documentation . . . 243
8.9.6.2.1 struct nt_control_keypad_data . . . 243
8.9.6.3 Enumeration Type Documentation . . . 245
8.9.6.3.1 nt_control_keypad_flags . . . 245
8.9.7 Matrix Control . . . 246
8.9.7.1 Overview . . . 246
8.9.7.2 Data Structure Documentation . . . 249
8.9.7.2.1 struct nt_control_matrix_data . . . 249
8.9.7.3 Macro Definition Documentation . . . 251
8.9.7.3.1 NT_MATRIX_UNLOCK_GESTURES . . . 251
8.9.7.4 Enumeration Type Documentation . . . 251
8.9.7.4.1 nt_control_matrix_flags . . . 251
8.9.8 Proxi Control . . . 253
8.9.8.1 Overview . . . 253
8.9.8.2 Data Structure Documentation . . . 254
8.9.8.2.1 struct nt_control_proxi_data . . . 254
8.9.8.3 Macro Definition Documentation . . . 256
8.9.8.3.1 NT_PROXI_ELECTRODES . . . 256
8.9.8.3.2 NT_PROXI_GESTURES . . . 256
8.9.8.4 Enumeration Type Documentation . . . 256
8.9.8.4.1 nt_control_prox_flags . . . 256
8.9.8.5 Function Documentation . . . 257
8.9.8.5.1 _nt_control_proxi_clear_flag . . . 257
8.9.8.5.2 _nt_control_proxi_get_flag . . . 257
8.9.8.5.3 _nt_control_proxi_set_flag . . . 257
8.9.9 Rotary Control . . . 258
8.9.9.1 Overview . . . 258
8.9.9.2 Data Structure Documentation . . . 259
8.9.9.2.1 struct nt_control_rotary_data . . . 259
8.9.9.3 Enumeration Type Documentation . . . 260

8.9.9.3.1 nt_control_rotary_flags . . . 260
8.9.10 Slider Control . . . 261
8.9.10.1 Overview . . . 261
8.9.10.2 Data Structure Documentation . . . 262
8.9.10.2.1 struct nt_control_slider_data . . . 262
8.9.10.3 Enumeration Type Documentation . . . 263
8.9.10.3.1 nt_control_slider_flags . . . 263

8.10 Controls . . . 264
8.10.1 Overview . . . 264
8.10.2 General API . . . 265
8.10.2.1 Overview . . . 265
8.10.2.2 Data Structure Documentation . . . 265
8.10.2.2.1 union nt_control_special_data . . . 265
8.10.2.2.2 struct nt_control_data . . . 267
8.10.2.2.3 struct nt_control_interface . . . 268
8.10.2.2.3.1 Field Documentation . . . 269
8.10.2.2.3.1.1 init . . . 269
8.10.2.2.3.1.2 name . . . 269
8.10.2.2.3.1.3 params_size . . . 269
8.10.2.2.3.1.4 process . . . 269
8.10.2.3 Enumeration Type Documentation . . . 269
8.10.2.3.1 nt_control_flags . . . 269
8.10.2.4 API Functions . . . 271
8.10.2.4.1 Overview . . . 271
8.10.2.4.2 Function Documentation . . . 271
8.10.2.4.2.1 _nt_control_check_data . . . 271
8.10.2.4.2.2 _nt_control_check_edge_electrodes . . . 272
8.10.2.4.2.3 _nt_control_check_neighbours_electrodes . . . 272
8.10.2.4.2.4 _nt_control_clear_flag . . . 272
8.10.2.4.2.5 _nt_control_clear_flag_all_elec . . . 273
8.10.2.4.2.6 _nt_control_get_data . . . 273
8.10.2.4.2.7 _nt_control_get_electrode . . . 275
8.10.2.4.2.8 _nt_control_get_electrodes_digital_state . . . 275
8.10.2.4.2.9 _nt_control_get_electrodes_state . . . 276
8.10.2.4.2.10 _nt_control_get_first_elec_touched . . . 278
8.10.2.4.2.11 _nt_control_get_flag . . . 278
8.10.2.4.2.12 _nt_control_get_last_elec_touched . . . 278
8.10.2.4.2.13 _nt_control_get_touch_count . . . 278
8.10.2.4.2.14 _nt_control_init . . . 278
8.10.2.4.2.15 _nt_control_overrun . . . 280
8.10.2.4.2.16 _nt_control_set_flag . . . 280
8.10.2.4.2.17 _nt_control_set_flag_all_elec . . . 281

8.11 CS Drivers . . . 282

8.11.1 Overview . . . 282
8.11.2 Data Structure Documentation . . . 283
8.11.2.1 struct nt_cs_user_config_t . . . 283
8.11.2.2 struct nt_cs_operation_mode_t . . . 284
8.11.2.3 struct nt_cs_state_t . . . 284
8.11.3 Typedef Documentation . . . 286
8.11.3.1 cs_callback_t . . . 286
8.11.4 Enumeration Type Documentation . . . 287
8.11.4.1 cs_status_t . . . 287
8.11.4.2 nt_cs_modes_t . . . 287
8.11.5 Variable Documentation . . . 288
8.11.5.1 g_csBase . . . 288
8.11.5.2 g_csIrqId . . . 288
8.11.5.3 g_csStatePtr . . . 288
8.11.6 API Functions . . . 289
8.11.6.1 Overview . . . 289
8.11.6.2 Function Documentation . . . 290
8.11.6.2.1 NT_CS_DRV_ChangeMode . . . 290
8.11.6.2.2 NT_CS_DRV_DeInit . . . 290
8.11.6.2.3 NT_CS_DRV_DisableLowPower . . . 291
8.11.6.2.4 NT_CS_DRV_EnableElectrode . . . 291
8.11.6.2.5 NT_CS_DRV_EnableLowPower . . . 292
8.11.6.2.6 NT_CS_DRV_GetCounter . . . 292
8.11.6.2.7 NT_CS_DRV_GetEnabledElectrodes . . . 293
8.11.6.2.8 NT_CS_DRV_GetMode . . . 293
8.11.6.2.9 NT_CS_DRV_GetStatus . . . 294
8.11.6.2.10 NT_CS_DRV_Init . . . 294
8.11.6.2.11 NT_CS_DRV_LoadConfiguration . . . 295
8.11.6.2.12 NT_CS_DRV_Measure . . . 295
8.11.6.2.13 NT_CS_DRV_SaveConfiguration . . . 296
8.11.6.2.14 NT_CS_DRV_SetCallBackFunc . . . 296

8.12 TSI Drivers . . . 298
8.12.1 Overview . . . 298
8.12.2 Data Structure Documentation . . . 299
8.12.2.1 struct nt_tsi_user_config_t . . . 299
8.12.2.2 struct nt_tsi_operation_mode_t . . . 300
8.12.2.3 struct nt_tsi_state_t . . . 301
8.12.3 Macro Definition Documentation . . . 303
8.12.3.1 TF_TSI_TOTAL_CHANNEL_COUNT . . . 303
8.12.4 Typedef Documentation . . . 303
8.12.4.1 tsi_callback_t . . . 303
8.12.5 Enumeration Type Documentation . . . 303
8.12.5.1 nt_tsi_modes_t . . . 303
8.12.5.2 nt_tsi_op_status_t . . . 304

8.12.5.3 tsi_status_t . . . 304
8.12.6 Variable Documentation . . . 305
8.12.6.1 g_tsiBase . . . 305
8.12.6.2 g_tsiIrqId . . . 305
8.12.6.3 g_tsiStatePtr . . . 305
8.12.7 API Functions . . . 306
8.12.7.1 Overview . . . 306
8.12.7.2 Function Documentation . . . 307
8.12.7.2.1 NT_TSI_DRV_ChangeMode . . . 307
8.12.7.2.2 NT_TSI_DRV_DeInit . . . 307
8.12.7.2.3 NT_TSI_DRV_DisableLowPower . . . 308
8.12.7.2.4 NT_TSI_DRV_EnableElectrode . . . 308
8.12.7.2.5 NT_TSI_DRV_EnableLowPower . . . 309
8.12.7.2.6 NT_TSI_DRV_GetEnabledElectrodes . . . 309
8.12.7.2.7 NT_TSI_DRV_GetMode . . . 310
8.12.7.2.8 NT_TSI_DRV_GetStatus . . . 310
8.12.7.2.9 NT_TSI_DRV_Init . . . 311
8.12.7.2.10 NT_TSI_DRV_InitSpecific . . . 311
8.12.7.2.11 NT_TSI_DRV_LoadConfiguration . . . 312
8.12.7.2.12 NT_TSI_DRV_Measure . . . 312
8.12.7.2.13 NT_TSI_DRV_MeasureAllElect . . . 313
8.12.7.2.14 NT_TSI_DRV_MeasureMoreElect . . . 313
8.12.7.2.15 NT_TSI_DRV_MeasureOneElect . . . 314
8.12.7.2.16 NT_TSI_DRV_Recalibrate . . . 314
8.12.7.2.17 NT_TSI_DRV_Resume . . . 315
8.12.7.2.18 NT_TSI_DRV_SaveConfiguration . . . 316
8.12.7.2.19 NT_TSI_DRV_SetCallBackFunc . . . 316
8.12.7.2.20 NT_TSI_DRV_Suspend . . . 317

Chapter 9
NXP Touch Private API

9.1 Overview . . . 319

9.2 Electrodes . . . 321
9.2.1 Overview . . . 321
9.2.2 Data Structure Documentation . . . 322
9.2.2.1 union nt_electrode_special_data . . . 322
9.2.2.2 struct nt_electrode_data . . . 322
9.2.3 Enumeration Type Documentation . . . 324
9.2.3.1 nt_electrode_flags . . . 324
9.2.4 API Functions . . . 325
9.2.4.1 Overview . . . 325

9.2.4.2 Function Documentation . . . 326
9.2.4.2.1 _nt_electrode_clear_flag . . . 326
9.2.4.2.2 _nt_electrode_get_data . . . 326
9.2.4.2.3 _nt_electrode_get_delta . . . 327
9.2.4.2.4 _nt_electrode_get_flag . . . 328
9.2.4.2.5 _nt_electrode_get_index_from_module . . . 328
9.2.4.2.6 _nt_electrode_get_last_status . . . 329
9.2.4.2.7 _nt_electrode_get_last_time_stamp . . . 330
9.2.4.2.8 _nt_electrode_get_raw_signal . . . 331
9.2.4.2.9 _nt_electrode_get_shield . . . 332
9.2.4.2.10 _nt_electrode_get_signal . . . 332
9.2.4.2.11 _nt_electrode_get_status . . . 333
9.2.4.2.12 _nt_electrode_get_time_offset . . . 334
9.2.4.2.13 _nt_electrode_get_time_offset_period . . . 335
9.2.4.2.14 _nt_electrode_get_time_stamp . . . 336
9.2.4.2.15 _nt_electrode_init . . . 337
9.2.4.2.16 _nt_electrode_is_touched . . . 338
9.2.4.2.17 _nt_electrode_normalization_process . . . 339
9.2.4.2.18 _nt_electrode_set_flag . . . 340
9.2.4.2.19 _nt_electrode_set_raw_signal . . . 340
9.2.4.2.20 _nt_electrode_set_signal . . . 341
9.2.4.2.21 _nt_electrode_set_status . . . 342
9.2.4.2.22 _nt_electrode_shielding_process . . . 343

9.3 Filters . . . 345
9.3.1 Overview . . . 345
9.3.2 Data Structure Documentation . . . 345
9.3.2.1 struct nt_filter_fbutt_data . . . 345
9.3.2.2 struct nt_filter_moving_average_data . . . 346
9.3.3 Enumeration Type Documentation . . . 347
9.3.3.1 nt_filter_state . . . 347
9.3.4 API Functions . . . 348
9.3.4.1 Overview . . . 348
9.3.4.2 Function Documentation . . . 349
9.3.4.2.1 _nt_abs_int32 . . . 349
9.3.4.2.2 _nt_filter_abs . . . 350
9.3.4.2.3 _nt_filter_deadrange_p . . . 350
9.3.4.2.4 _nt_filter_deadrange_u . . . 350
9.3.4.2.5 _nt_filter_fbutt_init . . . 351
9.3.4.2.6 _nt_filter_fbutt_process . . . 352
9.3.4.2.7 _nt_filter_iir_process . . . 352
9.3.4.2.8 _nt_filter_is_deadrange_u . . . 353
9.3.4.2.9 _nt_filter_limit_u . . . 354
9.3.4.2.10 _nt_filter_moving_average_init . . . 355
9.3.4.2.11 _nt_filter_moving_average_process . . . 355

9.3.4.2.12 _nt_filter_pos . . . 356
9.3.4.2.13 _nt_filter_range_s . . . 357
9.3.5 Advanced Filtering and Integrating Detection . . . 358
9.3.5.1 Overview . . . 358
9.3.5.2 Data Structure Documentation . . . 358
9.3.5.2.1 struct nt_keydetector_afid_asc_data . . . 358
9.3.5.2.2 struct nt_keydetector_afid_data . . . 359
9.3.5.3 Macro Definition Documentation . . . 361
9.3.5.3.1 NT_KEYDETECTOR_AFID_INITIAL_INTEGRATOR_VALUE . . . 361
9.3.5.3.2 NT_KEYDETECTOR_AFID_INITIAL_RESET_RELEASE_COUNTE-R_VALUE . . . 361
9.3.5.3.3 NT_KEYDETECTOR_AFID_INITIAL_RESET_TOUCH_COUNTER_-VALUE . . . 361

9.4 Key Detectors . . . 362
9.4.1 Overview . . . 362
9.4.2 Data Structure Documentation . . . 362
9.4.2.1 union nt_keydetector_data . . . 362
9.4.3 Safa Detector . . . 364
9.4.3.1 Overview . . . 364
9.4.3.2 Data Structure Documentation . . . 364
9.4.3.2.1 struct nt_keydetector_safa_data . . . 364
9.4.4 uSafa Detector . . . 368
9.4.4.1 Overview . . . 368
9.4.4.2 Data Structure Documentation . . . 368
9.4.4.2.1 struct nt_keydetector_usafa_data . . . 368
9.4.5 CS module . . . 372
9.4.5.1 Overview . . . 372
9.4.5.2 Data Structure Documentation . . . 373
9.4.5.2.1 struct nt_module_cs_data . . . 373
9.4.6 GPIO module . . . 375
9.4.6.1 Overview . . . 375
9.4.6.2 Data Structure Documentation . . . 375
9.4.6.2.1 struct nt_module_gpio_data . . . 375
9.4.7 GPIO interrupt module . . . 377
9.4.7.1 Overview . . . 377
9.4.7.2 Data Structure Documentation . . . 377
9.4.7.2.1 struct nt_module_gpioint_data . . . 377
9.4.8 TSI module . . . 379
9.4.8.1 Overview . . . 379
9.4.8.2 Data Structure Documentation . . . 379
9.4.8.2.1 struct nt_module_tsi_noise_data . . . 379
9.4.8.2.2 struct nt_module_tsi_data . . . 381
9.4.8.3 Macro Definition Documentation . . . 382
9.4.8.3.1 NT_TSI_NOISE_INITIAL_TOUCH_THRESHOLD . . . 382

9.4.8.3.2 NT_TSI_NOISE_TOUCH_RANGE . . . 382
9.4.8.4 Enumeration Type Documentation . . . 382
9.4.8.4.1 nt_module_tsi_flags . . . 382

9.5 Modules . . . 383
9.5.1 Overview . . . 383
9.5.2 General API . . . 384
9.5.2.1 Overview . . . 384
9.5.2.2 Data Structure Documentation . . . 384
9.5.2.2.1 union nt_module_special_data . . . 384
9.5.2.2.2 struct nt_module_data . . . 386
9.5.2.2.3 struct nt_module_interface . . . 387
9.5.2.2.3.1 Field Documentation . . . 388
9.5.2.2.3.1.1 change_mode . . . 388
9.5.2.2.3.1.2 electrode_disable . . . 389
9.5.2.2.3.1.3 electrode_enable . . . 389
9.5.2.2.3.1.4 init . . . 389
9.5.2.2.3.1.5 load_configuration . . . 389
9.5.2.2.3.1.6 name . . . 389
9.5.2.2.3.1.7 params_size . . . 389
9.5.2.2.3.1.8 process . . . 389
9.5.2.2.3.1.9 recalibrate . . . 389
9.5.2.2.3.1.10 save_configuration . . . 389
9.5.2.2.3.1.11 trigger . . . 389
9.5.2.3 API functions . . . 390
9.5.2.3.1 Overview . . . 390
9.5.2.3.2 Function Documentation . . . 390
9.5.2.3.2.1 _nt_module_check_signal_levels . . . 390
9.5.2.3.2.2 _nt_module_clear_flag . . . 391
9.5.2.3.2.3 _nt_module_count_electrodes . . . 392
9.5.2.3.2.4 _nt_module_get_data . . . 392
9.5.2.3.2.5 _nt_module_get_electrodes_state . . . 393
9.5.2.3.2.6 _nt_module_get_flag . . . 393
9.5.2.3.2.7 _nt_module_get_instance . . . 394
9.5.2.3.2.8 _nt_module_get_mode . . . 394
9.5.2.3.2.9 _nt_module_init . . . 394
9.5.2.3.2.10 _nt_module_process . . . 395
9.5.2.3.2.11 _nt_module_process_safety . . . 396
9.5.2.3.2.12 _nt_module_set_flag . . . 397
9.5.2.3.2.13 _nt_module_set_mode . . . 398
9.5.2.3.2.14 _nt_module_trigger . . . 399

Chapter 10
General API

10.1 Overview . . . 401

10.2 Function Documentation . . . 401
10.2.1 nt_safety_calc_cfg_crc . . . 401
10.2.2 nt_safety_check_cfg_data . . . 402

10.3 SafetyGPIO . . . 404
10.3.1 Overview . . . 404
10.3.2 Function Documentation . . . 405
10.3.2.1 configure_gpio_touch_sensing_pins . . . 405
10.3.2.2 dis_pin_pull . . . 405
10.3.2.3 get_pin_value . . . 406
10.3.2.4 init_timer . . . 407
10.3.2.5 set_pin_default_state . . . 407
10.3.2.6 set_pin_gpio_mode . . . 408
10.3.2.7 set_pin_high . . . 409
10.3.2.8 set_pin_input . . . 409
10.3.2.9 set_pin_low . . . 410
10.3.2.10 set_pin_ouput . . . 410
10.3.2.11 set_pin_pull_down . . . 411
10.3.2.12 set_pin_pull_up . . . 412
10.3.2.13 set_pin_tsi_mode . . . 413
10.3.2.14 start_timer . . . 414
10.3.2.15 stop_timer . . . 414
10.3.2.16 timer_get_counter . . . 414
10.3.2.17 timer_get_overrun . . . 414
10.3.2.18 timer_reset_counter . . . 415

10.4 safety . . . 416
10.4.1 Overview . . . 416
10.4.2 General API . . . 417
10.4.2.1 Overview . . . 417
10.4.2.2 Data Structure Documentation . . . 417
10.4.2.2.1 struct nt_module_safety_interface . . . 417
10.4.2.2.1.1 Field Documentation . . . 418
10.4.2.2.1.1.1 init . . . 418
10.4.2.2.1.1.2 name . . . 418
10.4.2.2.1.1.3 params_size . . . 418
10.4.2.2.1.1.4 process . . . 418
10.4.2.2.2 struct nt_safety_data . . . 418
10.4.2.3 Function Documentation . . . 419

10.4.2.3.1 _nt_module_safety_init . . . 419
10.4.2.3.2 _nt_module_safety_process . . . 420

10.5 FreeMASTER support . . . 421
10.5.1 Overview . . . 421
10.5.2 API functions . . . 422
10.5.2.1 Overview . . . 422
10.5.2.2 Function Documentation . . . 422
10.5.2.2.1 _nt_freemaster_add_variable . . . 422
10.5.2.2.2 _nt_freemaster_init . . . 423

10.6 Memory Management . . . 424
10.6.1 Overview . . . 424
10.6.2 Data Structure Documentation . . . 424
10.6.2.1 struct nt_mem . . . 424
10.6.3 API functions . . . 426
10.6.3.1 Overview . . . 426
10.6.3.2 Function Documentation . . . 426
10.6.3.2.1 _nt_mem_alloc . . . 426
10.6.3.2.2 _nt_mem_deinit . . . 427
10.6.3.2.3 _nt_mem_init . . . 427

10.7 System . . . 429
10.7.1 Overview . . . 429
10.7.2 Data Structure Documentation . . . 430
10.7.2.1 struct nt_kernel . . . 430
10.7.3 Macro Definition Documentation . . . 431
10.7.3.1 OSA_WAIT_FOREVER . . . 431
10.7.4 Typedef Documentation . . . 431
10.7.4.1 nt_mutex_t . . . 431
10.7.5 Enumeration Type Documentation . . . 431
10.7.5.1 nt_osa_status_t . . . 431
10.7.5.2 nt_system_control_call . . . 431
10.7.5.3 nt_system_module_call . . . 432
10.7.6 API Functions . . . 433
10.7.6.1 Overview . . . 433
10.7.6.2 Function Documentation . . . 434
10.7.6.2.1 _nt_system_check_data . . . 434
10.7.6.2.2 _nt_system_control_function . . . 434
10.7.6.2.3 _nt_system_count_pointer_array . . . 435
10.7.6.2.4 _nt_system_get . . . 435
10.7.6.2.5 _nt_system_get_module . . . 437
10.7.6.2.6 _nt_system_get_time_offset_from_period . . . 438
10.7.6.2.7 _nt_system_get_time_period . . . 438
10.7.6.2.8 _nt_system_increment_time_counter . . . 439

10.7.6.2.9 _nt_system_init . . . 439
10.7.6.2.10 _nt_system_invoke_callback . . . 440
10.7.6.2.11 _nt_system_module_function . . . 441
10.7.6.2.12 _nt_system_modules_data_ready . . . 443
10.7.6.2.13 nt_error . . . 443
10.7.6.3 NT_OSA . . . 445
10.7.6.3.1 Overview . . . 445
10.7.6.3.2 Function Documentation . . . 445
10.7.6.3.2.1 NT_OSA_EnterCritical . . . 445
10.7.6.3.2.2 NT_OSA_ExitCritical . . . 445
10.7.6.3.2.3 NT_OSA_Init . . . 446
10.7.6.3.2.4 NT_OSA_MutexCreate . . . 446
10.7.6.3.2.5 NT_OSA_MutexLock . . . 446
10.7.6.3.2.6 NT_OSA_MutexUnlock . . . 446

Chapter 11
Data Structure Documentation

11.0.7 nt_keydetector_interface Struct Reference . . . 447
11.0.7.1 Detailed Description . . . 447
11.0.7.2 Field Documentation . . . 448
11.0.7.2.1 name . . . 448
11.0.7.2.2 nt_keydetector_enable . . . 448
11.0.7.2.3 nt_keydetector_init . . . 448
11.0.7.2.4 nt_keydetector_measure . . . 448
11.0.7.2.5 nt_keydetector_process . . . 448
11.0.7.2.6 nt_keydetector_reset . . . 448
11.0.7.2.7 params_size . . . 448

Chapter 1
Introduction

This document describes the NXP Touch software library for implementing the touch-sensing applications on NXP MCU platforms. The touch-sensing algorithms contained in the library utilize either the dedicated Touch-Sensing Interface (TSI) module available on most Kinetis MCUs or Capacitive Sense (CS) on Q-N devices or the generic-pin I/O module to detect finger touch, movement, or gestures. See the license agreement document for the terms and conditions under which you can use the software. **Thank you for choosing the NXP solution!**

Figure 1: Capacitive Touch Sensing principle

The touch-detection algorithms and operation principles are based on the NXP TSS library version 3.-1 released in 2013, which is still available for download at www.nxp.com. However, the code in this NXP Touch library was completely rewritten to simplify the application coding and make better use of the 32-bit NXP MCU features. Unlike the TSS 3.1, the new NXP Touch (NT) library does not use pre-processor configuration macros. 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 C++ object-oriented applications.

The NXP Touch library is provided in a source-code form. When using the source-code form, 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.

The application code uses the specific 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

Find more details about the use of the NXP Touch software library in the following sections:

- Key Library Elements - Describes the main building blocks of the NXP Touch library.
- NXP Touch library integration. - Helps you to create your first NXP Touch application.
- Touch GUI Tool - Show the Touch GUI Tool enables you to generate the static configuration and online parameters tuning.
- Configuring the Library - Describes the configuration structures that must be defined in the application (or generated from the Touch GUI Tool).
- NXP Touch User API - Reference manual describing the structures and functions for you to use when building touch-sensing applications. - further divided into subsections:
  - System - The system contains general functionality for the whole NT and the basic NT API user interface.
  - Modules - The hardware interaction system that securely gets the raw data from the hardware and handles it to the electrode structure over the key detectors.
  - Key Detectors - The algorithms that secure the recognition of touch-sensing events (Touch and Release) in the input raw data from the modules.
  - Electrodes - The basic data container for the individual electrode data, and some basic Electrode API.
  - Controls - The high-level layer of NT that represents the captured data into the logical controls, such as keypad, slider, and other.
  - TSI Drivers or CS Drivers - The common and TSI version-specific low-level drivers that directly control the TSI peripheral.

Chapter 2
Release Notes

NXP Touch v2.5

Overview

These are the release notes for the NXP Touch software (NT) version 2.5. This is a full-function library to handle the touch events measured by the NXP TSI or CS module or the GPIO methods. It also has the capability to represent the measured results in various controls, such as Keypad, Rotary, Slider, Analog Slider, Analog Rotary, Proxi or Matrix.

2.1 What is new

This is the first public release, and it has these features:

- Supported devices
  - MKE16Z4 (TSI v5)
  - MKE157Z (TSI v5)
  - MK32L2A4S (TSI v4)
  - QN9080 (CS)
- Supported modules (peripherals/methods)
  - TSI v4
  - TSI v5
  - CS
- Supported controls
  - Slider
  - Rotary
  - Analog Slider
  - Analog Rotary
  - Proxi
  - Matrix
- Supported Development tools
  - IAR
  - Keil MDK
  - MCUXpresso IDE

2.2 Development tools

NXP Touch 2.5 was compiled and tested using these development tools:

- IAR embedded Workbench 8.50.1
- Keil MDK 5.29
- MCUXpresso IDE 11.2.0

- ARMGCC: 9-2019q4

2.3 Supported development systems

This release supports these boards and devices:

- FRDM-MKE16Z4 + FRDM-TOUCH
- FRDM-KE15Z + FRDM-TOUCH
- FRDM-MK32L2A4S
- DK-QN9080 + Touch Sense Board V1

NOTE: Because of the silicon errata on the initial silicon revisions, some features (such as the low-power mode wakeup) may not work properly.

2.4 NXP Touch release overview

NXP Touch (NT) is intended for use with the Kinetis MCU product family based on the ARM Cortex®-M series architectures. The whole library is designed to work with MCUXpresso SDK. The release consists of:

- Documentation - Reference manual
- Example application for the supported board
- FreeMASTER application for NT with Touch GUI tool
- NT library source code

Documentation

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

Example application for supported boards

There is a couple of example applications that run on standard NXP development boards and show all aspects of NT capabilities. All applications are suitable for all supported development tools.

NXP Touch GUI Tool in FreeMASTER

The NXP Touch GUI Tool (Touch GUI Tool) and the FreeMASTER application is provided for better understanding and tuning of the final configuration values. The FreeMASTER automatically generates the HTML pages for visualization touch quantities in real time according to the current configuration of the embedded application using TSA tables.

Figure 2.4.1: FreeMASTER example

NXP Touch library software

NT 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 into two main folders:

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

2.5 Known issues

Project paths

All the project paths are set up during the installation as an absolute path, relative to the MCUXpresso SDK installation. This enables you to have the project installed on the PC in the folder where the NT is installed, but it may cause issues on any other PCs.

2.6 Revision history

PBS**Revision number**	**Date**	**Substantial changes**
Rev 1.0	05/2015	Initial release
Rev 1.1	07/2015	SAFA keydec, LoPWR examples, Noise m. fix
Rev 2.0	04/2016	TSI v5, MCUXpresso SDK, Proximity control, distribution with MCUXpresso SDK, only for KE15Z
Rev 2.1	05/2017	uSAFA keydetector, Shielding, Noise level recovery, mutually-exclusive touch buttons, auto-tuning
Rev 2.2	11/2017	Matrix control + gestures, safety function, Touch GUI tool, proximity update
Rev 2.3	05/2018	CRC safety function, CS module (QN9080), graphic improvement of the Touch GUI tool, multitouch update
Rev 2.4	09/2018	FRDM-KE16Z support, auto TSI calibration, virtual touch/release function, Touch GUI tool improvements (adding safety, graphic bugfixes), fixed proxi movement and gesture recognition
Rev 2.4.1	1/2020	Add slider example for FRDM-MK32L2A4S (TSI v4) board
Rev 2.5	6/2020	Added TSIv5 feature to support specific HW configuration for each electrode and more HW configurations for same TSI channel, updated FreeMASTER to 3.0

Chapter 3
Key Library Elements

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

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

Figure 1: Object structure

- System (System API) - The general code of NT to provide a basic functionality for the user application.
- Modules (Modules API) - The modules are designed for hardware interaction and gathering of capacitance data.
- Key Detectors (Key Detectors API) - The key detectors are designed to recognize the touch/release events from the data measured by the modules.
- Controls (Controls API) - The high-level objects represent the the so-called "Decoder" and are used to create virtual devices (controls) from the individual electrodes.
- Electrodes (Electrodes API) - The electrode is a basic data container with the functionality for general control of the individual electrodes.
- Drivers (TSI Drivers API or CS Drivers API) - These drivers control the TSI periphery. There is a common part and a version-dependent part of the driver.

3.1 System

The main object encapsulating the lists of the other key objects used in the NXP 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 the initialization, global timing, and uniform access to the touch-detection modules.

3.2 Modules

This section describes the different Modules supported by the NXP Touch library.

The modules are a part of the NT code that secures the gathering of raw data from the NXP TSI or CS peripheral. The module describes the hardware configuration and control of the elementary functionality of the TSI peripheral by a generic, low-level driver API. The module also handle the noise_mode mode supported by TSI v4 (Kinetis K32L2A). All the different modules are implemented using the same API function. Users doesn't need to take care about the differences between individual TSI versions. Basically all modules behaves as TSI without any difference.

3.2.1 Module Types

The NT library defines **TSI module** or **CS module**. The module gathers the physical electrode capacitance data from the TSI peripheral. It is based on the MCUXpresso SDK 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_mode - Special noise mode is a hardware feature of the TSI module version 4. The module is implemented in the Kinetis KE32L2A MCU.)

3.3 Key Detectors

This section describes the different Key Detectors supported by the NXP Touch library. The key detector module determines whether an electrode was touched or released, according to 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.

3.3.1 Types

The NT library defines the objects of the keydetectors compound of the following division:

- **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 manually selected in the NT configuration. The key detector also provides automatic sensitivity calibration. The calibration periodically adjusts the level of the electrode sensitivity, which is calculated according to the touch-tracking information. Although the sensitivity must no longer be set manually, 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 the analog decoders and the proximity function. A debounce function is implemented in this module to eliminate the false detections caused by instantaneous noise.
  The main functions of the AFID key detector module are:
  - Two filters with the integration for touch detection
  - Electrode detection debouncing
  - Baseline generation
  - IIR filtering of the current capacitance signal
  - Proximity detection
  - Sensitivity autocalibration
  - Electrode status reporting
  - Fault reporting

Figure 3.3.1: AFID key detector signals

- **SAFA,uSAFA** - The SAFA (Signal Adaptive Filtering Algorithm) key detector operates by calculating three auxiliary signals (Adaptive Baseline, Predictive signal, and Noise signal). The raw measured signal is filtered using these signals. The SAFA key detector can handle signals that vary in level, polarity, and offset to zero. In the release state, there is the Adaptive Baseline signal following the Measured signal. There is the Dead Band proportional to the actual Noise signal and the signal-to-noise ratio around the Baseline signal. When the Measured signal leaves the Dead Band, the Touch event is detected. The Predicted signal is then calculated and almost reaches the Measured signal. When the Measured signal decreases under the Predicted signal, the Release event is detected. The SAFA key detector can be manually selected in the NT configuration. The SAFA keydetector is more recommended to pass the harsh EMC environment and EMC bench tests like Conducted Immunity test (IEC61000-4-6). The **uSAFA** is the "uni-directional" version of the SAFA keydetector, which, unlike the standard SAFA, evaluates just the positive delta signal changes as a valid touch event, while the negative signal drops are ignored. uSAFA is more recommended for usage with TSIv5, available on Kinetis KE1x family.
  The SAFA algorithm components are:
  - Adaptive Baseline calculation
  - Predictive signal calculation
  - Noise signal calculation
  - The Dead Band filter
  - Moving Average Filters

Figure 3.3.2: SAFA key detector signal diagram

3.4 Controls

This section describes the different Controls supported by the NXP Touch library.

The decoders provide the highest level of abstraction in the library. In this layer, the information about the touched and untouched electrodes is interpreted, and it shows the status of the control in a behavioral way. The decoders also provide additional functionalities. 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. The decoders can be described as classes of the object-oriented language, where each control has a decoder associated with 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:

- Keypad
- Analog rotary
- Analog slider

Types:

- Analog Slider Control The analog slider control works similarly to a standard slider, but with less electrodes. The calculated position also 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 must be linear. The figure shows an arrangement of electrodes used for a typical analog slider. The analog slider control provides these callback events:
  - direction change
  - movement
  - touch
  - release

Figure 3.4.1: Analog slider

- Keypad Control The keypad is a basic configuration for the arranged electrodes shown in the figure, because all that matters is to determine which electrode was touched. The Keypad Decoder is the module handling the boundary checking, controlling the events buffer, and reporting of events (depending on the user configuration). The keypad decoder must be used when the application needs the electrodes to behave like keyboard keys. If you need to detect movement, use another type of decoder. The keypad decoder is capable of using groups of electrodes that must be touched simultaneously to report the defined key. This enables you to create a control interface with more user inputs than the number of physical electrodes. The keypad enables to detect so called multi-touch event when more keys were touched simultaneously. The key combinations consist of two or more keys pressed together and can be defined by user for the group or single electrode based keypad. The actual touched keys are compared with stored key combinations from the last one to the first one. The detection of the sameness stops the comparison and execute the callback with the multi-touch combination number. The keypad control provides these callback events:
  - touch

  - release
  - autorepeat
  - multi-touch

Figure 3.4.2: Keypad

- Analog Rotary Control The analog rotary control is similar to the standard rotary control, but with less electrodes. The calculated position also has a higher resolution. For example, a 4-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 these callback events:
  - direction change
  - movement
  - touch
  - multi-touch
  - release

Figure 3.4.3: Analog rotary

- Matrix Control The matrix implements a similar functionality to the keypad but the matrix has the electrode surfaces placed closer to each other. Moreover, the matrix control allows processing the position tracking and gesture recognition. Usually, the matrix control is used for data processing from the touchpad hardware. The matrix control provides these callback events:
  - movement

- touch
- release
- double tap
- cover
- gesture

Figure 3.4.4: Matrix

- Proxi Control The proxi control enables the detection of a nger or object presence in the near eld of the electrode, so that the approaching nger/hand can be detected without a direct electrode touch. Moreover, the proximity position and direction of the movement can be evaluated for each proximity electrode. The proxi control provides these callback events:
- movement
- touch
- release
- gesture

Figure 3.4.5: Proxi

3.5 Electrodes

The electrodes are data objects that are used by data-acquisition algorithms to store per-electrode data, as well as the resulting signal and the touch/timestamp information. Each electrode provides a signal value information. The baseline value and the touch/timestamp buffer contain the time of the last few touch and release events. The electrode also contains information about the key detector used to detect touches

for this physical electrode. This brings the advantage that each electrode has its own settings of the key detector (independent of the module used). It contains information about the hardware pin, the immediate touch status, and the 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 NXP Touch library.

It is you as the application programmer and NXP Touch library user who specifies what Modules and Controls are going to be instantiated in the system and what electrodes are going to be serviced by each module. See more details about the various modules and their configuration in this document.

The information about the enabled electrodes and the TSI channel they are assigned to is newly stored in each electrode as a specific flag. This approach allows to access the same TSI channel by more electrodes (TSI v5 only) or use a different hardware TSI configuration for each electrode (TSI v5 only).

In TSI v4, there are 16 input channels that use the self-capacitance measurement only. In TSI v5, there are 25 input channels that use the self-capacitance measurement and some of these channels may be used for the mutual-capacitance measurement. The number of channels to be used for the mutual-capacitance sensing depends on the TSI v5 specification and it can change from device to device (ke15z has 4 RX and 4 TX channels, ke16z has 6 RX and 6 TX channels). There can be 16 or 36 mutual-capacitance electrodes plus the rest of the self-capacitance electrodes. The number of remaining electrodes is 16 or 12 self-capacitance sensing electrodes. The lower 25 channels describe the self-capacitance electrodes. In the ke16z device, the TSI channels from 0 to 5 can be configured as the tx electrodes during the mutual-capacitance measurements. The TSI channels from 6 to 11 can be configured as the RX electrodes during the mutual-capacitance measurements, so there can be up to 36 mutual-capacitance electrodes. The channels from 25 to 60 describe the mutual-capacitance electrodes (for example, channel 61 describes the mutual electrode formed by tx2 and rx10).

There may be a conflict when one channel is used for both the self-capacitance electrode and the RX/TX mutual-capacitance electrode. Such case causes the NT_ASSERT error. The NT_TSI_TRANSFORM_-MUTUAL macro can be used to easily define the mutual-capacitance electrodes.

3.6 Drivers

There are several versions of the TSI periphery. Each version of the TSI has different set of registers and different functionality.

3.6.1 TSI v4

TSI v4 module is a simplified version of the TSI v2 with some additional features. It is currently implemented in the KE32L2A4a ARM®Cortex®-M0+ MCU. 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.

3.6.2 TSI v5

The electrode capacitance change causes a voltage difference change at the input of the transconductance amplifier. The amplifier converts the voltage difference to a direct current. This current is integrated as voltage on the integration capacitor from Vm to Vp, creating a sawtooth signal. The period of the sawtooth signal depends on the electrode capacity. The TSI counter counts the TSI clock for one or several periods of the sawtooth signal. The TSI counter value then depends on the electrode capacity. TSI v5 supports self-capacitance measurements when only one TSI channel is used for one electrode. TSI v5 also adds the mutual-capacitance measurements where two electrodes (RX, TX) are used for one electrode. The TSI mutual channels can be combined into a matrix to increase the number of electrodes. The TSI v5 module measures just one electrode in one measurement cycle and enables the software or hardware trigger to start a scan.

The new software feature (implemented in the TSI v5 module only) develops the configuration capabilities by enabling more parameters to be defined for each electrode. The configuration structure of the electrode is expanded by the tsi_hw_config named item pointing to the tsi_config_t data type structure, containing all hardware setting for the TSI v5 module. The TSI v5 module is always reconfigured to measure the channel capacitance value according to the specific (or default) TSI hardware configuration. If different electrode dimensions are targeted, this feature enables to set the specific sensitivity, sample time, or other parameters given by the application without any limitation. The same TSI channel can be used for more electrodes with different hardware configurations, keydetectors, or the other parameters for tuning purposes. If the tsi_hw_config item is not defined for a specific electrode, the initialization process assigns the default TSI configuration (from the config member of the nt_module structure).

Figure 3.6.1: TSI peripheral version 5

3.6.3 CS

The capacitive sense (CS) determines proximity and touch detection based on self-capacitance. As a finger makes contact with a touch pad, the change in capacitance is measured between the input channel and ground. This is accomplished by utilizing a relaxation oscillator that increments a counter every cycle. When there is no contact on the touch pad, the relaxation oscillator will oscillate at a user-configurable frequency based on the parasitic capacitance of the system. When a finger makes contact, the total capacitance increases, causing the frequency of the oscillator to drop. It will cause the counter output to be noticeably lower than when there is no contact with the touch pad. Touch events are determined by observing the change in the counter output.

Figure 3.6.2: CS peripheral

Chapter 4
Touch GUI Tool

4.1 Touch GUI Tool

The NXP Touch GUI Tool enables you to generate static configuration and online parameters tuning for the NXP Touch Library easily using a user-friendly Graphic User Interface (GUI). The NXP Touch GUI Tool is integrated into the FreeMASTER software as the two bookmarks named ***TOUCH HW LAYER*** (support TSIv5 and CS module not TSIv4 module) and ***TOUCH SW LAYER***. This all-in-one solution enables you to read and watch the NXP Touch variables in real time (bookmarks: *MODULES*, *CONTROLS* and *ELECTRODES*) and to write (or modify) the variables as well. The NXP Touch GUI tool can generate static configuration files (used before the project is built) and dynamic tuning (or debugging) to modify the variables or parameters online while the MCU is running (requires enabling the FreeMASTER communication and placing the tuning parameters into the RAM).

Figure 4.1.1: NXP Touch GUI Tool - the HW configuration

The **Basic/Expert** switch hides the less important items (sets them to the default values) in the "Basic" mode or shows all items in the "Expert" mode (the "Expert" mode enables you to assign the same elec-

trode to more controls or configure the safety). The ***READ CONFIGURATION FROM BOARD*** button imports the configuration from the board to the NXP Touch GUI Tool (requires enabling the FreeMASTER communication). The most important NXP Touch GUI Tool feature is generating the NXP Touch Library configuration. The whole NXP Touch configuration is placed into the *nt_setup.c* and *nt_setup.h* files. The generated configuration files are located releatively in the *freemaster/src* directory. The *nt_setup.c* is shown in the NXP Touch GUI Tool *GENERATED CONFIGURATION* bookmark. The Touch GUI Tool enables you to generate both configuration files, but it cannot read the configuration back from the *nt_setup.c* and *nt_setup.h* files. Therefore, the tool can store and read the whole configuration to (or from) an .xml file located relatively in the *freemaster/src* directory as well. These are the recommended steps to use the tool:

- Save the .xml file name for the Touch Tool Configuration or load the .xml file configuration from another source (demo project).
- Set the *Basic* mode if you are a beginner or if you want to see only the important set of NXP Touch parameters. Otherwise, select the "Expert" mode.
- Set the Module Type (peripheral measuring the capacity).
- Create or change the configuration in the *TOUCH HW LAYER* and *TOUCH SW LAYER* bookmarks. Parse the *board.h* file from the MCUXpresso SDK to read the macros for the ***pin_input*** of each electrode or assign the self-cap or mutual-cap electrode numbers in the *TOUCH SW LAYER* bookmark. Check the ***Place configuration to RAM*** for the library layer determine for tuning and debugging in the *TOUCH SW LAYER* bookmark.
- Generate the *nt_setup.c* (including callbacks and other function templates as a comment) and *nt_setup.h* files.
- Compile the project and download the NXP Touch library code to the flash memory.
- Start the FreeMASTER communication (by clicking the red STOP button) to watch the variables and adjust the configuration parameters (only if the *Place configuration to RAM* checkbox is checked).

If the action after clicking the specific button is successful, a green checked icon is shown on the right-hand side of the button. In case of a failure, a red icon with a cross and a pop-up help with the error description appears (when you hover the mouse cursor above the icon).

Like the NXP Touch library, the NXP Touch GUI Tool is divided into these sections:

- System (*TOUCH SW LAYER* bookmark).
- Modules (*TOUCH HW LAYER* bookmark, software thresholds are in the *TOUCH SW LAYER* bookmark).
- Controls (*TOUCH SW LAYER* bookmark).
- Electrodes (*TOUCH SW LAYER* bookmark).

Figure 4.1.2: Control section configuration

Each section has its own name and other parameters corresponding to the NXP Touch Library parameters to be written, set, checked, or chosen from the list boxes. The specific configuration can be easily created or modified from the previously-defined configuration (stored as an .xml file). The NT Controls and NT Electrodes sections enable you to create more bookmarks corresponding to each next new electrode or control. To add a new electrode or control, click the plus button placed on the right-hand side of the control or electrode number (above the electrode or control bookmarks). In the "Basic" mode, when you assign an electrode to a specific control, the electrode is disabled for other controls (the "Expert" mode enables you to assign the same electrode to more controls or configure the safety). The electrode cannot be assigned to more controls and the electrode checkbox is disabled (the checkbox is grayed out). A similar behavior can be shown for the shielding electrode. The Touch GUI Tool can decode the pin setting from the board.h file or the self-cap or mutual-cap channel can be directly chosen from the channel list box. Each created electrode is assigned to the module by default. The control type and its parameters can by specified for each control. A short pop-up help is shown after you point to the parameter name. For each electrode, the key-detector and its own configuration parameters can be defined. To simplify the key-detector configuration, the key-detector setting can be shared (copied) among the electrodes. The real electrode data can be shown by clicking the ***Show*** button in the electrode section enabling to see the real touch or release electrode event when the keydetector is tuned. The waveform to be affected by the parameter modification is written in the parameter pop-up help. Each created control or electrode bookmark can be removed by clicking the close cross placed on the right-hand side of the bookmark.

Figure 4.1.3: Electrode section configuration

The NXP Touch GUI Tool provides the online tuning and debugging feature. Each section has its own checkbox named ***Place configuration to RAM***, modifying the structure type and removing the ***const*** keyword which moves the structure to the read/write memory to enable its modification for tuning and debugging purposes. After flashing such configuration, the GUI tool can write a specific parameter after you click the ***Write*** button. The ***Read*** button reads a value from the RAM (or flash) memory. The status of the writing or reading is shown as an icon after clicking the button and the pop-up help is displayed after you hover the mouse cursor above the icon.

The next MODULES, CONTROLS, and ELECTRODES bookmarks visualize the NXP touch quantities (live data) from defined electrodes and controls in real time and show the electrode assignment to modules or controls. The MODULES, CONTROLS, and ELECTRODES graphical objects are created according to the actual configuration loaded from the board. The NXP Touch quantities can be visualized as a

waveform. After clicking the links from the MODULES or CONTROLS page, the FreeMASTER scope window scope window opens. The FreeMASTER "Variable Watch" window shows the values of the NXP touch quantities by adding the quantities to the "Watch" properties.

TOUCH HW LAYERS | TOUCH SW LAYERS | MODULES | CONTROLS | ELECTRODES | DEBUG | GENERATED CONFIGURATION

tsi: module_tsi_0

detected

Electrode name	Keydetector	Raw Cnt	Baseline Cnt	Signal	Touch	Flags
electrode_usafa_0, scope	usafa	59300	59298	59295		16
electrode_usafa_1, scope	usafa	60141	60143	60141		16
electrode_usafa_2, scope	usafa	10650	10628	10647		16
electrode_usafa_3, scope	usafa	10736	10715	10739		16
electrode_usafa_4, scope	usafa	14570	14519	14566		16
electrode_usafa_5, scope	usafa	13400	13342	13392		16
electrode_usafa_6, scope	usafa	55442	55383	55443		16
electrode_usafa_7, scope	usafa	52782	52740	52783		16
electrode_usafa_8, scope	usafa	49798	48832	49794		19
electrode_usafa_9, scope	usafa	53538	52578	53545		19
electrode_usafa_10, scope	usafa	53588	53448	53589		19
electrode_usafa_11, scope	usafa	54407	54289	54408		19

Figure 4.1.4: Module visualization

The FreeMASTER "Scope" and "Variables" windows visualize the waveforms and quantities in real time from the NXP Touch Library concurrently with the NXP Touch GUI Tool running - see the following figure.

Figure 4.1.5: NXP Touch GUI Tool - layout with scope

Chapter 5
Configuring the Library

This section describes the library structures that must be initialized to use the NXP Touch library in an 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 the library initialization are then used in the application code to access their run-time properties.

For example: the user initializes the nt_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 run-time, the code uses a pointer to the same structure to access the calculated values (such as the finger position).

5.1 Configuration Example

The next sections describe the setup for each library structure.

5.1.1 Key Detectors

Key Detector setup for uSAFA. The following setup is common in nt_setup.c

```
/* uSAFA key detector setting for keypad control build from FRDM KE15Z self-cap electrodes */
const struct nt_keydetector_usafa nt_keydetector_usafa_El1 = {
  .signal_filter.coef1 = 2,
  .base_avrg.n2_order = 12,
  .non_activity_avrg.n2_order = 15,
  .entry_event_cnt = 4,
  .deadband_cnt = 4,
  .signal_to_noise_ratio = 6,
  .min_noise_limit = 100,
  .dc_track_enabled = 1,
  .dc_track_cnt = 100,
};
/* uSAFA key detector settings for keypad control build from FRDM-TOUCH mutual-cap electrodes */
const struct nt_keydetector_usafa nt_keydetector_usafa_El3 = {
  .signal_filter.coef1 = 2,
  .base_avrg.n2_order = 12,
  .non_activity_avrg.n2_order = 15,
  .entry_event_cnt = 4,
  .deadband_cnt = 5,
  .signal_to_noise_ratio = 4,
  .min_noise_limit = 60,
  .dc_track_enabled = 1,
  .dc_track_cnt = 100,
};
/* uSAFA key detector settings for slider and rotary control build from FRDM-TOUCH self-cap electrodes */
const struct nt_keydetector_usafa nt_keydetector_usafa_El7 = {
  .signal_filter.coef1 = 2,
  .base_avrg.n2_order = 12,
  .non_activity_avrg.n2_order = 15,
```

```
  .entry_event_cnt = 4,
  .deadband_cnt = 4,
  .signal_to_noise_ratio = 3,
  .min_noise_limit = 60,
  .dc_track_enabled = 1,
  .dc_track_cnt = 100,
};
```

5.1.2 Electrodes

Electrode setup. The following setup is common in nt_setup.c

```
/* keypad control build from FRDM KE15Z self-cap electrodes */
const struct nt_electrode El1 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_1,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_1,
  .gpio_input = {GPIO_PTC0},
  .baseline_level = 57593,
  .tsi_hw_config = &tsi_hw_config,
};
const struct nt_electrode El2 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_1,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_2,
  .gpio_input = {GPIO_PTC1},
  .baseline_level = 58346,
  .tsi_hw_config = &tsi_hw_config,
};
/* keypad control build from FRDM-TOUCH mutual-cap electrodes */
const struct nt_electrode El_3 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_3,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_1,
  .gpio_input = {GPIO_PTD5,GPIO_PTE11},
  .baseline_level = 11640,
  .tsi_hw_config = &tsi_hw_config,
};
const struct nt_electrode El_4 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
```

```
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_3,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_2,
  .gpio_input = {GPIO_PTD6,GPIO_PTE11},
  .baseline_level = 11557,
};
const struct nt_electrode El_5 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_3,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_3,
  .gpio_input = {GPIO_PTD5,GPIO_PTE10},
  .baseline_level = 14512,
};
const struct nt_electrode El_6 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_3,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_MUTUAL_ELECTRODE_4,
  .gpio_input = {GPIO_PTD6, GPIO_PTE10},
  .baseline_level = 14512,
};
/* slider control build from FRDM-TOUCH self-cap electrodes */
const struct nt_electrode El_7 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_7,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_SLIDER_ELECTRODE_1,
  .gpio_input = {GPIO_PTD7},
  .baseline_level = 55358,
};
const struct nt_electrode El_8 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_7,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_SLIDER_ELECTRODE_2,
  .gpio_input = {GPIO_PTD1},
  .baseline_level = 55247,
};
/* rotary control build from FRDM-TOUCH self-cap electrodes */
```

```
const struct nt_electrode El_9 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_7,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_1,
  .gpio_input = {GPIO_PTE5},
  .baseline_level = 50554,
};
const struct nt_electrode El_10 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_7,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_2,
  .gpio_input = {GPIO_PTE4},
  .baseline_level = 52651,
};
const struct nt_electrode El_11 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_7,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_3,
  .gpio_input = {GPIO_PTB5},
  .baseline_level = 53261,
};
const struct nt_electrode El_12 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_7,
  .keydetector_interface = &nt_keydetector_usafa_interface,
  .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_4,
  .gpio_input = {GPIO_PTB4},
  .baseline_level = 53803,
  .tsi_hw_config = &tsi_hw_config,
};
/* Testing electrode using same TSI channel but different HW config */
const struct nt_electrode El_13 = {
  .shielding_electrode = NULL,
  .multiplier = 0,
  .divider = 0,
  .shield_threshold = 5,
  .shield_gain = 30,
  .shield_attn = 1,
  .shield_sens = 800,
  .keydetector_params.usafa = &nt_keydetector_usafa_El_1,
  .keydetector_interface = &nt_keydetector_usafa_interface,
```

```
  .pin_input = FRDM_TOUCH_BOARD_TSI_1,
  .gpio_input = {GPIO_PTC0},
  .baseline_level = 57593,
  .tsi_hw_config = &tsi_hw_config1,
};
```

5.1.3 Modules

This object depends on the type of a module. This structure binds together electrodes assigned to module and other parameter (watermarks, autocalibration, safety, instance, ...) depending on module. The module setup is displayed below (including the hardware configuration). Firstly all module electrode must be assigned.

```
/* Modules */
const struct nt_electrode * const module_electrodes[] = {              \
  &El1,  &El2,                        /* FRDM-KE15Z electrodes */     \
  &El3,  &El4,  &El5,  &El6,          /* FRDM-TOUCH electrodes */     \
  &El7,  &El8,                        /* analog slider electrodes */  \
  &El9,  &El10,  &El11,  &El12,       /* analog rotary electrodes */  \
  &El13, NULL};                       /* debugging electrode using different TSI HW configuration */  \
```

5.1.3.1 TSI module

Hardware configuration of TSI v5 module is shown below.

```
/* Self-cap and mutual-cap mode configurations, the first one is default configuration, second is debuging
       configuration used for El_13 only */
const tsi_config_t tsi_hw_config =
{
    /* Self capacitance measurement config */
    .configSelfCap.commonConfig.mainClock = kTSI_MainClockSlection_0,      /* Set main clock
      */
    .configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,      /* Choose sensing mode */
    .configSelfCap.commonConfig.dvolt = kTSI_DvoltOption_0,                /* DVOLT option value  */
    .configSelfCap.commonConfig.cutoff = kTSI_SincCutoffDiv_0,             /* Cutoff divider  */
    .configSelfCap.commonConfig.order = kTSI_SincFilterOrder_2,            /* SINC filter order */
    .configSelfCap.commonConfig.decimation = kTSI_SincDecimationValue_4,   /* SINC decimation value */
    .configSelfCap.commonConfig.chargeNum = kTSI_SscChargeNumValue_7,      /* SSC output bit0's period
       setting */
    .configSelfCap.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_5,  /* SSC output bit1's period
       setting */
    .configSelfCap.enableSensitivity = true,                               /* Enable sensitivity boost of
       self-cap or not */
    .configSelfCap.xdn = kTSI_SensitivityXdnOption_3,                      /* Sensitivity XDN option   */
    .configSelfCap.ctrim = kTSI_SensitivityCtrimOption_0,                  /* Sensitivity CTRIM option */
    .configSelfCap.inputCurrent = kTSI_CurrentMultipleInputValue_0,        /* Input current multiple   */
    .configSelfCap.chargeCurrent = kTSI_CurrentMultipleChargeValue_0,      /* Charge/Discharge current
       multiple */
    /* Mutual capacitance measurement config */
    .configMutual.commonConfig.mainClock = kTSI_MainClockSlection_0,       /* Set main clock */
    .configMutual.commonConfig.mode = kTSI_SensingModeSlection_Mutual,     /* Choose sensing mode */
    .configMutual.commonConfig.dvolt = kTSI_DvoltOption_3,                 /* DVOLT option value  */
    .configMutual.commonConfig.cutoff = kTSI_SincCutoffDiv_0,              /* Cutoff divider  */
    .configMutual.commonConfig.order = kTSI_SincFilterOrder_2,             /* SINC filter order */
    .configMutual.commonConfig.decimation = kTSI_SincDecimationValue_4,    /* SINC decimation value */
    .configMutual.commonConfig.chargeNum = kTSI_SscChargeNumValue_4,       /* SSC output bit0's period
       setting */
    .configMutual.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_2,   /* SSC output bit1's period
```

```
        setting */
    .configMutual.preCurrent = kTSI_MutualPreCurrent_4uA,                    /* Vref generator current  */
    .configMutual.preResistor = kTSI_MutualPreResistor_4k,                   /* Vref generator resistor */
    .configMutual.senseResistor = kTSI_MutualSenseResistor_10k,              /* I-sense generator resistor */
    .configMutual.boostCurrent = kTSI_MutualSenseBoostCurrent_0uA,           /* Sensitivity boost current
        setting */
    .configMutual.txDriveMode = kTSI_MutualTxDriveModeOption_0,              /* TX drive mode control setting
        */
    .configMutual.pmosLeftCurrent = kTSI_MutualPmosCurrentMirrorLeft_32,     /* Pmos current mirror on the
        left side  */
    .configMutual.pmosRightCurrent = kTSI_MutualPmosCurrentMirrorRight_1,    /* Pmos current mirror on the
        right side */
    .configMutual.enableNmosMirror = true,                                   /* Enable/disable Nmos current
        mirror setting */
    .configMutual.nmosCurrent = kTSI_MutualNmosCurrentMirror_1,              /* Nmos current mirror setting
       */
};
const tsi_config_t tsi_hw_config1 = {
    .configSelfCap.commonConfig.mainClock = kTSI_MainClockSlection_0,
    .configSelfCap.commonConfig.ssc_mode = kTSI_ssc_prbs_method,
    .configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,
    .configSelfCap.commonConfig.dvolt = kTSI_DvoltOption_3,
    .configSelfCap.commonConfig.cutoff = kTSI_SincCutoffDiv_0,
    .configSelfCap.commonConfig.order = kTSI_SincFilterOrder_2,
    .configSelfCap.commonConfig.decimation = kTSI_SincDecimationValue_7,
    .configSelfCap.commonConfig.chargeNum = kTSI_SscChargeNumValue_4,
    .configSelfCap.commonConfig.prbsOutsel = kTSI_SscPrbsOutsel_2,
    .configSelfCap.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_2,
    .configSelfCap.commonConfig.ssc_prescaler = kTSI_ssc_div_by_2,
    .configSelfCap.enableSensitivity = true,
    .configSelfCap.enableShield = false,
    .configSelfCap.xdn = kTSI_SensitivityXdnOption_3,
    .configSelfCap.ctrim = kTSI_SensitivityCtrimOption_0,
    .configSelfCap.inputCurrent = kTSI_CurrentMultipleInputValue_0,
    .configSelfCap.chargeCurrent = kTSI_CurrentMultipleChargeValue_0,
    .configMutual.commonConfig.mainClock = kTSI_MainClockSlection_0,
    .configMutual.commonConfig.ssc_mode = kTSI_ssc_prbs_method,
    .configMutual.commonConfig.mode = kTSI_SensingModeSlection_Mutual,
    .configMutual.commonConfig.dvolt = kTSI_DvoltOption_3,
    .configMutual.commonConfig.cutoff = kTSI_SincCutoffDiv_0,
    .configMutual.commonConfig.order = kTSI_SincFilterOrder_2,
    .configMutual.commonConfig.decimation = kTSI_SincDecimationValue_4,
    .configMutual.commonConfig.chargeNum = kTSI_SscChargeNumValue_4,
    .configMutual.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_2,
    .configMutual.commonConfig.prbsOutsel = kTSI_SscPrbsOutsel_2,
    .configMutual.commonConfig.ssc_prescaler = kTSI_ssc_div_by_2,
    .configMutual.preCurrent = kTSI_MutualPreCurrent_4uA,
    .configMutual.preResistor = kTSI_MutualPreResistor_4k,
    .configMutual.senseResistor = kTSI_MutualSenseResistor_10k,
    .configMutual.boostCurrent = kTSI_MutualSenseBoostCurrent_0uA,
    .configMutual.txDriveMode = kTSI_MutualTxDriveModeOption_0,
    .configMutual.pmosLeftCurrent = kTSI_MutualPmosCurrentMirrorLeft_32,
    .configMutual.pmosRightCurrent = kTSI_MutualPmosCurrentMirrorRight_1,
    .configMutual.enableNmosMirror = true,
    .configMutual.nmosCurrent = kTSI_MutualNmosCurrentMirror_1,
    .thresl = 0,
    .thresh = 65535,
};
/* GPIO interface for safety tests */
extern const struct nt_module_gpio_user_interface gpio_interface;
/* Safety parameter structure */
const struct nt_module_safety_gpio_params my_safety_params = {
  .user_interface = &gpio_interface,
};
```

For operation with nt_module, see the code below.

```
const struct nt_module module = {
  .interface = &nt_module_tsi_interface,    /* Module interface. Can't be
       NULL. */
  .electrodes = &module_electrodes[0],      /* A pointer to the list of electrodes. Can't be NULL. */
  .config = (void*)&tsi_hw_config,          /* An instance module params. Can't be NULL.*/
  .wtrmark_hi = 65535,                      /* signal too high watermark */
  .wtrmark_lo = 0,                          /* signal too low watermark */
  .instance = 0,                            /* An instance of the module. */
  .safety_interface = &nt_safety_interface, /* An instance module Safety
       params. NULL when safety not needed */
  .safety_params.gpio = &my_safety_params,  /* Module Safety interface. NULL when safety
       not needed */
  .recalib_config = (void*)&recalib_config, /* Recalibration structure for autocalibration */

};
```

5.1.3.2 TSI module

Hardware configuration of TSI v4 module is shown below.

```
/* Self-cap and mutual-cap mode config */
const tsi_config_t tsi_hw_config =
{
  .prescaler = kTSI_ElecOscPrescaler_64div,
  .extchrg = kTSI_ExtOscChargeCurrent_4uA,
  .refchrg = kTSI_RefOscChargeCurrent_32uA,
  .nscn = kTSI_ConsecutiveScansNumber_24time,
  .mode = kTSI_AnalogModeSel_Capacitive,
  .dvolt = kTSI_OscVolRailsOption_0,
  .thresh = 0,
  .thresl = 0,
  .filter = kTSI_FilterBits_0,                  /* Must be set due to common init for noise mode and
       non-noise mode */
};
```

For operation with nt_module, see the code below.

```
const struct nt_module module = {
  .interface = &nt_module_tsi_interface,    /* Module interface. Can't be
       NULL. */
  .electrodes = &module_electrodes[0],      /* A pointer to the list of electrodes. Can't be NULL. */
  .config = (void*)&tsi_hw_config,          /* An instance module params. Can't be NULL.*/
  .wtrmark_hi = 0,                          /* signal too high watermark */
  .wtrmark_lo = 0,                          /* signal too low watermark */
  .instance = 0,                            /* An instance of the module. */
};
```

5.1.3.3 CS module

Hardware configuration of CS module is shown below.

```
/* Self-cap config */
const cs_config_t tsi_hw_config =
{
    /* For acitve mode */
    .activeClockDivider = 99U,  /* Active mode clock divider, occupy 9 bits, generates CLK_CS clock from
       CLK_APB clock using F(CS) = F(APB) / (clockDivider+1) */
    .activeDetectPeriod = 350U, /* Detect period (how many CLK_CS clock cycles to scan a channel, occupy 16
```

```
      bits */
    .activeIdlePeriod   = 5U,   /* Idle period (how many clock cycles to wait between channel scans),
       occupy 16 bits */
    .activeOscFreq      = 63U,  /* Relaxation oscillator frequency configuration. The value will adjust the
       current being driven to input channels by equation: 0.4 + 0.2(OSC_FREQ+1) [uA], occupy 6 bits */
    .activeCounterGain  = 25U,  /* Can increase counter value by multiplication with this gain, cccupy 8
       bits */
    /* For low power mode */
    .lowPowerChannelNum = 4U,   /* Indicates how many consecutive CLK_32K cycles need to meet the threshold
       value before the wake-up interrupt is triggered. The actual interpretation by the hardware of this value is
       such that one more than the value programmed here is used, occupy 4 bits */
    .lowPowerThreshold  = 3150U,/* The value represents the upper bound of the counter output in terms of
       what is considered a valid touch event, occupy 16 bits*/
    .lowPowerDebonceNum = 50U,  /* Selects which channel index to monitor while in LP mode, occupy 3 bits*/
    .lowPowerIdlePeriod = 20U,  /* Idle period between previous scanning end and next scanning start,
       occupy 16 bits */
    .lowPowerOscFreq    = 30U,  /* OSC bias select, occupy 6 bits */
};
```

For operation with nt_module, see the code below.

```
const struct nt_module module =
{
  .interface = &nt_module_cs_interface,     /* Module interface. Can't be
       NULL. */
  .electrodes = &module_electrodes[0],      /* A pointer to the list of electrodes. Can't be NULL. */
  .config = (void*)&tsi_hw_config,          /* An instance module params. Can't be NULL.*/
  .wtrmark_hi = 0,                          /* signal too high watermark */
  .wtrmark_lo = 0,                          /* signal too low watermark */
  .instance = 0,                            /* An instance of the module. */
};
```

5.1.4 Controls

Control setup - see the description below.

5.1.4.1 Keypad

For operation using the nt_control_keypad see the code below.

```
const struct nt_electrode  * const control_0_electrodes[] = { \
    &El1,  &El2                 /* FRDM-KE15Z electrodes */   \
    &El3,  &El4,  &El5,  &El6,  /* FRDM-TOUCH electrodes */
NULL};
const struct nt_control_keypad nt_control_keypad_keypad_0 = {
  .groups = NULL,
  .groups_size = 0,
};
const struct nt_control keypad_0 = {
  .electrodes = &keypad_0_controls[0],
  .interface = &nt_control_keypad_interface,
  .control_params.keypad = &nt_control_keypad_keypad_0,
};
```

5.1.4.2 Analog slider

For operation using the nt_control_aslider, see the code below.

```
const struct nt_electrode * const aslider_0_controls[] = { \
  &El7,  &El8, NULL
};
const struct nt_control_aslider nt_control_aslider_aslider_0 = {
    .range = 160,       /* Maximum range for the ram->position value. */
    .insensitivity = 2, /* Set the insensitivity to ignore changes less than 4. */
};
const struct nt_control aslider_0 = {   /
  .electrodes = &aslider_0_controls[0],
  .interface = &nt_control_aslider_interface,
  .control_params.aslider = &nt_control_aslider_aslider_0,
};
```

5.1.4.3 Analog rotary

For operation using the nt_control_arotary, see the code below.

```
const struct nt_electrode * const arotary_0_controls[] = { \
  &El9,  &El10,  &El11,  &El12, NULL
};
const struct nt_control_arotary nt_control_arotary_arotary_0 = {
  .range = 72,  /* Maximum range for the ram->position value. */
};
const struct nt_control arotary_0 = {
  .electrodes = &arotary_0_controls[0],
  .interface = &nt_control_arotary_interface,
  .control_params.arotary = &nt_control_arotary_arotary_0,
};
```

5.1.5 System

The System is represented by the nt_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 the list of all electrodes used)
- time period and initialization time
- multiplication of time period for calling safety check
- relative path for GUI configuration

```
const struct nt_control * const system_0_controls[] = {
  &keypad_0,  &aslider_0,  &arotary_0, NULL
};
const struct nt_module * const system_0_modules[] = {
  &module, NULL
};
const struct nt_system system_0 = {
    .controls = &controls[0],
    .modules = &modules[0],
    .time_period = 5,
    .init_time = 400,
    .gui_path = "..\frdm_ke15z_config.xml",
    .safety_period_multiple = 0,
};
```

5.1.6 Low-Power mode

The NT library implements the low-power function that enables the MCU to wake up from the Low-Power mode if the defined source electrode detects a touch event. NOTE: Just one electrode can be selected and used as a wake-up source for the Low-Power mode. The Low-Power mode is not supported in FRDM touch-sensing Demo for FRDM-KE15Z and FRDM-KE16Z.

5.1.7 Proximity mode

The Proximity feature is newly supported in the current NT revision. The Proximity feature is not supported in FRDM touch-sensing Demo for FRDM-KE15Z and FRDM-KE16Z.

5.1.8 System Callbacks

System callbacks may be useful to signalize system events, such as the touch-sensing module measured DATA READY, OVERRUN, or OVERFLOW.

Chapter 6
NXP Touch library integration.

This chapter describes how to integrate the NXP Touch library into an existing application project. There are several ways of using the NXP Touch library. The simplest way is to use the prepared MCUXpresso SDK-based example projects or the NXP Touch library can be add to your existing application project.

6.1 Creating Application with NXP Touch Library

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.

This chapter presents the option where all library files are added to the user application project, and compiled together with the application. The other option is to use the prepared MCUXpresso SDK-based projects where all toolchains are supported (see: the MCUXpresso SDK/boards/*your_board*/demo_apps/touch_sensing). This chapter describes the NXP Touch library integration into the IAR Embedded Workbench IDE only. If you want to use another toolchain, see the *Getting Started with MCUXpresso SDK* device documentation, where the integration into all toolchains is explained using an example project. The application demonstrated in this section uses keypad consists of two touch electrodes on the FRDM-K-E15Z board, four touch electrodes, and the analog slider and analog rotary controls on the FRDM-TOUCH board. The following paragraphs explain how to create such projects.

6.1.1 Adding Library Files into the Project

The library can be easily integrated into your application by adding the NXP Touch source files into your project. See the directory section to understand the files and folders of the NXP 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 /middleware/touch/include folder).
- The source code files must be added into the project (the MCUXpresso SDK/middleware/touch/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 the startup code, the linker files, and some low-level driver code from the MCUXpresso SDK or from the existing application project, (if used). This code is not considered to be a part of the NXP Touch library, but it serves as a base for example applications. You can also reuse the MCUXpresso SDK linker files and drivers in your own custom applications. However, it is better to get the latest MCUXpresso SDK version from the NXP website.

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

Figure 6.1.1: Workspace directory

6.1.2 Setting 'include' Search Paths

The figure shows how to set up the search paths in IAR Embedded Workbench IDE. Only one include path is needed from the NXP Touch point of view. The pre-processor symbols should be defined to identify the CPU and Board for the MCUXpresso SDK low-level code. The valid options can be found in the fsl_device_registers.h file, located in the MCUXpresso SDK/devices/directory.

Figure 6.1.2: Include search path for KE15z

Figure 6.1.3: Include search path for QN908x

Figure 6.1.4: Include search path for K32L2A4S

All search paths: Only the last four paths are currently mandatory for the NXP Touch library, the others are required by MCUXpresso SDK. The path list for the KE15z based board with TSI v5.

```
PROJ_DIR$/../../../../../CMSIS/Include
$PROJ_DIR$/../../../../../devices
$PROJ_DIR$/..
$PROJ_DIR$/../../../../../devices/MKE15Z7/drivers
$PROJ_DIR$/../../../../../devices/MKE15Z7
$PROJ_DIR$/../../../../../devices/MKE15Z7/utilities/io
$PROJ_DIR$/../../../../../devices/MKE15Z7/utilities/str
$PROJ_DIR$/../../../../../devices/MKE15Z7/utilities/log
$PROJ_DIR$/../../../../../devices/MKE15Z7/utilities
$PROJ_DIR$/../../../../../middleware/touch/boards/frdmke15z
$PROJ_DIR$/../../../../../middleware/touch/freemaster/driver/src_common
$PROJ_DIR$/../../../../../middleware/touch/freemaster/driver/src_platforms/Kxx
$PROJ_DIR$/../../../../../middleware/touch/include
```

The path list for the QN908x based board with CS module.

```
PROJ_DIR$/../../../../../CMSIS/Include
$PROJ_DIR$/../../../../../devices
$PROJ_DIR$/..
```

```
$PROJ_DIR$/../../../../../devices/QN908XC/drivers
$PROJ_DIR$/../../../../../devices/QN908XC
$PROJ_DIR$/../../../../../devices/QN908XC/utilities/io
$PROJ_DIR$/../../../../../devices/QN908XC/utilities/str
$PROJ_DIR$/../../../../../devices/QN908XC/utilities/log
$PROJ_DIR$/../../../../../devices/QN908XC/utilities
$PROJ_DIR$/../../../../../middleware/touch/boards/qn908xcdk
$PROJ_DIR$/../../../../../middleware/touch/freemaster/driver/src_common
$PROJ_DIR$/../../../../../middleware/touch/freemaster/driver/src_platforms/QN908x
$PROJ_DIR$/../../../../../middleware/touch/include
```

TThe path list for the K32L2A based board with TSIv 4.

```
$PROJ_DIR$/..
$PROJ_DIR$/../../../../../middleware/touch/boards/frdmk32l2a4s
$PROJ_DIR$/../../../../../CMSIS/Include
$PROJ_DIR$/../../../../../devices
$PROJ_DIR$/../middleware/touch/include
$PROJ_DIR$/../../../../../devices/K32L2A41A/drivers
$PROJ_DIR$/../../../../../middleware/touch/include
$PROJ_DIR$/../../../../../devices/K32L2A41A
$PROJ_DIR$/../../../../../devices/K32L2A41A/utilities/str
$PROJ_DIR$/../../../../../devices/K32L2A41A/utilities/debug_console
$PROJ_DIR$/../../../../../components/uart
$PROJ_DIR$/../../../../../components/serial_manager
$PROJ_DIR$/../../../../../components/lists
$PROJ_DIR$/../../../../../middleware/freemaster/src/common
$PROJ_DIR$/../../../../../middleware/freemaster/src/drivers/mcuxsdk/serial
$PROJ_DIR$/../../../../../middleware/freemaster/src/drivers/mcuxsdk/can
$PROJ_DIR$/../../../../../middleware/freemaster/src/platforms/gen32le
$PROJ_DIR$/../../../../../devices/K32L2A41A/utilities
```

6.1.3 Setting the 'linker' path

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

Figure 6.1.5: Linker command file path for KE15z

Figure 6.1.6: Linker command file path for QN908x

Figure 6.1.7: Linker command file path for K32L2A4S

6.1.4 Application Setup

To define the modules, electrodes, controls, and system, create the initialized instances of the structure types, as described in section Configuring the Library.

The code below shows an example configuration of two electrodes on the FRDM-KE15Z board, further four electrodes, analog rotary, and analog slider controls on the FRDM-TOUCH auxiliary board.

There are several key detectors available in the NXP Touch library. The electrode structure types must always match the module and algorithm types.

This is an example of the uSAFA key detector configuration:

```
/* uSAFA keydetector settings */
const struct nt_keydetector_usafa keydec_usafa =
{
/* Electrodes */
    .signal_filter = {2},
    .base_avrg = {.n2_order = 12},
    .non_activity_avrg =  {.n2_order = NT_FILTER_MOVING_AVERAGE_MAX_ORDER
      },
    .entry_event_cnt = 4,
```

```
    .signal_to_noise_ratio = 6,
    .deadband_cnt = 4,
    .min_noise_limit = 100,
};

const struct nt_keydetector_usafa keydec_usafa_slider =
{
    .signal_filter = {2},
    .base_avrg = {.n2_order = 12},
    .non_activity_avrg =  {.n2_order = NT_FILTER_MOVING_AVERAGE_MAX_ORDER
      },
    .entry_event_cnt = 4,
    .signal_to_noise_ratio = 3,
    .deadband_cnt = 4,
    .min_noise_limit = 50,
};

const struct nt_keydetector_usafa keydec_usafa_mutual =
{
    .signal_filter = {2},
    .base_avrg = {.n2_order = 12},
    .non_activity_avrg =  {.n2_order = NT_FILTER_MOVING_AVERAGE_MAX_ORDER
      },
    .entry_event_cnt = 4,
    .signal_to_noise_ratio = 3,
    .deadband_cnt = 4,
    .min_noise_limit = 50,
};
```

The electrode structure types must match the hardware module used for the data-measurement algorithm in the application. In this case, it is the nt_electrode type. Define the electrode parameters and the nt_-keydetector interface.

```
/* Electrodes */
const struct nt_electrode electrode_0 =
{
    .pin_input = BOARD_TSI_ELECTRODE_1,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa,

};

const struct nt_electrode electrode_1 =
{
    .pin_input = BOARD_TSI_ELECTRODE_2,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa,
};

const struct nt_electrode electrode_2 =
{

    .pin_input = NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_1,
      FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_1),
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_mutual,
};

const struct nt_electrode electrode_3 =
{

    .pin_input = NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_2,
      FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_2),
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_mutual,
```

```
};

const struct nt_electrode electrode_4 =
{
    .pin_input = NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_3,
      FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_3),
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_mutual,
};

const struct nt_electrode electrode_5 =
{
    .pin_input = NT_TSI_TRANSFORM_MUTUAL(FRDM_TOUCH_BOARD_TSI_MUTUAL_RX_ELECTRODE_4,
      FRDM_TOUCH_BOARD_TSI_MUTUAL_TX_ELECTRODE_4),
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_mutual,
};

const struct nt_electrode electrode_6 =
{
    .pin_input = FRDM_TOUCH_BOARD_TSI_SLIDER_ELECTRODE_1,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_slider,
};

const struct nt_electrode electrode_7 =
{
    .pin_input = FRDM_TOUCH_BOARD_TSI_SLIDER_ELECTRODE_2,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_slider,
};

const struct nt_electrode electrode_8 =
{
    .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_1,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_slider,
};

const struct nt_electrode electrode_9 =
{
    .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_2,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_slider,
};

const struct nt_electrode electrode_10 =
{
    .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_3,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_slider,
};

const struct nt_electrode electrode_11 =
{
    .pin_input = FRDM_TOUCH_BOARD_TSI_ROTARY_ELECTRODE_4,
    .keydetector_interface = &nt_keydetector_usafa_interface,
    .keydetector_params.usafa = &keydec_usafa_slider,
};
```

The Kinetis E family MCUs contain the TSI module version 5. The TSI hardware setup and configuraion for recalibration function are shown below.

```
/* Modules */
const struct nt_electrode * const module_0_electrodes[] = { \
```

```
    &electrode_0, &electrode_1,                                        /* FRDM-KE15Z electrodes */ \
    &electrode_2, &electrode_3, &electrode_4, &electrode_5,            /* FRDM-TOUCH electrodes */ \
    &electrode_6, &electrode_7,                                        /* analog slider elect. */  \
    &electrode_8, &electrode_9, &electrode_10, &electrode_11, NULL}; /* analog rotary elect. */

/* Self-cap and mutual-cap mode config */
const tsi_config_t hw_config =
{
    /* Self capacitance measurement config */
    .configSelfCap.commonConfig.mainClock = kTSI_MainClockSlection_0,
    .configSelfCap.commonConfig.mode = kTSI_SensingModeSlection_Self,
    .configSelfCap.commonConfig.dvolt = kTSI_DvoltOption_0,
    .configSelfCap.commonConfig.cutoff = kTSI_SincCutoffDiv_0,
    .configSelfCap.commonConfig.order = kTSI_SincFilterOrder_2,
    .configSelfCap.commonConfig.decimation = kTSI_SincDecimationValue_4,
    .configSelfCap.commonConfig.chargeNum = kTSI_SscChargeNumValue_7,
    .configSelfCap.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_5,
    .configSelfCap.enableSensitivity = true,
    .configSelfCap.xdn = kTSI_SensitivityXdnOption_3,
    .configSelfCap.ctrim = kTSI_SensitivityCtrimOption_0,
    .configSelfCap.inputCurrent = kTSI_CurrentMultipleInputValue_0,
    .configSelfCap.chargeCurrent = kTSI_CurrentMultipleChargeValue_0,
    /* Mutual capacitance measurement config */
    .configMutual.commonConfig.mainClock = kTSI_MainClockSlection_0,
    .configMutual.commonConfig.mode = kTSI_SensingModeSlection_Mutual,
    .configMutual.commonConfig.dvolt = kTSI_DvoltOption_3,
    .configMutual.commonConfig.cutoff = kTSI_SincCutoffDiv_0,
    .configMutual.commonConfig.order = kTSI_SincFilterOrder_2,
    .configMutual.commonConfig.decimation = kTSI_SincDecimationValue_4,
    .configMutual.commonConfig.chargeNum = kTSI_SscChargeNumValue_4,
    .configMutual.commonConfig.noChargeNum = kTSI_SscNoChargeNumValue_2,
    .configMutual.preCurrent = kTSI_MutualPreCurrent_4uA,
    .configMutual.preResistor = kTSI_MutualPreResistor_4k,
    .configMutual.senseResistor = kTSI_MutualSenseResistor_10k,
    .configMutual.boostCurrent = kTSI_MutualSenseBoostCurrent_0uA,
    .configMutual.txDriveMode = kTSI_MutualTxDriveModeOption_0,
    .configMutual.pmosLeftCurrent = kTSI_MutualPmosCurrentMirrorLeft_32,
    .configMutual.pmosRightCurrent = kTSI_MutualPmosCurrentMirrorRight_1,
    .configMutual.enableNmosMirror = true,
    .configMutual.nmosCurrent = kTSI_MutualNmosCurrentMirror_1,
};

struct nt_tsi_recalib_config recalib_config = {
    .SelfRangeMin     = 50000,
    .SelfRangeMax     = 60000,
    .SelfRangeDeltaMin= 400,
    .SelfRangeDeltaMax= 1300,
    .MutRangeMin      = 12000,
    .MutRangeMax      = 16000,
    .MutRangeDeltaMin = 300,
    .MutRangeDeltaMax = 600,
};

const struct nt_module tsi_module =
{
    .interface = &nt_module_tsi_interface,
    .electrodes = &module_0_electrodes[0],
    .config = (void*)&hw_config,
    .recalib_config = (void*)&recalib_config,
    .instance = 0,
};
```

When the modules and electrodes are set up, you may define the Controls. In this case, the control_0 is the Keypad control.

```
/* Keypad */
```

```
const struct nt_electrode  * const control_0_electrodes[] = { \
    &electrode_0, &electrode_1,                                    /* FRDM-KE15Z electrodes */  \
    &electrode_2, &electrode_3, &electrode_4, &electrode_5, NULL}; /* FRDM-TOUCH electrodes */

const struct nt_control_keypad keypad_params =
{
  .groups = NULL,
  .groups_size = 0,
  .multi_touch = (uint32_t[]){0x0C,0x18,0x30,0x24,0x3C},
  .multi_touch_size = 5,
};

const struct nt_control keypad_0 =
{
  .interface = &nt_control_keypad_interface,
  .electrodes = control_0_electrodes,
  .control_params.keypad = &keypad_params,
};
```

The next control is the Analog Slider control with the following setup.

```
/*Analog Slider*/
const struct nt_electrode  * const control_1_electrodes[] = { \
    &electrode_6, &electrode_7, NULL};

struct nt_control_aslider aslider_params = {
    .range = 160,
    .insensitivity = 2,
};

const struct nt_control aslider_0 =
{
  .interface = &nt_control_aslider_interface,
  .electrodes = control_1_electrodes,
  .control_params.aslider = &aslider_params,
};
```

The last control is the Analog Rotary control with the following setup.

```
/*Analog Rotary*/
const struct nt_electrode  * const control_2_electrodes[] = { \
    &electrode_8, &electrode_9, &electrode_10, &electrode_11, NULL};

struct nt_control_arotary arotary_params = {
    .range = 72,
};

const struct nt_control arotary_0 =
{
  .interface = &nt_control_arotary_interface,
  .electrodes = control_2_electrodes,
  .control_params.arotary = &arotary_params,
};
```

Now you can connect all the pieces together in the system structure.

```
/* System */
const struct nt_control * const controls[] = {&keypad_0, &aslider_0, &arotary_0,
      NULL};
const struct nt_module * const modules[] = {&tsi_module, NULL};
```

```
const struct nt_system system_0 = {
    .controls = &controls[0],
    .modules = &modules[0],
    .time_period = 5,
    .init_time = 400,
};
```

6.1.5 The main() Function

The application code is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include "fsl_device_registers.h"
#include "fsl_debug_console.h"

#include "fsl_ftm.h"
#include "fsl_lpit.h"

#include "main.h"
#include "board.h"
#include "freemaster.h"
#include "fsl_lpuart.h"
#include "pin_mux.h"

extern void TSI_DRV_IRQHandler(uint32_t instance);

static void pit_init(void);
static void ftm_init(void);

static void init_freemaster_uart(void);
static void keypad_callback(const struct nt_control *control,
                            enum nt_control_keypad_event event,
                            uint32_t index);

static void aslider_callback(const struct nt_control *control,
                             enum nt_control_aslider_event event,
                             uint32_t position);

static void arotary_callback(const struct nt_control *control,
                             enum nt_control_arotary_event event,
                             uint32_t position);

/* Call when the TSI counter overflows 65535 */
static void system_callback(uint32_t event);

static void SetHueBrightness(uint32_t hue_angle, uint32_t brightness);

uint8_t nt_memory_pool[5000];

uint32_t brightness_global, hue_angle_global;
uint8_t duty_cycle_red, duty_cycle_green, duty_cycle_blue;

const ftm_config_t my_ftm_config = {
    .prescale = kFTM_Prescale_Divide_1,                 /* Divide FTM clock by 1 */
    .bdmMode = kFTM_BdmMode_0,                          /* FTM behavior in BDM mode */
    .pwmSyncMode = kFTM_SoftwareTrigger,                /* Software trigger is used to update registers */
    .reloadPoints = 0,                                  /* No intermediate register load */
    .faultMode = kFTM_Fault_Disable,                    /* Fault control disabled for all channels */
    .faultFilterValue = 0,                              /* Disable the fault filter */
    .deadTimePrescale = kFTM_Deadtime_Prescale_1,       /* Divide the system clock by 1 */
    .deadTimeValue = 0,                                 /* No counts are inserted */
    .extTriggers = 0,                                   /* No external trigger */
```

```
    .chnlInitState = 0,                                 /* Initialization value is 0 for all channels */
    .chnlPolarity = 0x00,                               /* Active high polarity for all channels */
    .useGlobalTimeBase = false,                         /* Use internal FTM counter as timebase */
};

const ftm_chnl_pwm_signal_param_t my_ftm_channel_config[3] = {
    {
    .chnlNumber = kFTM_Chnl_0,
    .level = kFTM_LowTrue,
    .dutyCyclePercent = 0U,
    .firstEdgeDelayPercent = 0U,
    },
    {
    .chnlNumber = kFTM_Chnl_1,
    .level = kFTM_LowTrue,
    .dutyCyclePercent = 0U,
    .firstEdgeDelayPercent = 0U,
    },
    {
    .chnlNumber = kFTM_Chnl_2,
    .level = kFTM_LowTrue,
    .dutyCyclePercent = 0U,
    .firstEdgeDelayPercent = 0U,
    },
};

const lpit_config_t my_pit_config = {
    .enableRunInDebug = false,
    .enableRunInDoze  = false,
};

/*
 * This list describes all TSA tables that should be exported to the
 * FreeMASTER application.
 */
#ifndef FMSTR_PE_USED
  FMSTR_TSA_TABLE_LIST_BEGIN()
      FMSTR_TSA_TABLE(nt_frmstr_tsa_table)
  FMSTR_TSA_TABLE_LIST_END()
#endif


#define nt_printf(...) /* do nothing - the debug lines are used by FreeMASTER */


int main(void)
{
  int32_t result;

  /* Init board hardware */
  BOARD_InitPins();

  BOARD_BootClockRUN();

  /*LED init*/
  LED_RED1_INIT(1);
  LED_GREEN1_INIT(1);
  LED_BLUE_INIT(1);

  /* Init FTM0 */
  ftm_init();

  /* Initialize the OS abstraction layer */
  NT_OSA_Init();

  /* FreeMASTER communicates over the default UART instance */
  init_freemaster_uart();
```

```
/* FreeMASTER initialization */
(void)FMSTR_Init();

if ((result = nt_init(&system_0, nt_memory_pool, sizeof(nt_memory_pool))) !=
    NT_SUCCESS)
{
  switch(result)
  {
  case NT_FAILURE:
    nt_printf("\nCannot initialize NXP Touch due to a non-specific error.\n");
    break;
  case NT_OUT_OF_MEMORY:
    nt_printf("\nCannot initialize NXP Touch due to a lack of free memory.\n");
    printf("\nCannot initialize NXP Touch due to a non-specific error.\n");
    break;
  }
  while(1); /* add code to handle this error */
}


nt_printf("\nNXP Touch is successfully initialized.\n");

nt_printf("Unused memory: %d bytes, you can make the memory pool smaller without affecting the
     functionality.\n", (int)nt_mem_get_free_size());
/* Keypad electrodes*/
nt_electrode_enable(&electrode_0, 0);
nt_electrode_enable(&electrode_1, 0);
nt_electrode_enable(&electrode_2, 0);
nt_electrode_enable(&electrode_3, 0);
nt_electrode_enable(&electrode_4, 0);
nt_electrode_enable(&electrode_5, 0);
nt_control_enable(&keypad_0);
nt_control_keypad_set_autorepeat_rate(&keypad_0, 100, 1000);
nt_control_keypad_register_callback(&keypad_0, &keypad_callback);


/* Slider electrodes */
nt_electrode_enable(&electrode_6, 0);
nt_electrode_enable(&electrode_7, 0);
nt_control_enable(&aslider_0);
nt_control_aslider_register_callback(&aslider_0, &aslider_callback);

/* Rotary electrodes */
nt_electrode_enable(&electrode_8, 0);
nt_electrode_enable(&electrode_9, 0);
nt_electrode_enable(&electrode_10, 0);
nt_electrode_enable(&electrode_11, 0);
nt_control_enable(&arotary_0);
nt_control_arotary_register_callback(&arotary_0, &arotary_callback);

/* System TSI overflow warning callback */
nt_system_register_callback(&system_callback);

/* Auto TSI register recalibration function, not used as default */
if (recalib_enabled)
{
    recalib_status = (tsi_status_t) nt_module_recalibrate(&tsi_module, (
    void*) &recalib_config);
}
if (one_key_only)
nt_control_keypad_only_one_key_valid(&keypad_0, true);

pit_init();

while(1)
{
  nt_task();
```

```
    /* The FreeMASTER poll call must be called in the main application loop
    to handle the communication interface and protocol.
       In the LONG_INTR FreeMASTER interrupt mode, all the processing is done
    during the communication interrupt routine and the FMSTR_Poll() is
    compiled empty. */
    FMSTR_Poll();
  }
}


void LPIT0_IRQHandler(void)
{
  /* Clear the interrupt flag.*/
  LPIT_ClearStatusFlags(LPIT0, LPIT_MSR_TIF0_MASK);
  nt_trigger();
}

void TSI_IRQHandler(void)
{
  TSI_DRV_IRQHandler(0);
}


static void ftm_init(void)
{
  /* FTM0 Init */
  FTM_Init(FTM0, &my_ftm_config);

  /* Configure FTM0 channels 0, 1, 2 */
  FTM_SetupPwm(FTM0, my_ftm_channel_config, 3U, kFTM_EdgeAlignedPwm, 10000U, CLOCK_GetFreq(
      kCLOCK_CoreSysClk));

  /* Start timer */
  FTM_StartTimer(FTM0, kFTM_SystemClock);
}

static void pit_init(void)
{
  /* Set the source for the LPIT module */
  CLOCK_SetIpSrc(kCLOCK_Lpit0, kCLOCK_IpSrcSircAsync);

  /* Init PIT */
  LPIT_Init(LPIT0, &my_pit_config);

  /* Set PIT0 period 5ms */
  LPIT_SetTimerPeriod(LPIT0, kLPIT_Chnl_0, 5*CLOCK_GetFreq(kCLOCK_ScgSircAsyncDiv2Clk)/1000);

    /* Enable the PIT interrupt */
  LPIT_EnableInterrupts(LPIT0, LPIT_MIER_TIE0_MASK);
  EnableIRQ(LPIT0_IRQn);

  /* Start the PIT timer */
  LPIT_StartTimer(LPIT0, kLPIT_Chnl_0);
}

static void keypad_callback(const struct nt_control *control,
                            enum nt_control_keypad_event event,
                            uint32_t index)
{
  switch(event)
  {
  case NT_KEYPAD_RELEASE:

    switch (index) {
        case 0:
            break;
        case 1:
```

```
            break;
        case 2:
            break;
        case 3:
            break;
        case 4:
            break;
        case 5:
            break;
        default:
            break;
    }
    break;
  case NT_KEYPAD_TOUCH:

    switch (index) {
        case 0:
              /* YELLOW on, full brightness */
              hue_angle_global = 12;
              brightness_global = 120;
              SetHueBrightness(hue_angle_global, brightness_global);
            break;
        case 1:
              /* CYAN on, full brightness */
              hue_angle_global = 36;
              brightness_global = 120;
              SetHueBrightness(hue_angle_global, brightness_global);
            break;
        case 2:
              /* RED on, full brightness */
              hue_angle_global = 0;
              brightness_global = 120;
              SetHueBrightness(hue_angle_global, brightness_global);
            break;
        case 3:
              /* GREEN on, full brightness */
              hue_angle_global = 24;
              brightness_global = 120;
              SetHueBrightness(hue_angle_global, brightness_global);
            break;
        case 4:
              /* BLUE on, full brightness */
              hue_angle_global = 48;
              brightness_global = 120;
              SetHueBrightness(hue_angle_global, brightness_global);
            break;
        case 5:
              /* WHITE on, full brightness */
              hue_angle_global = 73;
              brightness_global = 120;
              SetHueBrightness(hue_angle_global, brightness_global);
            break;
        default:
            break;
    }
    break;

  case NT_KEYPAD_AUTOREPEAT:

    switch (index) {
        case 0:
            break;
        case 1:
            break;
        case 2:
            break;
        case 3:
            break;
```

```
        case 4:
            break;
        case 5:
            break;
        default:
            break;
    }
    break;
  default:
    break;

  }
}

static void aslider_callback(const struct nt_control *control,
                             enum nt_control_aslider_event event,
                             uint32_t position)
{
  switch(event)
  {
  case NT_ASLIDER_INITIAL_TOUCH:
    break;
  case NT_ASLIDER_MOVEMENT:
    break;
  case NT_ASLIDER_ALL_RELEASE:
    break;
  default:
    break;
  }

  /* Recalculate and set RGB LED */
  brightness_global = position;
  SetHueBrightness(hue_angle_global, brightness_global);
}

static void arotary_callback(const struct nt_control *control,
                             enum nt_control_arotary_event event,
                             uint32_t position)
{
  switch (event) {
    case NT_AROTARY_MOVEMENT:
      break;
    case NT_AROTARY_ALL_RELEASE:
      break;
    case NT_AROTARY_INITIAL_TOUCH:
      break;
    default:
      break;
  }

  /* Recalculate and set RGB LED */
  hue_angle_global = position;
  SetHueBrightness(hue_angle_global, brightness_global);
}

/* Call on the TSI CNTR overflow 16-bit range (65535) */
void system_callback(uint32_t event)
{
  switch(event)
  {
    case NT_SYSTEM_EVENT_DATA_OVERFLOW:
      /* TSI counter overflow warning */
      nt_printf("TSI CNTR overflow warning on electrode %d.\n", (int)index);
      break;
    default:
      break;
  }
}
```

```
static void SetHueBrightness(uint32_t hue_angle, uint32_t brightness)
{
  /* Calculate Hue */
  if(hue_angle<=24)
  {
    duty_cycle_red   = 24-hue_angle;
    duty_cycle_green = hue_angle;
    duty_cycle_blue  = 0;
  }
  if(hue_angle>24 && hue_angle<=48)
  {
    duty_cycle_red   = 0;
    duty_cycle_green = 48-hue_angle;
    duty_cycle_blue  = hue_angle-24;
  }
  if(hue_angle>48)
  {
    duty_cycle_red   = hue_angle-48;
    duty_cycle_green = 0;
    duty_cycle_blue  = 72-hue_angle;
  }

  /* If hue_angle > 72, set RGB to white */
  if(hue_angle > 72)
  {
    duty_cycle_red   = 16;
    duty_cycle_green = 16;
    duty_cycle_blue  = 16;
  }

  /* Slider never returns position < 60 */
  brightness = brightness<60 ? 0 : (brightness-60);

  /* Set Hue multiplied by Brightness */
  FTM0->CONTROLS[kFTM_Chnl_0].CnV = (duty_cycle_blue * (brightness));
  FTM0->CONTROLS[kFTM_Chnl_1].CnV = (duty_cycle_green * (brightness));
  FTM0->CONTROLS[kFTM_Chnl_2].CnV = (duty_cycle_red * (brightness));

  /* Software trigger to update registers */
  FTM_SetSoftwareTrigger(FTM0, true);
}

static void init_freemaster_uart(void)
{

    lpuart_config_t lpuartConfig;

    CLOCK_SetIpSrc(kCLOCK_Lpuart1, BOARD_DEBUG_UART_CLKSRC);
    uint32_t uartClkSrcFreq = CLOCK_GetIpFreq(kCLOCK_Lpuart1);

    /* UART configuration structure */
    LPUART_GetDefaultConfig(&lpuartConfig);
    lpuartConfig.baudRate_Bps     = BOARD_DEBUG_UART_BAUDRATE;
    lpuartConfig.enableTx         = true;
    lpuartConfig.enableRx         = true;

    /* Init UART */
    LPUART_Init((LPUART_Type *)BOARD_DEBUG_UART_BASEADDR, &lpuartConfig, uartClkSrcFreq);

    /* Enable UART interrupt */
    EnableIRQ(LPUART1_IRQn);

    /* The base address is configured dynamically in this application, making the code slightly
       bigger/slower but more generic. See freemaster_cfg.h for more details */
    /*   The base address is offset by 0x10 due to some new regs in the KE15Z LPUART. */
    FMSTR_SetSciBaseAddress((FMSTR_ADDR)(LPUART_Type *)(BOARD_DEBUG_UART_BASEADDR + 0x10));
}
```

```
void BOARD_UART_IRQ_HANDLER(void)
{
    FMSTR_Isr();
}
```

Chapter 7
Example

Overview

Provided example

- FRDM touch-sensing Demo for FRDM-KE15Z and FRDM-KE16Z.
- FRDM touch-sensing Demo for FRDM-K32L2A4S.
- DK touch sensing Demo for QN908xCDK.

Supported boards

- FRDM-KE15Z.
- FRDM-KE16Z.
- FRDM-K32L2A4S.
- DK-QN9080.

7.1 FRDM touch-sensing Demo for FRDM-KE15Z and FRDM-KE16Z

This demo is intended for the FRDM-KE15Z and FRDM-KE16Z base boards combined with the FRDM-TOUCH plug-in board. This application can work without the FRDM-TOUCH board (two electrodes only), requiring to add the definition #define NT_FRDM_TOUCH_SUPPORT 0 into the main.h file. The two touch electrodes (placed directly on the FRDM-KE15Z or FRDM-KE16Z boards) are used to set the LED to the purple or azure colors. There is an RGB LED controlled by controls from the FRDM-TOUCH board or from the FRDM-KE15Z (FRDM-KE16Z) board (four plus two-button keypad, rotary and slider). The rotary controls the LED's hue (color) and its electrodes use the self-capacitance measurement. The slider controls the LED's brightness and its electrodes use the self-capacitance measurement. The buttons set the LED to the red/green/blue/white colors with full brightness and their electrodes use mutual capacity sensing. The slider, rotary, button state, position, and movement direction can be monitored by the FreeMASTER GUI in the "Controls" tab.

Figure 7.1.1: FRDM-KE15Z (FRDM-KE16Z) + FRDM-TOUCH example app

7.2 FRDM touch-sensing Demo for FRDM-K32L2A4S

This demo is intended for the FRDM-K32L2A4S boards. The application for Freedom boards with on-board analog slider is created by two electrodes. The on-board LED is turned on when the touch event is detected, and it is turned off when the nger release event is detected. The LED toggles when the nger moves on the analog slider, showing that the position is being changed. The color of the LED indicates the slider position and is changing from red (low position detected) through yellow, green and purple to blue (high position detected) color.

Figure 7.2.1: FRDM-K32L2A4S example app

7.3 DK touch sensing Demo for QN908xCDK

This demo is intended for the NXP QN908xCDK board combined with the Touch Sense board V1 plug-in board. There is an RGB LED controlled by the Touch Sense board V1 controls (four button keypad, rotary, and slider). The rotary and slider control the red LED's brightness. The "up" keypad button switches on the blue LED, the right or left buttons toggle the blue LED and the down button switches off the blue LED. The Touch Sense board V1 and the example enable the switching between the touch elements (by default, only the rotary and keypad are activated). The slider and circle button keypad can be activated by the following configuration:

```
                     switch       elements
Rotary + Keypad:     SW2:  1-OFF, 2-OFF, 3-ON,  4-ON
                     SW3:  1-ON,  2-ON,  3-OFF, 4-OFF
                     SW4:  1-ON,  2-ON,  3-OFF, 4-OFF
                     SW5:  1-ON,  2-ON,  3-OFF, 4-OFF

                     switch       elements
Slider + CS Keypad:  SW2:  1-ON,  2-ON,  3-OFF, 4-OFF
                     SW3:  1-OFF, 2-OFF, 3-ON,  4-ON
                     SW4:  1-OFF, 2-OFF, 3-ON,  4-ON
                     SW5:  1-OFF, 2-OFF, 3-ON,  4-ON
```

All the electrodes in the Capacitive Sense module use the self-capacitance measurement. The slider, rotary, button state, position, and movement direction can be monitored by the FreeMASTER GUI application in the "Controls" tab.

Figure 7.3.1: Dk-QN9080 + FRDM-TOUCH example app

Chapter 8
NXP Touch User API

8.1 Overview

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

Collaboration diagram for NXP Touch User API:

Modules

- Controls
- Electrodes
- Filters
- Key Detectors
- Modules
- Safety
- System

- General Types
- CS Drivers
- TSI Drivers
- SafetyGPIO

8.1.1 Analog Rotary Control

8.1.1.1 Overview

Analog Rotary enables the detection of jog-dial-like finger movement using three or more electrodes; it is represented by the nt_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 8.1.1: Analog Rotary Electrodes

Collaboration diagram for Analog Rotary Control:

Modules

- Analog Rotary Control API

Data Structures

- struct nt_control_arotary

Macros

- #define NT_CONTROL_AROTARY_NAME

Typedefs

- typedef void(∗ nt_control_arotary_callback)(const struct nt_control ∗control, enum nt_control_-arotary_event event, uint32_t position)

Enumerations

- enum nt_control_arotary_event {
  NT_AROTARY_MOVEMENT,
  NT_AROTARY_ALL_RELEASE,
  NT_AROTARY_INITIAL_TOUCH }

Variables

- struct nt_control_interface nt_control_arotary_interface

8.1.1.2 Data Structure Documentation

8.1.1.2.1 struct nt_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 nt_control_arotary:

Data Fields

uint8_t	range	Range.

8.1.1.3 Macro Definition Documentation

8.1.1.3.1 #define NT_CONTROL_AROTARY_NAME

8.1.1.4 Typedef Documentation

8.1.1.4.1 typedef void(∗ nt_control_arotary_callback)(const struct nt_control ∗control, enum nt_control_arotary_event event, uint32_t position)

Analog Rotary event callback function pointer type.

8.1.1.5 Enumeration Type Documentation

8.1.1.5.1 enum nt_control_arotary_event

Analog Rotary event types.

Enumerator

NT_AROTARY_MOVEMENT Finger movement event.
NT_AROTARY_ALL_RELEASE Release event.
NT_AROTARY_INITIAL_TOUCH Initial-touch event.

8.1.1.6 Variable Documentation

8.1.1.6.1 struct nt_control_interface nt_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 nt_control to define the control behaviour. Can't be NULL.

8.1.1.7 Analog Rotary Control API

8.1.1.7.1 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 nt_electrode  * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*    &electrode_2, &electrode_3, NULL};
*
*  // Define additional parameters of the Analog Rotary
*  const struct nt_control_arotary my_arotary_params =
*  {
*   .range = 255,
*  };
*
*  // Definition of the Analog Rotary control
*  const struct nt_control my_arotary_control =
*  {
*    .interface = &nt_control_arotary_control_interface,
*    .electrodes = control_0_electrodes,
*    .control_params.arotary = &my_arotary_params,
*  };
*
*
```

Collaboration diagram for Analog Rotary Control API:

Functions

- void nt_control_arotary_register_callback (const struct nt_control ∗control, nt_control_arotary_-callback callback)
  Registers the Analog Rotary events handler function.
- uint32_t nt_control_arotary_get_position (const struct nt_control ∗control)
  Get the Analog Rotary 'Position' value.
- uint32_t nt_control_arotary_is_touched (const struct nt_control ∗control)
  Get 'Touched' state.
- uint32_t nt_control_arotary_movement_detected (const struct nt_control ∗control)
  Get 'Movement' flag.
- uint32_t nt_control_arotary_get_direction (const struct nt_control ∗control)
  Get 'Direction' flag.
- uint32_t nt_control_arotary_get_invalid_position (const struct nt_control ∗control)
  Returns invalid position flag.

8.1.1.7.2 Function Documentation

8.1.1.7.2.1 uint32_t nt_control_arotary_get_direction (const struct nt_control * *control*)

Parameters

control	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 = nt_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:

8.1.1.7.2.2 uint32_t nt_control_arotary_get_invalid_position (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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.");
*
```

Here is the call graph for this function:

8.1.1.7.2.3 uint32_t nt_control_arotary_get_position (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

Position. The returned value is in the range of zero to the maximum value configured in the nt_-control structure.

This function retrieves the actual finger position value. Example:

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

Here is the call graph for this function:

8.1.1.7.2.4 uint32_t nt_control_arotary_is_touched (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

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

```
* uint32_t touched;
* // Get state of arotary control
* touched = nt_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:

8.1.1.7.2.5 uint32_t nt_control_arotary_movement_detected (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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.);
*
```

Here is the call graph for this function:

8.1.1.7.2.6 void nt_control_arotary_register_callback (const struct nt_control * *control,* nt_control_arotary_callback *callback*)

Parameters

control	Pointer to the control.
callback	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 nt_control *control,
*                            enum nt_control_arotary_event event,
*                            uint32_t position)
*  {
*    (void)control;
*    char* event_names[] =
*     {
*      "NT_AROTARY_MOVEMENT",
*      "NT_AROTARY_ALL_RELEASE",
*      "NT_AROTARY_INITIAL_TOUCH",
*      };
*    printf("New analog rotary control event %s on position: %d.", event_names[event], position);
*  }
*
*  // register the callback function for arotary
*  nt_control_arotary_register_callback(&my_arotary_control,
      my_arotary_cb);
*
```

Here is the call graph for this function:

8.1.2 Analog Slider Control

8.1.2.1 Overview

Analog Slider enables detection of linear finger movement using two or more electrodes; it is represented by the nt_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 8.1.2: Analog Slider Electrodes

Collaboration diagram for Analog Slider Control:

Modules

- Analog Slider Control API

Data Structures

- struct nt_control_aslider

Macros

- #define NT_CONTROL_ASLIDER_NAME

Typedefs

- typedef void(∗ nt_control_aslider_callback)(const struct nt_control ∗control, enum nt_control_-aslider_event event, uint32_t position)

Enumerations

- enum nt_control_aslider_event {
  NT_ASLIDER_MOVEMENT,
  NT_ASLIDER_ALL_RELEASE,
  NT_ASLIDER_INITIAL_TOUCH }

Variables

- struct nt_control_interface nt_control_aslider_interface

8.1.2.2 Data Structure Documentation

8.1.2.2.1 struct nt_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 nt_control_aslider:

Data Fields

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

8.1.2.3 Macro Definition Documentation

8.1.2.3.1 #define NT_CONTROL_ASLIDER_NAME

8.1.2.4 Typedef Documentation

8.1.2.4.1 typedef void(∗ nt_control_aslider_callback)(const struct nt_control ∗control, enum nt_control_aslider_event event, uint32_t position)

Analog Slider event callback function pointer type.

8.1.2.5 Enumeration Type Documentation

8.1.2.5.1 enum nt_control_aslider_event

Analog Slider event types.

Enumerator

NT_ASLIDER_MOVEMENT Finger movement event
NT_ASLIDER_ALL_RELEASE Release event
NT_ASLIDER_INITIAL_TOUCH Initial-touch event

8.1.2.6 Variable Documentation

8.1.2.6.1 struct nt_control_interface nt_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 nt_control_aslider to define the control behaviour.

8.1.2.7 Analog Slider Control API

8.1.2.7.1 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 nt_electrode  * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*    NULL};
*
*  // Define additional parameters of Analog Slider
*  const struct nt_control_aslider my_aslider_params =
*  {
*   .range = 100,
*  };
*
*  // Definition of the Analog Slider control
*  const struct nt_control my_aslider_control =
*  {
*    .interface = &nt_control_aslider_control_interface,
*    .electrodes = control_0_electrodes,
*    .control_params.aslider = &my_aslider_params,
*  };
*
*
```

Collaboration diagram for Analog Slider Control API:

Functions

- void nt_control_aslider_register_callback (const struct nt_control ∗control, nt_control_aslider_-callback callback)
  Registers the Analog Slider events handler function.
- uint32_t nt_control_aslider_get_position (const struct nt_control ∗control)
  Get the Analog Slider 'Position' value.
- uint32_t nt_control_aslider_is_touched (const struct nt_control ∗control)
  Get 'Touched' state.
- uint32_t nt_control_aslider_movement_detected (const struct nt_control ∗control)
  Get 'Movement' flag.
- uint32_t nt_control_aslider_get_direction (const struct nt_control ∗control)
  Get 'Direction' flag.
- uint32_t nt_control_aslider_get_invalid_position (const struct nt_control ∗control)
  Returns invalid position flag.

8.1.2.7.2 Function Documentation

8.1.2.7.2.1 uint32_t nt_control_aslider_get_direction (const struct nt_control * *control*)

Parameters

control	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 = nt_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:

8.1.2.7.2.2 uint32_t nt_control_aslider_get_invalid_position (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_control_aslider_get_invalid_position(&
      my_aslider_control);
* 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:

8.1.2.7.2.3 uint32_t nt_control_aslider_get_position (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

Position. The returned value is in the range of zero to maximum value configured in the nt_control-_aslider structure.

This function retrieves the actual finger position value. Example:

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

Here is the call graph for this function:

8.1.2.7.2.4 uint32_t nt_control_aslider_is_touched (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

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

```
* uint32_t touched;
* // Get state of aslider control
* touched = nt_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:

8.1.2.7.2.5 uint32_t nt_control_aslider_movement_detected (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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.);
*
```

Here is the call graph for this function:

8.1.2.7.2.6 void nt_control_aslider_register_callback (const struct nt_control * *control,* nt_control_aslider_callback *callback*)

Parameters

control	Pointer to the control.
callback	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 nt_control *control,
*                            enum nt_control_aslider_event event,
*                            uint32_t position)
*  {
*    (void)control;
*    char* event_names[] =
*     {
*      "NT_ASLIDER_MOVEMENT",
*      "NT_ASLIDER_ALL_RELEASE",
*      "NT_ASLIDER_INITIAL_TOUCH",
*      };
     printf("New analog slider control event %s on position: %d.", event_names[event], position);
   }

   // register the callback function for aslider movement
   nt_control_aslider_register_callback(&my_aslider_control,
      my_aslider_cb);
```

Here is the call graph for this function:

8.1.3 Keypad Control

8.1.3.1 Overview

Keypad implements the keyboard-like functionality on top of an array of electrodes; it is represented by the nt_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 enables to detect so called multi-touch event when more keys were touched simultaneously. The key combinations consist of two or more "key press" events together can be defined by user for the group or single electrode based keypad.

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

The figures below show simple and grouped Keypad electrode layouts.

Figure 8.1.3: Keypad Electrodes

Collaboration diagram for Keypad Control:

Modules

- Keypad Control API

Data Structures

- struct nt_control_keypad

Macros

- #define NT_CONTROL_KEYPAD_NAME

Typedefs

- typedef void(∗ nt_control_keypad_callback)(const struct nt_control ∗control, enum nt_control_keypad_event event, uint32_t index)

Enumerations

- enum nt_control_keypad_event {
  NT_KEYPAD_RELEASE,
  NT_KEYPAD_TOUCH,
  NT_KEYPAD_AUTOREPEAT,
  NT_KEYPAD_MULTI_TOUCH }

Variables

- struct nt_control_interface nt_control_keypad_interface

8.1.3.2 Data Structure Documentation

8.1.3.2.1 struct nt_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.

Collaboration diagram for nt_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. Enable key groups feature.
uint8_t	groups_size	Number of groups.
uint32_t const *	multi_touch	Key combination for multitouch, depending on the key representation by one or more electrodes. Enable muti-touch feature.
uint8_t	multi_touch_-size	Number of multi-touch key combination.

8.1.3.3 Macro Definition Documentation

8.1.3.3.1 #define NT_CONTROL_KEYPAD_NAME

8.1.3.4 Typedef Documentation

8.1.3.4.1 typedef void(∗ nt_control_keypad_callback)(const struct nt_control ∗control, enum nt_control_keypad_event event, uint32_t index)

Keypad event callback function pointer type.

8.1.3.5 Enumeration Type Documentation

8.1.3.5.1 enum nt_control_keypad_event

Keypad event types.

Enumerator

NT_KEYPAD_RELEASE Release event
NT_KEYPAD_TOUCH Key-touch event
NT_KEYPAD_AUTOREPEAT Auto-repeat event
NT_KEYPAD_MULTI_TOUCH Key-multi touch event

8.1.3.6 Variable Documentation

8.1.3.6.1 struct nt_control_interface nt_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 nt_control_keypad, to define the control behaviour.

8.1.3.7 Keypad Control API

8.1.3.7.1 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 nt_electrode  * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*    &electrode_2, &electrode_3, NULL};
*
*  const struct nt_control_keypad keypad_params =
*  {
*     .groups = NULL,
*     .groups_size = 0,
*  };
*
*  // Definition of rotary control
*  const struct nt_control my_keypad_control =
*  {
*    .interface = &nt_control_keypad_control_interface,
*    .electrodes = control_0_electrodes,
*    .control_params.keypad = keypad_params,
*  };
*
*
```

Collaboration diagram for Keypad Control API:

Functions

- void nt_control_keypad_only_one_key_valid (const struct nt_control ∗control, uint32_t enable)
  Enable or disable the functionality that only one key press is valid.
- void nt_control_keypad_register_callback (const struct nt_control ∗control, nt_control_keypad_-callback callback)
  Registers the Keypad event handler function.
- void nt_control_keypad_set_autorepeat_rate (const struct nt_control ∗control, uint32_t value, uint32_t start_value)
  Set the auto-repeat rate.
- uint32_t nt_control_keypad_get_autorepeat_rate (const struct nt_control ∗control)
  Get the auto-repeat rate.
- uint32_t nt_control_keypad_is_button_touched (const struct nt_control ∗control, uint32_t index)
  Get the button touch status.

8.1.3.7.2 Function Documentation

8.1.3.7.2.1 uint32_t nt_control_keypad_get_autorepeat_rate (const struct nt_control * *control*)

Parameters

control	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 = nt_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:

8.1.3.7.2.2 uint32_t nt_control_keypad_is_button_touched (const struct nt_control ∗ *control,* uint32_t *index*)

Parameters

control	Pointer to the Keypad control.
index	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 = nt_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:

8.1.3.7.2.3 void nt_control_keypad_only_one_key_valid (const struct nt_control ∗ *control,* uint32_t *enable*)

Parameters

control	Pointer to the control.
enable	enable the only one key pressed is valid.

Returns

none

Enable or Disable the only one key press is valid at once. The behaviour 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
*  nt_control_keypad_only_one_key_valid(&my_keypad_control, 0);
*
```

Here is the call graph for this function:

8.1.3.7.2.4 void nt_control_keypad_register_callback (const struct nt_control ∗ *control,* nt_control_keypad_callback *callback*)

Parameters

control	Pointer to the control.
callback	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 nt_control *control,
*                           enum nt_control_keypad_event event,
*                           uint32_t index)
*  {
*    (void)control;
*     char* event_names[] =
*     {
*      "NT_KEYPAD_RELEASE",
*      "NT_KEYPAD_TOUCH",
*      "NT_KEYPAD_AUTOREPEAT",
*      };
*
*    printf("New keypad control event %s on key: %d.", event_names[event], index);
*  }
*
*  // register the callback function for keypad
*  nt_control_keypad_register_touch_callback(&my_keypad_control, my_keypad_touch_cb);
*
```

Here is the call graph for this function:

8.1.3.7.2.5 void nt_control_keypad_set_autorepeat_rate (const struct nt_control ∗ *control,* uint32_t *value,* uint32_t *start_value*)

Parameters

control	Pointer to the Keypad control.
value	Auto-repeat value. Value 0 disables the auto-repeat feature.
start_value	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
*  nt_control_keypad_set_autorepeat_rate(&my_keypad_control, 100, 1000
      );
*
```

Here is the call graph for this function:

8.1.4 Matrix Control

8.1.4.1 Overview

Matrix enables the one or two finger position and gestures detection on the touchpad surface. Collaboration diagram for Matrix Control:

Modules

- Matrix Control API

Data Structures

- struct nt_coordinates8_xy
- struct nt_coordinates64_xy
- struct nt_control_matrix

Macros

- #define NT_CONTROL_MATRIX_NAME

Typedefs

- typedef nt_coordinates8_xy nt_points8 [2]
- typedef void(∗ nt_control_matrix_callback)(const struct nt_control ∗control, enum nt_control_-matrix_event event, nt_coordinates8_xy ∗position, uint32_t index)
- typedef nt_coordinates64_xy nt_points64 [2]

Enumerations

- enum nt_control_matrix_event {
  NT_MATRIX_RELEASE,
  NT_MATRIX_TOUCH,
  NT_MATRIX_MOVEMENT,
  NT_MATRIX_RELEASE_SET_GESTURE,
  NT_MATRIX_RELEASE_GET_GESTURE,
  NT_MATRIX_GESTURE_TOO_LONG,
  NT_MATRIX_GET_CONTROL_GESTURE }
- enum nt_control_matrix_control_gestures {
  NT_MATRIX_NO_GESTURE,
  NT_MATRIX_DOUBLE_TAP,
  NT_MATRIX_COVER,
  NT_MATRIX_SWIPE_UP,
  NT_MATRIX_SWIPE_DOWN,
  NT_MATRIX_SWIPE_LEFT,
  NT_MATRIX_SWIPE_RIGHT,
  NT_MATRIX_ZOOM_IN,
  NT_MATRIX_ZOOM_OUT }

Variables

- struct nt_control_interface nt_control_matrix_interface

8.1.4.2 Data Structure Documentation

8.1.4.2.1 struct nt_coordinates8_xy

The Matrix RAM structure used to store x and y axis data as group.

Collaboration diagram for nt_coordinates8_xy:

Data Fields

uint8_t	x	x axis
uint8_t	y	y axis

8.1.4.2.2 struct nt_coordinates64_xy

The Matrix RAM structure used to store x and y axis data as group.

Collaboration diagram for nt_coordinates64_xy:

Data Fields

uint64_t	x	x axis
uint64_t	y	y axis

8.1.4.2.3 struct nt_control_matrix

The main structure representing the Matrix Control.

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

Collaboration diagram for nt_control_matrix:

Data Fields

uint16_t	dissimilarity_-max	Maximum difference for unlock gesture recognition
uint32_t	double_tap_-interval	Max. time interval between taps for double tap.
nt_points64 ∗	gesture	Stored matrix gesture array. The gesture means the time change of fingers x and y coordinates written in hexadecimal format from right to left. For example, x coordinate gesture 0x4321 means that firstly x position 1 was touched, then x position 2, 3 and 4. The gesture array must be finished by last zero member {{0,0},{0,0}}},
uint8_t	increase_-resolution	False value means default resolution, true value means that two the nearest touched electrodes create addition position value, it extends resolution according: new_resolution = 2 ∗ default_resolution - 1.
uint8_t	sensing_mode	False value means touchpad is based on self-capacitance sensing mode, true value means touchpad is based on mutual-capacitance sensing mode.

uint8_t	touchpad_size-_horizontal	Number of touchpad segments on x axis (horizontal segments). First touchpad_size_horizontal number of electrodes assigned to the matrix control are horizontal electrodes.
uint8_t	touchpad_size-_vertical	Number of touchpad segments on y axis (vertical segments).

8.1.4.3 Macro Definition Documentation

8.1.4.3.1 #define NT_CONTROL_MATRIX_NAME

8.1.4.4 Typedef Documentation

8.1.4.4.1 typedef void(∗ nt_control_matrix_callback)(const struct nt_control ∗control, enum nt_control_matrix_event event, nt_coordinates8_xy ∗position, uint32_t index)

Matrix event callback function pointer type.

8.1.4.4.2 typedef nt_coordinates64_xy nt_points64[2]

The Matrix RAM structure used to store x and y axis data of few points.

8.1.4.4.3 typedef nt_coordinates8_xy nt_points8[2]

The Matrix RAM structure used to store x and y axis data of few points.

8.1.4.5 Enumeration Type Documentation

8.1.4.5.1 enum nt_control_matrix_control_gestures

Matrix control gesture event types.

Enumerator

NT_MATRIX_NO_GESTURE No control gesture recognized
NT_MATRIX_DOUBLE_TAP Double tap control gesture
NT_MATRIX_COVER Covered control gesture
NT_MATRIX_SWIPE_UP Swipe up control gesture
NT_MATRIX_SWIPE_DOWN Swipe down control gesture
NT_MATRIX_SWIPE_LEFT Swipe left control gesture
NT_MATRIX_SWIPE_RIGHT Swipe right control gesture
NT_MATRIX_ZOOM_IN Zoom in control gesture
NT_MATRIX_ZOOM_OUT Zoom out control gesture

8.1.4.5.2 enum nt_control_matrix_event

Matrix event types.

Enumerator

NT_MATRIX_RELEASE Release event
NT_MATRIX_TOUCH Touch event
NT_MATRIX_MOVEMENT Movement event
NT_MATRIX_RELEASE_SET_GESTURE New unlock gesture is stored
NT_MATRIX_RELEASE_GET_GESTURE Stored unlock gesture is recognized
NT_MATRIX_GESTURE_TOO_LONG Unlock gesture is too long
NT_MATRIX_GET_CONTROL_GESTURE Recognized control gesture

8.1.4.6 Variable Documentation

8.1.4.6.1 struct nt_control_interface nt_control_matrix_interface

An interface structure, which contains pointers to the entry points of the Matrix algorithms. A pointer to this structure must be assigned to any instance of the nt_control_matrix, to define the control behaviour.

8.1.4.7 Matrix Control API

8.1.4.7.1 Overview

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:

```
* // Add following macros to nt_setup.h to configure array dimension
* #define NT_MATRIX_UNLOCK_GESTURES  4
*
*  // definition of electrode array used by control (more info in electrodes)
*  const struct nt_electrode  * const control_0_electrodes[] = {&electrode_0, &electrode_1,&
      electrode_2, &electrode_3,
*    &electrode_4,&electrode_5,&electrode_6,&electrode_7,&electrode_8,&electrode_9,&electrode_10,&
      electrode_11, NULL};
*
*  const struct nt_control_matrix matrix_params =
*  {
*     .double_tap_interval     = 500,
*     .dissimilarity_max       = 5,
*     .touchpad_size_horizontal= 6,
*     .touchpad_size_vertical  = 6,
*     .sensing_mode            = false,
*     .increase_resolution     = true,
*     .gesture = (nt_points64 [])){{{0x0BA9977555533211,0x0BBA987765443221},{0x0000000000000000,
      0x0000000000000000}},//
* gesture "\"
*
       {{0x0000001111223578,0x000000567889ABAA},{0x0000000000000000,0x0000000000000000}}, //
* gesture "O"
*
       {{0x00000BAAABBBBAA9,0x00000BBA98765422},{0x0000000000000000,0x0000000000000000}}, //
* gesture "Z"
*
       {{0x0022222222222211,0x00BABA987654322B},{0x009AAAAAAAAAAAA9,0x009AAAAAAAAAAAA9}}, //
* gesture "="
*                                {{0,0},{0,0}}},
*  };
*
*  // Definition of matrix control
*  const struct nt_control my_matrix_control =
*  {
*    .interface = &nt_control_matrix_control_interface,
*    .electrodes = control_0_electrodes,
*    .control_params.matrix = matrix_params,
*  };
*
*
```

Collaboration diagram for Matrix Control API:

Functions

- void nt_control_matrix_register_callback (const struct nt_control *control, nt_control_matrix_-callback callback)
  Registers the Matrix event handler function.
- int32_t nt_control_matrix_set_gesture (const struct nt_control *control, uint8_t gesture)
  Enable matrix control to store unlock gesture.

8.1.4.7.2 Function Documentation

8.1.4.7.2.1 void nt_control_matrix_register_callback (const struct nt_control * *control,* nt_control_matrix_callback *callback*)

Parameters

control	Pointer to the control.
callback	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 matrix
*  static void my_matrix_cb(const struct nt_control *control,
*                           enum nt_control_matrix_event event,
*                           nt_points8 *position,
*                           uint32_t index)
*  {
*    (void)control;
*     char* event_names[] =
*     {
*      "NT_MATRIX_RELEASE            ",
*      "NT_MATRIX_TOUCH              ",
*      "NT_MATRIX_MOVEMENT           ",
*      "NT_MATRIX_RELEASE_SET_GESTURE",
*      "NT_MATRIX_RELEASE_GET_GESTURE",
*      "NT_MATRIX_GESTURE_TOO_LONG   ",
*      "NT_MATRIX_GET_CONTROL_GESTURE",
*      };
*
*    printf("New matrix control event %s on key: %d.", event_names[event], index);
*  }
*
*  // register the callback function for matrix
*  nt_control_matrix_register_touch_callback(&my_matrix_control, my_matrix_touch_cb);
*
```

Here is the call graph for this function:

8.1.4.7.2.2 int32_t nt_control_matrix_set_gesture (const struct nt_control ∗ *control,* uint8_t *gesture*)

Parameters

control	Pointer to the Matrix control.
gesture	Gesture number to store.

Returns

result of operation nt_result.

In case that gesture parameter is lower or equal to the defined value in the NT_MATRIX_UNLOCK_GESTURES macro, the function set the select_gesture variable (select_gesture) to positive value corresponding gesture number to store after drawing and return NT_SUCCESS value. The function return the NT_FAILURE value if the input gesture number is the gesture higher then the NT_MATRIX_UNLOCK_GESTURES macro.

Example:

```
* // Want to store second unlock gesture (first unlock gesture has index 0).
* uint8_t gesture = 1;
* nt_control_matrix_set_gesture(&my_matrix_control, gesture);
*
```

Here is the call graph for this function:

8.1.5 Proxi Control

8.1.5.1 Overview

Proxi implements object presence detection in the near field (approaching finger or hand), it is represented by the nt_control_proxi structure.

The Proxi Control provides the proximity Key status values and is able to generate Proxi Touch, Movement, and Release events.

The figures below show simple and grouped Proxi electrode layouts.

Figure 8.1.4: Proxi Electrodes

Collaboration diagram for Proxi Control:

Modules

- Proxi Control API

Data Structures

- struct nt_control_proxi

Macros

- #define NT_CONTROL_PROXI_NAME

Typedefs

- typedef void(∗ nt_control_proxi_callback)(const struct nt_control ∗control, enum nt_control_proxi-_event event, uint32_t index_or_gesture, uint32_t proximity)

Enumerations

- enum nt_control_proxi_event {
  NT_PROXI_MOVEMENT,
  NT_PROXI_RELEASE,
  NT_PROXI_TOUCH,
  NT_PROXI_SET_GESTURE,
  NT_PROXI_GET_GESTURE }

Variables

- struct nt_control_interface nt_control_proxi_interface

8.1.5.2 Data Structure Documentation

8.1.5.2.1 struct nt_control_proxi

The main structure representing the Proxi Control.

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

Collaboration diagram for nt_control_proxi:

Data Fields

uint32_t ∗	gesture	Stored proxi gesture array. The gesture means the time change of activated proxi electrodes written in hexadecimal format from right to left. For example, gesture 0x321 means that firstly electrode 1 was touched, then electrode 2 and 3.
uint8_t	insensitivity	Insensitivity for the callbacks invokes when the position is changed within selected range.
uint8_t	range	Proxi scale (i.e. 0-100% or 0-255) value.
uint16_t	scale	Max signal delta level for max. prox (100%) value
uint8_t	threshold	Proxi Touch/Release threshold value from within selected range.

8.1.5.3 Macro Definition Documentation

8.1.5.3.1 #define NT_CONTROL_PROXI_NAME

8.1.5.4 Typedef Documentation

8.1.5.4.1 typedef void(∗ nt_control_proxi_callback)(const struct nt_control ∗control, enum nt_control_proxi_event event, uint32_t index_or_gesture, uint32_t proximity)

Proxi event callback function pointer type.

8.1.5.5 Enumeration Type Documentation

8.1.5.5.1 enum nt_control_proxi_event

Proxi event types.

Enumerator

NT_PROXI_MOVEMENT Release event
NT_PROXI_RELEASE Key-touch event
NT_PROXI_TOUCH Proximity movement event
NT_PROXI_SET_GESTURE Gesture recognized event event
NT_PROXI_GET_GESTURE Gesture stored event event

8.1.5.6 Variable Documentation

8.1.5.6.1 struct nt_control_interface nt_control_proxi_interface

An interface structure, which contains pointers to the entry points of the Proxi algorithms. A pointer to this structure must be assigned to any instance of the nt_control_proxi, to define the control behaviour.

8.1.5.7 Proxi Control API

8.1.5.7.1 Overview

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

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

```
* // Add following macros to nt_setup.h to configure array dimensions
* #define NT_PROXI_ELECTRODES 4
* #define NT_PROXI_GESTURES 3
*
*  // definition of electrode array used by control (more info in electrodes )
* const struct nt_electrode  * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*   NULL};
*
* const struct nt_control_proxi proxi_params =
* {
*   .range = 5000,
*   .scale = 100,
*   .threshold = 3,
*   .insensitivity = 2,
*   .gesture = (uint32_t []){0x0123, 0x3210, 0},
* };
*
*  // Definition of proxi control
* const struct nt_control proxi_0 =
* {
*   .interface = &nt_control_proxi_interface,
*   .electrodes = control_0_electrodes,
*   .control_params.proxi = &proxi_params,
* };
*
*
```

Collaboration diagram for Proxi Control API:

Functions

- void nt_control_proxi_register_callback (const struct nt_control ∗control, nt_control_proxi_-callback callback)
  Registers the Proxi event handler function.
- int32_t nt_control_proxi_set_gesture (const struct nt_control ∗control, uint8_t gesture)
  Enable proxi control to store unlock gesture.

8.1.5.7.2 Function Documentation

8.1.5.7.2.1 void nt_control_proxi_register_callback (const struct nt_control * *control,* nt_control_proxi_callback *callback*)

Parameters

control	Pointer to the control.
callback	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 proxi
*  static void my_proxi_cb(const struct nt_control *control,
*                          enum nt_control_proxi_event event,
*                          uint32_t index_or_gesture,
*                          uint32_t proximity)
*  {
*    (void)control;
*     char* event_names[] =
*     {
*      "NT_PROXI_MOVEMENT",
*      "NT_PROXI_RELEASE",
*      "NT_PROXI_TOUCH",
*      "NT_PROXI_GESTURE_RECOGNIZED",
*      };
*
*    printf("New proxi control event %s on key: %d.", event_names[event], index);
*  }
*
*  // register the callback function for proxi
*  nt_control_proxi_register_touch_callback(&my_proxi_control, my_proxi_touch_cb);
*
```

Here is the call graph for this function:

8.1.5.7.2.2 int32_t nt_control_proxi_set_gesture (const struct nt_control * *control,* uint8_t *gesture*)

Parameters

control	Pointer to the Proxi control.
gesture	Gesture number to store.

Returns

result of operation nt_result.

In case that gesture parameter is lower or equal to the defined value in the NT_PROXI_UNLOCK_GESTURES macro, the function set the select_gesture variable (select_gesture) to positive value corresponding gesture number to store after drawing and return NT_SUCCESS value. The function return the NT_FAILURE value if the input gesture number is the gesture higher then the NT_PROXI_UNLOCK_GESTURES macro.

Example:

```
* // Want to store second unlock gesture (first unlock gesture has index 0).
* uint8_t gesture = 1;
* nt_control_proxi_set_gesture(&my_proxi_control, gesture);
*
```

Here is the call graph for this function:

8.1.6 Rotary Control

8.1.6.1 Overview

The Rotary control enables the detection of jog-dial-like finger movement using discrete electrodes; it is represented by the nt_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 8.1.5: Rotary Electrodes

Collaboration diagram for Rotary Control:

Modules

- Rotary Control API

Macros

- #define NT_CONTROL_ROTARY_NAME

Typedefs

- typedef void(∗ nt_control_rotary_callback)(const struct nt_control ∗control, enum nt_control_-rotary_event, uint32_t position)

Enumerations

- enum nt_control_rotary_event {
  NT_ROTARY_MOVEMENT,
  NT_ROTARY_ALL_RELEASE,
  NT_ROTARY_INITIAL_TOUCH }

Variables

- struct nt_control_interface nt_control_rotary_interface

8.1.6.2 Macro Definition Documentation

8.1.6.2.1 #define NT_CONTROL_ROTARY_NAME

8.1.6.3 Typedef Documentation

8.1.6.3.1 typedef void(∗ nt_control_rotary_callback)(const struct nt_control ∗control, enum nt_control_rotary_event, uint32_t position)

Rotary event callback function pointer type.

8.1.6.4 Enumeration Type Documentation

8.1.6.4.1 enum nt_control_rotary_event

Rotary event types.

Enumerator

NT_ROTARY_MOVEMENT Finger movement event
NT_ROTARY_ALL_RELEASE Release event
NT_ROTARY_INITIAL_TOUCH Initial-touch event

8.1.6.5 Variable Documentation

8.1.6.5.1 struct nt_control_interface nt_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 nt_control_rotary_control to define the control behaviour.

8.1.6.6 Rotary Control API

8.1.6.6.1 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 nt_electrode  * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*    &electrode_2, &electrode_3, NULL};
*
*  // Definition of the Rotary control
*  const struct nt_control my_rotary_control =
*  {
*    .interface = &nt_control_rotary_interface,
*    .electrodes = control_0_electrodes,
*  };
*
*
```

Collaboration diagram for Rotary Control API:

Functions

- void nt_control_rotary_register_callback (const struct nt_control ∗control, nt_control_rotary_-callback callback)
  Registers the events handler function.
- uint32_t nt_control_rotary_get_position (const struct nt_control ∗control)
  Get the Rotary 'Position' value.
- uint32_t nt_control_rotary_is_touched (const struct nt_control ∗control)
  Get 'Touched' state.
- uint32_t nt_control_rotary_movement_detected (const struct nt_control ∗control)
  Get 'Movement' flag.
- uint32_t nt_control_rotary_get_direction (const struct nt_control ∗control)
  Get 'Direction' flag.
- uint32_t nt_control_rotary_get_invalid_position (const struct nt_control ∗control)
  Get 'Invalid' flag.

8.1.6.6.2 Function Documentation

8.1.6.6.2.1 uint32_t nt_control_rotary_get_direction (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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:

8.1.6.6.2.2 uint32_t nt_control_rotary_get_invalid_position (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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.");
*
```

Here is the call graph for this function:

8.1.6.6.2.3 uint32_t nt_control_rotary_get_position (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_control_rotary_get_position(&my_rotary_control);
* printf("Position of Rotary control is: %d.", position);
*
```

Here is the call graph for this function:

8.1.6.6.2.4 uint32_t nt_control_rotary_is_touched (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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:

8.1.6.6.2.5 uint32_t nt_control_rotary_movement_detected (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

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

```
* uint32_t movement;
* // Get state of rotary control
* movement = nt_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.);
*
```

Here is the call graph for this function:

8.1.6.6.2.6 void nt_control_rotary_register_callback (const struct nt_control * *control,* nt_control_rotary_callback *callback*)

Parameters

control	Pointer to the control.
callback	Address 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 nt_control *control,
*                           enum nt_control_arotary_event event,
*                           uint32_t position)
*  {
*    (void)control;
*    char* event_names[] =
*     {
*      "NT_ROTARY_MOVEMENT",
*      "NT_ROTARY_ALL_RELEASE",
*      "NT_ROTARY_INITIAL_TOUCH",
*      };
*    printf("New rotary control event %s on position: %d.", event_names[event], position);
*  }
*
*  // register the callback function for rotary
*  nt_control_rotary_register_callback(&my_rotary_control, my_rotary_cb)
      ;
*
```

Here is the call graph for this function:

8.1.7 Slider control

8.1.7.1 Overview

Slider control enables the detection of linear finger movement using discrete electrodes; it is represented by the nt_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 8.1.6: Slider Electrodes

Collaboration diagram for Slider control:

Modules

- Slider Control API

Macros

- #define NT_CONTROL_SLIDER_NAME

Typedefs

- typedef void(∗ nt_control_slider_callback)(const struct nt_control ∗control, enum nt_control_slider_event, uint32_t position)

Enumerations

- enum nt_control_slider_event {
  NT_SLIDER_MOVEMENT,
  NT_SLIDER_ALL_RELEASE,
  NT_SLIDER_INITIAL_TOUCH }

Variables

- struct nt_control_interface nt_control_slider_interface

8.1.7.2 Macro Definition Documentation

8.1.7.2.1 #define NT_CONTROL_SLIDER_NAME

8.1.7.3 Typedef Documentation

8.1.7.3.1 typedef void(∗ nt_control_slider_callback)(const struct nt_control ∗control, enum nt_control_slider_event, uint32_t position)

Slider event callback function pointer type.

8.1.7.4 Enumeration Type Documentation

8.1.7.4.1 enum nt_control_slider_event

Slider event types.

Enumerator

NT_SLIDER_MOVEMENT Finger movement event.
NT_SLIDER_ALL_RELEASE Release event.
NT_SLIDER_INITIAL_TOUCH Initial-touch event.

8.1.7.5 Variable Documentation

8.1.7.5.1 struct nt_control_interface nt_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 instace of nt_control_slider_control to define the control behaviour.

8.1.7.6 Slider Control API

8.1.7.6.1 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 nt_electrode  * const control_0_electrodes[] = {&electrode_0, &electrode_1,
*    NULL};
*
*  // Definition of the Slider control
*  const struct nt_control my_slider_control =
*  {
*    .interface = &nt_control_slider_interface,
*    .electrodes = control_0_electrodes,
*  };
*
*
```

Collaboration diagram for Slider Control API:

Functions

- void nt_control_slider_register_callback (const struct nt_control ∗control, nt_control_slider_-callback callback)
  Registers the events handler function.
- uint32_t nt_control_slider_get_position (const struct nt_control ∗control)
  Get the Slider 'Position' value.
- uint32_t nt_control_slider_is_touched (const struct nt_control ∗control)
  Get 'Touched' state.
- uint32_t nt_control_slider_movement_detected (const struct nt_control ∗control)
  Get 'Movement' flag.
- uint32_t nt_control_slider_get_direction (const struct nt_control ∗control)
  Get 'Direction' flag.
- uint32_t nt_control_slider_get_invalid_position (const struct nt_control ∗control)
  Get 'Invalid' flag.

8.1.7.6.2 Function Documentation

8.1.7.6.2.1 uint32_t nt_control_slider_get_direction (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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:

8.1.7.6.2.2 uint32_t nt_control_slider_get_invalid_position (const struct nt_control ∗ *control*)

Parameters

control	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 = nt_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.");
*
```

Here is the call graph for this function:

8.1.7.6.2.3 uint32_t nt_control_slider_get_position (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

Position. The returned value is in the range of zero to the maximum value configured in the nt_-control structure.

This function retrieves the actual finger position value. Example:

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

Here is the call graph for this function:

8.1.7.6.2.4 uint32_t nt_control_slider_is_touched (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

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

```
* uint32_t touched;
* // Get state of Slider control
* touched = nt_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:

8.1.7.6.2.5 uint32_t nt_control_slider_movement_detected (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control.

Returns

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

```
* uint32_t movement;
* // Get state of Slider control
* movement = nt_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.");
*
```

Here is the call graph for this function:

8.1.7.6.2.6 void nt_control_slider_register_callback (const struct nt_control * *control,* nt_control_slider_callback *callback*)

Parameters

control	Pointer to the control.
callback	Address 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 nt_control *control,
*                           enum nt_control_aslider_event event,
*                           uint32_t position)
*  {
*    (void)control;
*    char* event_names[] =
*     {
*      "NT_SLIDER_MOVEMENT",
*      "NT_SLIDER_ALL_RELEASE",
*      "NT_SLIDER_INITIAL_TOUCH",
*      };
*    printf("New slider control event %s on position: %d.", event_names[event], position);
*  }
*
*  // register the callback function for slider
*  nt_control_slider_register_callback(&my_slider_control, my_slider_cb)
      ;
*
```

Here is the call graph for this function:

8.2 Controls

8.2.1 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
- Proxi Control
- Rotary Control
- Slider control
- General API

8.2.2 General API

8.2.2.1 Overview

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

Modules

- API Functions

Data Structures

- union nt_control_params
- struct nt_control

8.2.2.2 Data Structure Documentation

8.2.2.2.1 union nt_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.

Collaboration diagram for nt_control_params:

Data Fields

struct nt_control_arotary *	arotary	Pointer to the Analog Rotary control parameters structure
struct nt_control_aslider *	aslider	Pointer to the Analog Slider control parameters structure
const void *	general	Pointer for general use
struct nt_control_keypad *	keypad	Pointer to the Keypad control parameters structure
struct nt_control_matrix *	matrix	Pointer to the Matrix control parameters structure
struct nt_control_proxi *	proxi	Pointer to the Proxi control parameters structure

8.2.2.2.2 struct nt_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 behaviour. Note that the "control_params" must correspond to the control type.

This structure can be allocated in ROM.

Collaboration diagram for nt_control:

Data Fields

union nt_-control_params	control_params	An instance of the control params. Cannot be NULL.
struct nt_electrode ∗const ∗	electrodes	List of electrodes. Cannot be NULL.
struct nt_control_-interface ∗	interface	An instance of the control interface. Cannot be NULL.

8.2.2.3 API Functions

8.2.2.3.1 Overview

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

Functions

- void nt_control_enable (const struct nt_control ∗control)
  Enable control.
- void nt_control_disable (const struct nt_control ∗control)
  Disable control.
- int32_t nt_control_get_touch_button (const struct nt_control ∗control, uint32_t index)
  Get touched electrode.
- uint64_t nt_control_get_electrodes_state (struct nt_control ∗control)
  Get the state of all control electrodes.
- uint32_t nt_control_count_electrodes (const struct nt_control ∗control)
- struct nt_electrode ∗ nt_control_get_electrode (const struct nt_control ∗control, uint32_t index)
  Return the electrode by index.

8.2.2.3.2 Function Documentation

8.2.2.3.2.1 uint32_t nt_control_count_electrodes (const struct nt_control ∗ *control*)

Here is the caller graph for this function:

8.2.2.3.2.2 void nt_control_disable (const struct nt_control ∗ *control*)

Parameters

control	Pointer to the control instance.

Returns

none

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

```
*  // The NT control my_nt_control_keypad is disabled
*  nt_control_disable(&my_nt_control_keypad);
*
```

Here is the call graph for this function:

8.2.2.3.2.3 void nt_control_enable (const struct nt_control * *control*)

Parameters

control	Pointer to the control instance.

Returns

none

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

```
*  // The NT control my_nt_control_keypad is enabled
*  nt_control_enable(&my_nt_control_keypad);
*
```

Here is the call graph for this function:

8.2.2.3.2.4 struct nt_electrode* nt_control_get_electrode (const struct nt_control * *control,* uint32_t *index*)

Parameters

control	Pointer to the control.
index	

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 NT library:

```
*  // Get the pointer of electrode on index 2 for my_control
*  nt_electrode *my_electrode = nt_control_get_electrode(&my_control, 2
      );
*
```

Here is the call graph for this function:

8.2.2.3.2.5 uint64_t nt_control_get_electrodes_state (struct nt_control * *control*)

Parameters

control	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.

```
*  uint64_t touched_electrode = 0;
*  touched_electrode = nt_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:

8.2.2.3.2.6 int32_t nt_control_get_touch_button (const struct nt_control ∗ *control,* uint32_t *index*)

Parameters

control	Pointer to the control.
index	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 NT_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 NT library:

```
*  int32_t last_touched_electrode = 0;
*  uint32_t electrode_count = nt_control_count_electrodes(&my_control);
*  last_touched_electrode = nt_control_get_touch_button(&my_control,
      last_touched_electrode);
*  while(last_touched_electrode != NT_FAILURE)
*  {
*    printf("The electrode %d in my control is touched", last_touched_electrode);
*    last_touched_electrode = nt_control_get_touch_button(&my_control,
      last_touched_electrode);
*  }
*
```

Here is the call graph for this function:

8.3 Electrodes

8.3.1 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 nt_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 nt_electrode_status
- struct gpio_inputs
- struct nt_electrode

Macros

- #define NT_ELECTRODE_STATUS_HISTORY_COUNT
- #define NT_ELECTRODE_SHIELD_THRESHOLD
- #define NT_ELECTRODE_SHIELD_GAIN
- #define NT_ELECTRODE_SHIELD_SENS
- #define NT_ELECTRODE_SHIELD_ATTN

Enumerations

- enum nt_electrode_state {
  NT_ELECTRODE_STATE_INIT,
  NT_ELECTRODE_STATE_RELEASE,
  NT_ELECTRODE_STATE_TOUCH }

8.3.2 Data Structure Documentation

8.3.2.1 struct nt_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 nt_electrode_status:

Data Fields

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

8.3.2.2 struct gpio_inputs

Collaboration diagram for gpio_inputs:

Data Fields

uint32_t	gpio1	
uint32_t	gpio2	

8.3.2.3 struct nt_electrode

The main structure representing the Electrode instance. There are all the parameters needed to define the behaviour of the NXP 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 nt_electrode:

Data Fields

uint16_t	baseline_level	typical Baseline level value used for safety failure detection
uint8_t	divider	Divider.
struct gpio_inputs	gpio_input	corresponding GPIO input pins
struct nt_keydetector-_interface *	keydetector_-interface	Pointer to Key Detector interface.

union nt_keydetector-_params	keydetector_-params	Pointer to Key Detector params.
uint8_t	multiplier	Multiplier.
uint32_t	pin_input	Input pin.
uint8_t	shield_attn	SH Attenuate for shielding el. signal
uint8_t	shield_gain	SH Gain for shielding el. signal
uint16_t	shield_sens	SH Sensitivity for shielding limitation
uint8_t	shield_-threshold	SH Threshold for shield activating
struct nt_electrode *	shielding_-electrode	Shielding electrode.

8.3.3 Macro Definition Documentation

8.3.3.1 #define NT_ELECTRODE_SHIELD_ATTN

Definition of the max signal value on the shield electrode used for offset compensation

8.3.3.2 #define NT_ELECTRODE_SHIELD_GAIN

Definition of the lowest signal value on the shield electrode used for offset compensation

8.3.3.3 #define NT_ELECTRODE_SHIELD_SENS

Definition of the max signal value on the shield electrode used for offset compensation

8.3.3.4 #define NT_ELECTRODE_SHIELD_THRESHOLD

Definition of the lowest signal value on the shield electrode used for offset compensation

8.3.3.5 #define NT_ELECTRODE_STATUS_HISTORY_COUNT

Definition of the electrode history status depth. Default setting is 4.

8.3.4 Enumeration Type Documentation

8.3.4.1 enum nt_electrode_state

Electrode states.

Enumerator

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

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

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

8.3.5 API Functions

8.3.5.1 Overview

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

Functions

- int32_t nt_electrode_enable (const struct nt_electrode ∗electrode, uint32_t touch)
  Enable the electrode. The function is used to enable the electrode; it should be used after the NT initialization, because the default state after the startup of the NT is electrode disabled.
- int32_t nt_electrode_resume (const struct nt_electrode ∗electrode)
  Resume the electrode after the Suspending (temp.disabled), without reseting the keydetector.
- int32_t nt_electrode_suspend (const struct nt_electrode ∗electrode)
  Suspend (temporary disable) the electrode.
- int32_t nt_electrode_disable (const struct nt_electrode ∗electrode)
  Disable the electrode.
- int32_t nt_electrode_reset (const struct nt_electrode ∗electrode)
  Reset the electrode to the current state.
- uint32_t nt_electrode_get_signal (const struct nt_electrode ∗electrode)
  Get the normalized and processed electrode signal.
- int32_t nt_electrode_get_last_status (const struct nt_electrode ∗electrode)
  Get the last known electrode status.
- uint32_t nt_electrode_get_time_offset (const struct nt_electrode ∗electrode)
  Get the time from the last electrode event.
- uint32_t nt_electrode_get_last_time_stamp (const struct nt_electrode ∗electrode)
  Get the last known electrode time stamp.
- uint32_t nt_electrode_get_raw_signal (const struct nt_electrode ∗electrode)
  Get the raw electrode signal.
- struct nt_electrode_data ∗ nt_electrode_get_data (const struct nt_electrode ∗electrode)
  Get electrode data structure pointer.
- uint32_t nt_electrode_is_enabled (const struct nt_electrode ∗electrode)
  Get electrode enabled status.
- struct nt_module_data ∗ nt_electrode_get_module_data (const struct nt_electrode ∗electrode)
  Get module data structure pointer.

8.3.5.2 Function Documentation

8.3.5.2.1 int32_t nt_electrode_disable (const struct nt_electrode * *electrode*)

Parameters

electrode	Pointer to the electrode params that identify the electrode.

Returns

result of operation nt_result. This is an example of using this function in code:

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

Here is the call graph for this function:

Here is the caller graph for this function:

8.3.5.2.2 int32_t nt_electrode_enable (const struct nt_electrode ∗ *electrode,* uint32_t *touch*)

Parameters

electrode	Pointer to the electrode params that identify the electrode.
touch	Default parameter if the electrode was touched or not after the enable process.

Returns

result of operation nt_result. This is an example of using this function in the code:

```
//The electrode that is defined in the setup of NT after the initialization must be enabled.
if (nt_init(&system_0, nt_memory_pool, sizeof(nt_memory_pool)) <
      NT_SUCCESS)
{
  while(1); // add code to handle this error
}
// Enable electrode_0 that is defined in the setup of NT
if(nt_electrode_enable(&electrode_0, 0) != NT_SUCCESS)
{
  printf("Enable electrode_0 failed.");
}
```

Here is the call graph for this function:

8.3.5.2.3 struct nt_electrode_data* nt_electrode_get_data (const struct nt_electrode * *electrode*)

Parameters

electrode	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.

Here is the call graph for this function:

Here is the caller graph for this function:

8.3.5.2.4 int32_t nt_electrode_get_last_status (const struct nt_electrode * *electrode*)

Parameters

electrode	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 = nt_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:

8.3.5.2.5 uint32_t nt_electrode_get_last_time_stamp (const struct nt_electrode ∗ *electrode*)

Parameters

electrode	Pointer to the electrode data.

Returns

Current electrode status.

```
*  // Get the time stamp of the last change of the electrode status
*  uint32_t time = nt_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:

8.3.5.2.6 struct nt_module_data∗ nt_electrode_get_module_data (const struct nt_electrode ∗ *electrode*)

Parameters

electrode	Pointer to the electrode user parameter structure.

Returns

The pointer to the module data 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:

8.3.5.2.7 uint32_t nt_electrode_get_raw_signal (const struct nt_electrode ∗ *electrode*)

Parameters

electrode	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.", nt_electrode_get_raw_signal(&
      my_electrode));
*
```

Here is the call graph for this function:

8.3.5.2.8 uint32_t nt_electrode_get_signal (const struct nt_electrode * *electrode*)

Parameters

electrode	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.", nt_electrode_get_signal(&my_electrode)
      );
*
```

Here is the call graph for this function:

8.3.5.2.9 uint32_t nt_electrode_get_time_offset (const struct nt_electrode * *electrode*)

Parameters

electrode	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 = nt_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:

8.3.5.2.10 uint32_t nt_electrode_is_enabled (const struct nt_electrode * *electrode*)

Parameters

electrode	Pointer to the electrode user parameter structure.

Returns

TRUE if the electrode is enabled, FALSE if disabled

Here is the call graph for this function:

8.3.5.2.11 int32_t nt_electrode_reset (const struct nt_electrode * *electrode*)

Parameters

electrode	Pointer to the electrode params that identify the electrode.

Returns

result of operation nt_result. This is an example of using this function in code:

```
*  // Reset electrode_0 that is defined in the setup of NT
*  if(nt_electrode_reset(&electrode_0) != NT_SUCCESS)
*  {
*    printf("reset electrode_0 failed.");
*  }
*
```

Here is the call graph for this function:

8.3.5.2.12 int32_t nt_electrode_resume (const struct nt_electrode * *electrode*)

Parameters

electrode	Pointer to the electrode params that identify the electrode.

Returns

result of operation nt_result. This is an example of using this function in the code:

```
// re-enable electrode_0 that is defined in the setup of NT
if(nt_electrode_resume(&electrode_0) != NT_SUCCESS)
{
  printf("Enable electrode_0 failed.");
}
```

Here is the call graph for this function:

8.3.5.2.13 int32_t nt_electrode_suspend (const struct nt_electrode * *electrode*)

Parameters

electrode	Pointer to the electrode params that identify the electrode.

Returns

result of operation nt_result. This is an example of using this function in code:

```
*  // Suspend electrode_0 that is defined in the setup of NT
*  if(nt_electrode_suspend(&electrode_0) != NT_SUCCESS)
*  {
*    printf("Suspending electrode_0 failed.");
*  }
*
```

Here is the call graph for this function:

8.4 Filters

8.4.1 Overview

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

Data Structures

- struct nt_filter_fbutt
- struct nt_filter_iir
- struct nt_filter_dctracker
- struct nt_filter_moving_average

Macros

- #define NT_FILTER_MOVING_AVERAGE_MAX_ORDER

8.4.2 Data Structure Documentation

8.4.2.1 struct nt_filter_fbutt

The butterworth filter input parameters.

Collaboration diagram for nt_filter_fbutt:

Data Fields

int32_t	cutoff	The coefficient for the implemented butterworth filter polynomial.

8.4.2.2 struct nt_filter_iir

The IIR filter input parameters.

Collaboration diagram for nt_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.

8.4.2.3 struct nt_filter_dctracker

The DC tracker filter input parameters.

Collaboration diagram for nt_filter_dctracker:

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.

8.4.2.4 struct nt_filter_moving_average

The moving average filter input parameters.

Collaboration diagram for nt_filter_moving_average:

Data Fields

int32_t	n2_order	The order 2^n moving average filter

8.4.3 Macro Definition Documentation

8.4.3.1 #define NT_FILTER_MOVING_AVERAGE_MAX_ORDER

8.4.4 Advanced Filtering and Integrating Detection

8.4.4.1 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 nt_keydetector_afid_asc
- struct nt_keydetector_afid

Macros

- #define NT_KEYDETECTOR_AFID_ASC_DEFAULT

Variables

- struct nt_keydetector_interface nt_keydetector_afid_interface

8.4.4.2 Data Structure Documentation

8.4.4.2.1 struct nt_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 NT_KEYDETECTOR_AFID_ASC_DEFAULT. This structure must be filled in.

Collaboration diagram for nt_keydetector_afid_asc:

Data Fields

uint32_t	noise_resets_-minimum	Noise Resets Minimum. If touch threshold is lower then noise reset minimum (low sensitivity) the touch threshold will be limited to noise level minimum value.
int16_t	resets_for_-touch	Number of resets required for touch. The low number causes quicker touch event recognition, high number causes longer integration process.
int16_t	touch_-threshold_-fall_rate	Rate of how often the touch threshold can fall. The value represent the time period after each the touch threshold will fall down if touch threshold is higher then destination threshold

8.4.4.2.2 struct nt_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 nt_keydetector_afid:

Data Fields

struct nt_keydetector-_afid_asc	asc	Automatic Sensitive Calibration structure for the AFID detector.
struct nt_filter-_moving_-average	base_avrg	Settings of the moving average filter for the baseline in release state of electrode.
struct nt_filter_fbutt	fast_signal_-filter	Signal butterworth signal (fast).
uint16_t	reset_rate	Reset rate of the AFID. Cannot be 0. It can clear integration value to avoid the long-term temperature drift
struct nt_filter_iir	signal_filter	Coefficient of the input IIR signal filter, used to suppress high frequency noise.
struct nt_filter_fbutt	slow_signal_-filter	Baseline butterworth signal (slow).

8.4.4.3 Macro Definition Documentation

8.4.4.3.1 #define NT_KEYDETECTOR_AFID_ASC_DEFAULT

AFID Automatic Sensitive Calibration default ASC settings:

- touch_threshold_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 nt_keydetector_afid keydec =
{
    .signal_filter = {
        .cutoff = 8
    },
    .baseline_filter = {
        .cutoff = 4
    },
    .reset_rate = 1000,
    .asc = NT_KEYDETECTOR_AFID_ASC_DEFAULT,
};
```

8.4.4.4 Variable Documentation

8.4.4.4.1 struct nt_keydetector_interface nt_keydetector_afid_interface

AFID key detector interface structure.

8.4.5 Safa Detector

8.4.5.1 Overview

The Safa key detector is a method for recognizing the touch or release states. It can be used for each type of control.

If the measured sample is reported as a valid sample, the module calculates the delta value from the actual signal and baseline values. The delta value is compared to the threshold value computed from the expected signal and baseline values. Based on the result, it determines the electrode state, which can be released, touched, changing from released to touch, and changing from touched to release. This method is using moving average filters to determine the baseline and the expected signal values, with different depth of the filter, depending on the state of the electrode. The deadband filters in the horizontal and vertical directions are also implemented. Collaboration diagram for Safa Detector:

Data Structures

- struct nt_keydetector_safa

Variables

- struct nt_keydetector_safa nt_keydetector_safa_default
- struct nt_keydetector_interface nt_keydetector_safa_interface

8.4.5.2 Data Structure Documentation

8.4.5.2.1 struct nt_keydetector_safa

The main structure representing the Safa key detector. An instance of this data type represents the Safa key detector. It consists of used filters' parameters.

You're responsible for initializing all the members before registering the Safa in the module. This structure can be allocated in ROM.

Collaboration diagram for nt_keydetector_safa:

Data Fields

struct nt_filter-_moving_-average	base_avrg	Settings of the moving average filter for the baseline in the release state of an electrode.
uint32_t	deadband_cnt	Sample count for the deadband filter. This field specifies the number of samples that cannot proceed to the next event. For Example: after the touch event, a release event with "deadband_cnt" samples can follow.
uint32_t	entry_event_-cnt	Sample count for the touch event. This means that this count of samples must meet the touch condition to trigger a real touch event.
uint32_t	min_noise_-limit	Minimum noise value. Used for predicted signal calculation and delta signal filter limits

struct nt_filter-_moving_-average	non_activity_-avrg	Settings of the moving average filter for the signals in the inactivity state of an electrode. (for example baseline in a touch state).
struct nt_filter_iir	signal_filter	Coefficient of the input IIR signal filter, used to suppress high-frequency noise.
uint32_t	signal_to_-noise_ratio	Signal-to-noise ratio – it is used for counting the minimum size of the signal that is ignored.

8.4.5.3 Variable Documentation

8.4.5.3.1 struct nt_keydetector_safa nt_keydetector_safa_default

The default Safa settings of the key detector are:

- signal_filter (2 by default)
- base_avrg (9 by default, which means 512 samples)
- non_activity_avrg (NT_FILTER_MOVING_AVERAGE_MAX_ORDER by default, which means 65K samples)
- entry_event_cnt (8 by default)
- signal_to_noise_ratio (16 by default)
- deadband_cnt (10 by default)
- min_noise_limit (20 by default) Here is an example definition of the default values for SAFA ASC:

```
const struct nt_keydetector_safa nt_keydetector_safa_default =
{
    .signal_filter = 2,
    .base_avrg = {.n2_order = 9},
    .non_activity_avrg =  {.n2_order = NT_FILTER_MOVING_AVERAGE_MAX_ORDER
      },
    .entry_event_cnt = 8,
    .signal_to_noise_ratio = 16,
    .deadband_cnt = 10,
    .min_noise_limit = 20,
};
```

8.4.5.3.2 struct nt_keydetector_interface nt_keydetector_safa_interface

SAFA key detector interface structure.

8.4.6 Usafa Detector

8.4.6.1 Overview

The usafa key detector is a method for recognizing the touch or release states. It can be used for each type of control.

If the measured sample is reported as a valid sample, the module calculates the delta value from the actual signal and baseline values. The delta value is compared to the threshold value computed from the expected signal and baseline values. Based on the result, it determines the electrode state, which can be released, touched, changing from released to touch, and changing from touched to release. This method is using moving average filters to determine the baseline and the expected signal values with a different depth of the filter, depending on the state of the electrode. The dead band filters in the horizontal and vertical directions are also implemented. Collaboration diagram for Usafa Detector:

Data Structures

- struct nt_keydetector_usafa

Variables

- struct nt_keydetector_usafa nt_keydetector_usafa_default
- struct nt_keydetector_interface nt_keydetector_usafa_interface

8.4.6.2 Data Structure Documentation

8.4.6.2.1 struct nt_keydetector_usafa

The main structure representing the Usafa key detector. An instance of this data type represents the Usafa key detector. It consists of the used filters' parameters.

You're responsible for the initialization of all members before registering the usafa in the module. This structure can be allocated in ROM.

Collaboration diagram for nt_keydetector_usafa:

Data Fields

struct nt_filter-_moving_-average	base_avrg	Settings of the moving average filter for the baseline in the release state of an electrode.
uint32_t	dc_track_cnt	Define minimum number of cycles when signal must fall down below tha baseline to reset the usafa keydetector if DC track functionality is enabled.
uint32_t	dc_track_-enabled	Enable or disable DC track feature to avoid negative signal drop below the baseline - used to debounce the short glitches. If enabled and the signal is under baseline longer then defined in dc_track_cnt value the usafa keydetector will be reset.

uint32_t	deadband_cnt	Sample count for the deadband filter. This field specifies the number of samples that cannot proceed to the next event. For Example: after the touch event, a release event with "deadband_cnt" samples can follow.
uint32_t	entry_event_-cnt	Sample count for the touch event. This means that this count of samples must meet the touch condition to trigger a real touch event.
uint32_t	min_noise_-limit	Minimum noise value.
struct nt_filter-_moving_-average	non_activity_-avrg	Settings of the moving average filter for the signals in the inactivity state of an electrode. (for example baseline in a touch state).
struct nt_filter_iir	signal_filter	Coefficient of the input IIR signal filter, used to suppress high-frequency noise.
uint32_t	signal_to_-noise_ratio	Signal-to-noise ratio – it is used to count the minimum size of the signal that is ignored.

8.4.6.3 Variable Documentation

8.4.6.3.1 struct nt_keydetector_usafa nt_keydetector_usafa_default

The default Usafa settings of the key detector are:

- signal_filter (2 by default)
- base_avrg (9 by default, which means 512 samples)
- non_activity_avrg (NT_FILTER_MOVING_AVERAGE_MAX_ORDER by default, which means 65K samples)
- entry_event_cnt (8 by default)
- signal_to_noise_ratio (16 by default)
- deadband (10 by default)
- min_noise_limit (20 by default)
- dc_track_enabled (enabled by default), to avoid negative signal drop below baseline
- dc_track_cnt (100 by default), debouncing period Here is an example definition of the default values for usafa ASC:

```
const struct nt_keydetector_usafa nt_keydetector_usafa_default =
{
    .signal_filter = 2,
    .base_avrg = {.n2_order = 9},
    .non_activity_avrg =  {.n2_order = NT_FILTER_MOVING_AVERAGE_MAX_ORDER
      },
    .entry_event_cnt = 8,
    .signal_to_noise_ratio = 16,
    .deadband = 10,
    .min_noise_limit = 20,
    .dc_track_enabled = 1,
    .dc_track_cnt = 100,
};
```

8.4.6.3.2 struct nt_keydetector_interface nt_keydetector_usafa_interface

USAFA key detector interface structure.

8.5 Key Detectors

8.5.1 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 nt_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
- Safa Detector
- Usafa Detector

8.5.2 CS module

8.5.2.1 Overview

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

Data Structures

- struct nt_module_cs_params

Macros

- #define NT_MODULE_CS_NAME
- #define CUSTOM_DELAY()

Functions

- void delay (void)

Variables

- struct nt_module_interface nt_module_cs_interface

8.5.2.2 Data Structure Documentation

8.5.2.2.1 struct nt_module_cs_params

Collaboration diagram for nt_module_cs_params:

Data Fields

const cs_config_t *	config	A pointer to the HW configuration. Can't be NULL.

8.5.2.3 Macro Definition Documentation

8.5.2.3.1 #define CUSTOM_DELAY( )

8.5.2.3.2 #define NT_MODULE_CS_NAME

8.5.2.4 Function Documentation

8.5.2.4.1 void delay (void)

8.5.2.5 Variable Documentation

8.5.2.5.1 struct nt_module_interface nt_module_cs_interface

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

interface cs module

8.5.3 GPIO module

8.5.3.1 Overview

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

Data Structures

- struct nt_module_gpio_user_interface
- struct nt_module_gpio_params

Macros

- #define NT_MODULE_GPIO_NAME

Functions

- int32_t nt_gpio_data_check (const struct nt_module_data ∗module)

Variables

- struct nt_module_interface nt_module_gpio_interface

8.5.3.2 Data Structure Documentation

8.5.3.2.1 struct nt_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 nt_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)
- void(∗ set_pin_pull_up)(uint32_t port, uint32_t pin)
- void(∗ set_pin_pull_down)(uint32_t port, uint32_t pin)
- void(∗ set_pin_tsi_mode)(uint32_t port, uint32_t pin)
- void(∗ set_pin_gpio_mode)(uint32_t port, uint32_t pin)

8.5.3.2.1.1 Field Documentation

8.5.3.2.1.1.1 uint32_t(∗ nt_module_gpio_user_interface::get_pin_value)(uint32_t port, uint32_t pin)

Get pin value

8.5.3.2.1.1.2 void(∗ nt_module_gpio_user_interface::init_timer)(void)

Init timer

8.5.3.2.1.1.3 void(∗ nt_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

8.5.3.2.1.1.4 void(∗ nt_module_gpio_user_interface::set_pin_gpio_mode)(uint32_t port, uint32_t pin)

Set pin to GPIO mode

8.5.3.2.1.1.5 void(∗ nt_module_gpio_user_interface::set_pin_high)(uint32_t port, uint32_t pin)

Set pin to logic high

8.5.3.2.1.1.6 void(∗ nt_module_gpio_user_interface::set_pin_input)(uint32_t port, uint32_t pin)

Set pin direction to input

8.5.3.2.1.1.7 void(∗ nt_module_gpio_user_interface::set_pin_low)(uint32_t port, uint32_t pin)

Set pin to logic low

8.5.3.2.1.1.8 void(∗ nt_module_gpio_user_interface::set_pin_output)(uint32_t port, uint32_t pin)

Set pin direction to output

8.5.3.2.1.1.9 void(∗ nt_module_gpio_user_interface::set_pin_pull_down)(uint32_t port, uint32_t pin)

Set pin to GPIO with pull-down

8.5.3.2.1.1.10 void(∗ nt_module_gpio_user_interface::set_pin_pull_up)(uint32_t port, uint32_t pin)

Set pin to GPIO with pull-up

8.5.3.2.1.1.11 void(∗ nt_module_gpio_user_interface::set_pin_tsi_mode)(uint32_t port, uint32_t pin)

Set pin to TSI mode

8.5.3.2.1.1.12 void(∗ nt_module_gpio_user_interface::start_timer)(void)

Start timer

8.5.3.2.1.1.13 void(∗ nt_module_gpio_user_interface::stop_timer)(void)

Stop timer

8.5.3.2.1.1.14 uint32_t(∗ nt_module_gpio_user_interface::timer_get_counter)(void)

Get timer counter

8.5.3.2.1.1.15 uint32_t(∗ nt_module_gpio_user_interface::timer_get_overrun)(void)

Get timer overrun

8.5.3.2.1.1.16 void(∗ nt_module_gpio_user_interface::timer_reset_counter)(void)

Reset timer counter

8.5.3.2.2 struct nt_module_gpio_params

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

Collaboration diagram for nt_module_gpio_params:

Data Fields

struct nt_module_-gpio_user_-interface *	user_interface	

8.5.3.3 Macro Definition Documentation

8.5.3.3.1 #define NT_MODULE_GPIO_NAME

8.5.3.4 Function Documentation

8.5.3.4.1 int32_t nt_gpio_data_check (const struct nt_module_data * *module*)

8.5.3.5 Variable Documentation

8.5.3.5.1 struct nt_module_interface nt_module_gpio_interface

Can't be NULL.

interface gpio module

8.5.4 GPIO interrupt basic module

8.5.4.1 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 basic module:

Data Structures

- struct nt_module_gpioint_user_interface
- struct nt_module_gpioint_params

Macros

- #define NT_MODULE_GPIOINT_NAME

Functions

- void nt_module_gpioint_isr (const struct nt_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 nt_module_gpioint_isr() only if any of the GPIO modules' electrodes caused an interrupt.
- void nt_module_gpioint_overflow_isr (const struct nt_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 NXP Touch GPIO Interrupt module. It should be called after the user-defined maximal timeout for one measurement.

Variables

- struct nt_module_interface nt_module_gpioint_interface

8.5.4.2 Data Structure Documentation

8.5.4.2.1 struct nt_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 nt_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)

8.5.4.2.1.1 Field Documentation

8.5.4.2.1.1.1 void(∗ nt_module_gpioint_user_interface::clear_pin_interrupt)(uint32_t port, uint32_t pin)

Disable the pin to generate an interrupt

8.5.4.2.1.1.2 void(∗ nt_module_gpioint_user_interface::init_pin)(uint32_t port, uint32_t pin)

Initialize the pin to a state ready for measurement

8.5.4.2.1.1.3 void(∗ nt_module_gpioint_user_interface::init_timer)(void)

Init timer

8.5.4.2.1.1.4 void(∗ nt_module_gpioint_user_interface::set_pin_high)(uint32_t port, uint32_t pin)

Set the pin to logic high

8.5.4.2.1.1.5 void(∗ nt_module_gpioint_user_interface::set_pin_input)(uint32_t port, uint32_t pin)

Set the pin direction to input

8.5.4.2.1.1.6 void(∗ nt_module_gpioint_user_interface::set_pin_interrupt)(uint32_t port, uint32_t pin)

Enable the pin to generate an interrupt

8.5.4.2.1.1.7 void(∗ nt_module_gpioint_user_interface::set_pin_low)(uint32_t port, uint32_t pin)

Set the pin to logic low

8.5.4.2.1.1.8 void(∗ nt_module_gpioint_user_interface::set_pin_output)(uint32_t port, uint32_t pin)

Set the pin direction to output

8.5.4.2.1.1.9 void(∗ nt_module_gpioint_user_interface::start_timer)(void)

Start timer

8.5.4.2.1.1.10 void(∗ nt_module_gpioint_user_interface::stop_timer)(void)

Stop timer

8.5.4.2.1.1.11 uint32_t(∗ nt_module_gpioint_user_interface::timer_get_counter)(void)

Get timer counter

8.5.4.2.1.1.12 void(∗ nt_module_gpioint_user_interface::timer_reset_counter)(void)

Reset timer counter

8.5.4.2.2 struct nt_module_gpioint_params

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

Collaboration diagram for nt_module_gpioint_params:

Data Fields

struct nt_module_-gpioint_user_-interface *	user_interface	

8.5.4.3 Macro Definition Documentation

8.5.4.3.1 #define NT_MODULE_GPIOINT_NAME

8.5.4.4 Function Documentation

8.5.4.4.1 void nt_module_gpioint_isr (const struct nt_module * *module*)

Parameters

module	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:

8.5.4.4.2 void nt_module_gpioint_overflow_isr (const struct nt_module * *module*)

Parameters

module	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:

8.5.4.5 Variable Documentation

8.5.4.5.1 struct nt_module_interface nt_module_gpioint_interface

Can't be NULL.

interface gpio module

8.5.5 TSI module

8.5.5.1 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 nt_module_tsi_noise
- struct nt_module_tsi_params

Macros

- #define NT_MODULE_TSI_NAME

Functions

- int32_t nt_module_TSI_ReleaseAll (uint32_t instance)
  The function make virtual release on all electrodes assigned to TSI module by adding pull-up/down resistor.
- int32_t nt_module_TSI_Release (uint32_t instance, struct nt_electrode ∗∗electrodes, uint8_t counterNumber)
  The function make virtual the release on the specified electrodes assigned in the second function parameter by switching back to TSI mode.
- int32_t nt_module_TSI_ReleaseOneElect (struct nt_electrode ∗electrode)
  The function make virtual release on one specified electrode by switching back to the TSI mode.
- int32_t nt_module_TSI_TouchAll (uint32_t instance)
  The function make the virtual release on one specified electrode assigned to the TSI module by switching back to the TSI mode.
- int32_t nt_module_TSI_Touch (uint32_t instance, struct nt_electrode ∗∗electrodes, uint8_t el-Number)

The function make the virtual touch on the selected electrodes assigned in the second function parameter by adding pull-up/down resistor.

- int32_t nt_module_TSI_TouchOneElect (struct nt_electrode *electrode)
  The function make virtual touch on one specified electrode by adding the pull-up/down resistor.
- void delay (void)

Variables

- struct nt_module_interface nt_module_tsi_interface

8.5.5.2 Data Structure Documentation

8.5.5.2.1 struct nt_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 nt_module_tsi_noise:

Data Fields

struct nt_filter_iir	noise_filter	Noise filter parameters
uint8_t	noise_mode_-timeout	Time of switching back to capacitive mode since last noise event
uint8_t	update_rate	Period of the noise checking on the measured electrodes

8.5.5.2.2 struct nt_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.

Collaboration diagram for nt_module_tsi_params:

Data Fields

tsi_config_t *	config	A pointer to the HW configuration. Can't be NULL.
struct nt_module_tsi-_noise	noise	

8.5.5.3 Macro Definition Documentation

8.5.5.3.1 #define NT_MODULE_TSI_NAME

8.5.5.4 Function Documentation

8.5.5.4.1 void delay (void)

8.5.5.4.2 int32_t nt_module_TSI_Release (uint32_t *instance*, struct nt_electrode ** *electrodes*, uint8_t *counterNumber*)

This function simulate the release events on the specified electrodes defined as the bits in the second function parameters. The release event is simulated by changing the pin mode from GPIO mode back to TSI mode without pull-up/down resistor. The function requires to assign the gpio_input into the nt_electrode structure, one gpio for the self-capacitance based electrode or the array of two gpios for the mutual-capacitance based electrode.

```
int32_t returned_status;
struct nt_kernel * system = _nt_system_get();
struct nt_electrode **electrodes = (struct nt_electrode **)system->
      rom->modules[0]->electrodes;
returned_status = nt_module_TSI_Release(0, electrodes, 1);
if (returned_status != NT_SUCCESS)
{
  // virtual release was not successful for the specified electrodes
}
```

Parameters

instance	The TSI module instance.
electrodes	The pointer array to enabled electrodes to be measured.
elNumber	The number enabled electrodes to be measured.

Returns

- NT_SUCCESS if the specified electrodes have been switched back to the TSI mode
- NT_FAILURE if the specified electrodes have not been switched back to the TSI mode

8.5.5.4.3 int32_t nt_module_TSI_ReleaseAll (uint32_t *instance*)

This function simulate the release events on all electrodes assigned to the TSI module. The release event is simulated by changing the pin mode from the GPIO mode back to the TSI mode without the pull-up/down resistor. The function requires to assign the gpio_input into the nt_electrode structure, one gpio for the self-capacitance based electrode or the array of two gpios for the mutual-capacitance based electrode.

```
int32_t returned_status;
returned_status = nt_module_TSI_ReleaseAll(0);
if (returned_status != NT_SUCCESS)
{
  // virtual release was not successful for all electrodes
}
```

Parameters

instance	The TSI module instance.

Returns

- NT_SUCCESS if TSI register has been updated
- NT_FAILURE if TSI register has not been updated

8.5.5.4.4 int32_t nt_module_TSI_ReleaseOneElect (struct nt_electrode * *electrode*)

This function simulate the release events on one electrode assigned as function input electrode pointer. The release event is simulated by changing the pin mode from TSI channel to GPIO mode with pull-up/down resistor. For self-capacitance based electrode the pull-up resistor is used to simulate release event and for mutual-capacitance based the pull-down resistor applied on receiver pin is used to simulate release event. Considering the fact that all receiver pins are routed together the capacity change will be applied to all electrodes with connected rx pins. The function requires to assign the gpio_input into the nt_electrode structure, one gpio for the self-capacitance based electrode or the array of two gpios for the mutual-capacitance based electrode.

```
int32_t returned_status;
returned_status = nt_module_TSI_ReleaseOneElect(&El_0);
if (returned_status != NT_SUCCESS)
{
  // virtual release was not successful for one specified electrode
}
```

Parameters

instance	The TSI module instance.

Returns

- NT_SUCCESS if specified electrode has been switched back to the TSI mode
- NT_FAILURE if specified electrode has not been switched back to the TSI mode

8.5.5.4.5 int32_t nt_module_TSI_Touch (uint32_t *instance,* struct nt_electrode ∗∗ *electrodes,* uint8_t *elNumber*)

This function simulate the touch events on the selected electrodes defined as the bits in the second function parameter. The touch event is simulated by changing the pin mode from TSI channel to GPIO mode with pull-up/down resistor. For self-capacitance based electrode the pull-up resistor is used to simulate touch event and for mutual-capacitance based the pull-down resistor applied on receiver pin is used to simulate touch event. Considering the fact that all receiver pins are routed together the capacity change will be applied to all electrodes with connected rx pins.The function requires to assign the gpio_input into the nt_electrode structure, one gpio for the self-capacitance based electrode or the array of two gpios for the mutual-capacitance based electrode.

```
int32_t returned_status;
 struct nt_kernel * system = _nt_system_get();
 struct nt_electrode **electrodes = (struct nt_electrode **)system->
      rom->modules[0]->electrodes;
returned_status = nt_module_TSI_Touch(0, electrodes, 1);
if (returned_status != NT_SUCCESS)
{
  // virtual touch was not successful for the specified electrodes
}
```

Parameters

instance	The TSI module instance.
electrodes	The pointer array to enabled electrodes to be measured.
elNumber	The number of enabled electrodes to be measured.

Returns

- NT_SUCCESS if the specified electrodes have been switched to GPIO mode with pull-up/down
- NT_FAILURE if the specified electrodes have not been switched to GPIO mode with pull-up/down

8.5.5.4.6 int32_t nt_module_TSI_TouchAll (uint32_t *instance*)

This function simulate the release events on all electrodes assigned to the TSI modul. The release event is simulated by changing back the pin mode from the GPIO mode with the pull-up/down resistor to the TSI channel. The function requires to assign the gpio_input into the nt_electrode structure, one gpio for the self-capacitance based electrode or the array of two gpios for the mutual-capacitance based electrode.

```
int32_t returned_status;
returned_status = nt_module_TSI_TouchAll(0);
if (returned_status != NT_SUCCESS)
{
  // virtual touch was not successful for all electrodes
}
```

Parameters

instance	The TSI module instance.

Returns

- NT_SUCCESS if all electrodes have been switched to GPIO mode with pull-up/down
- NT_FAILURE if all electrodes have not been switched to GPIO mode with pull-up/down

8.5.5.4.7 int32_t nt_module_TSI_TouchOneElect (struct nt_electrode * *electrode*)

This function simulate the touch events on one electrode assigned as the function input electrode pointer. The touch event is simulated by changing the pin mode from the TSI channel to the GPIO mode with pull-up/down resistor. For the self-capacitance based electrode the pull-up resistor is used to simulate the touch event and for mutual-capacitance based the pull-down resistor applied on receiver pin is used to simulate the touch event. Considering the fact that all receiver pins are routed together the capacity change will be applied to all electrodes with connected rx pins. The function requires to assign the gpio_input into the nt_electrode structure, one gpio for the self-capacitance based electrode or the array of two gpios for the mutual-capacitance based electrode.

```
int32_t returned_status;
returned_status = nt_module_TSI_TouchOneElect(&El_0);
if (returned_status != NT_SUCCESS)
{
  // virtual touch was not successful for one specified electrode
}
```

Parameters

instance	The TSI module instance.

Returns

- NT_SUCCESS if one specified electrode has been switched to GPIO mode with pull-up/down
- NT_FAILURE if one specified electrodes has not been switched to GPIO mode with pull-up/down

8.5.5.5 Variable Documentation

8.5.5.5.1 struct nt_module_interface nt_module_tsi_interface

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

interface tsi module

8.6 Modules

8.6.1 Overview

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

Each Module implements a set of functions contained in the nt_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

- CS module
- GPIO module
- GPIO interrupt basic module
- TSI module
- General API

8.6.2 General API

8.6.2.1 Overview

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

Modules

- API Functions

Data Structures

- union nt_module_params
- struct nt_module_safety_params
- struct nt_module

Macros

- #define NT_MODULE_SIGNAL_WTRMARK_LOW
- #define NT_MODULE_SIGNAL_WTRMARK_HIGH

Enumerations

- enum nt_module_mode {
  NT_MODULE_MODE_NORMAL,
  NT_MODULE_MODE_PROXIMITY,
  NT_MODULE_MODE_LOW_POWER }
- enum nt_module_flags {
  NT_MODULE_NEW_DATA_FLAG,
  NT_MODULE_TRIGGER_DISABLED_FLAG,
  NT_MODULE_DIGITAL_RESULTS_FLAG,
  NT_MODULE_OVERFLOW_FLAG,
  NT_MODULE_BUSY_FLAG }

8.6.2.2 Data Structure Documentation

8.6.2.2.1 union nt_module_params

Container that covers all possible variants of the module parameters.

Collaboration diagram for nt_module_params:

Data Fields

struct nt_module_cs-_params *	cs	Pointer to the TSI module specific parameters.

const void *	general	Pointer for general use
struct nt_module_-gpio_params *	gpio	Pointer to the GPIO module specific parameters.
struct nt_module_-gpioint_params *	gpioint	Pointer to the GPIO interrupt module specific parameters.
struct nt_module_tsi-_params *	tsi	Pointer to the TSI module specific parameters.

8.6.2.2.2 struct nt_module_safety_params

Container that covers all possible variants of the Safety parameters.

Collaboration diagram for nt_module_safety_params:

Data Fields

struct nt_module_-safety_gpio_-params *	gpio	Pointer to user defined GPIO functions

8.6.2.2.3 struct nt_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 nt_module:

Data Fields

void *	config	A pointer to the hardware configuration. Can't be NULL.
struct nt_electrode *const *	electrodes	A pointer to the list of electrodes. Can't be NULL.
uint8_t	instance	An instance of the module.
struct nt_module_-interface *	interface	Module interface. Can't be NULL.
union nt_module_-params	module_-params	An instance module params. Can't be NULL.
void *	recalib_config	Recalibration structure for auto-calibration
struct nt_-module_safety-_interface *	safety_-interface	Module Safety interface. NULL when safety not needed
struct nt_module_-safety_params	safety_params	An instance module Safety params. NULL when safety not needed
uint16_t	wtrmark_hi	signal too high watermark
uint16_t	wtrmark_lo	signal too low watermark

8.6.2.3 Macro Definition Documentation

8.6.2.3.1 #define NT_MODULE_SIGNAL_WTRMARK_HIGH

Definition of the max signal value on the shield electrode used for offset compensation

8.6.2.3.2 #define NT_MODULE_SIGNAL_WTRMARK_LOW

Definition of the max signal value on the shield electrode used for offset compensation

8.6.2.4 Enumeration Type Documentation

8.6.2.4.1 enum nt_module_flags

Generic flags for Module processing.

Enumerator

NT_MODULE_NEW_DATA_FLAG The new data is ready to be processed.
NT_MODULE_TRIGGER_DISABLED_FLAG Disables the trigger for the current module (in fact, the module is disabled).
NT_MODULE_DIGITAL_RESULTS_FLAG The digital data only flag (only touch / release information - no analog value).
NT_MODULE_OVERFLOW_FLAG Measured module data out of the specified range
NT_MODULE_BUSY_FLAG Module busy flag

8.6.2.4.2 enum nt_module_mode

Module's modes.

Enumerator

NT_MODULE_MODE_NORMAL The module is in a standard touch measure mode.
NT_MODULE_MODE_PROXIMITY The module is in a proximity mode.
NT_MODULE_MODE_LOW_POWER The module is in a low-power mode.

8.6.2.5 API Functions

8.6.2.5.1 Overview

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

Functions

- uint32_t nt_module_recalibrate (const struct nt_module ∗module)
  Recalibrate the module. The function forces the recalibration process of the module to get optimized parameters.
- int32_t nt_module_change_mode (struct nt_module ∗module, const enum nt_module_mode mode, const struct nt_electrode ∗electrode)
  Changes the module mode of the operation.
- int32_t nt_module_load_configuration (struct nt_module ∗module, const enum nt_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 nt_module_save_configuration (struct nt_module ∗module, const enum nt_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.

8.6.2.5.2 Function Documentation

8.6.2.5.2.1 int32_t nt_module_change_mode (struct nt_module ∗ *module,* const enum nt_module_mode *mode,* const struct nt_electrode ∗ *electrode*)

Parameters

module	Pointer to the module.

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

Returns

- NT_SUCCESS if the mode was properly changed
- NT_FAILURE if the mode cannot be changed This is an example of changing the mode of the module operation in the NT library:

```
if(nt_module_change_mode(&my_nt_module,
      NT_MODULE_MODE_PROXIMITY, &my_proximity_electrode) ==
      NT_FAILURE)
{
  printf("The change of mode for my_nt_module failed.");
}
// The NT successfully changed mode of my_nt_module
```

Here is the call graph for this function:

8.6.2.5.2.2 int32_t nt_module_load_configuration (struct nt_module ∗ *module,* const enum nt_module_mode *mode,* const void ∗ *config*)

Parameters

module	Pointer to the module.

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

Returns

- NT_SUCCESS if the load operation was properly done
- NT_FAILURE if the load operation cannot be finished This is an example of loading the configuration data of the module in the NT library:

```
   // I want to load new configuration for the TSI module and the proximity mode
   if(nt_module_load_configuration(&my_nt_module,
      NT_MODULE_MODE_PROXIMITY, &my_module_tsi_proximity_configuration) ==
NT_FAILURE)
   {
     printf("Loading of new configuration for the my_nt_module failed.");
   }
   // The NT successfully loaded the new configuration of the my_nt_module.
```

Here is the call graph for this function:

8.6.2.5.2.3 uint32_t nt_module_recalibrate (const struct nt_module ∗ *module*)

Parameters

configuration	Pointer to the module configuration defining recalibration ranges.
module	Pointer to the module to be recalibrated.

Returns

TSI status. This is an example of recalibrating the module settings of the NT library:

```
if(nt_module_recalibrate(&my_nt_module, &my_nt_configuration) ==
      NT_FAILURE)
{
  printf("The change of mode for my_nt_module failed.");
}
// The NT successfully change mode of my_nt_module
```

Here is the call graph for this function:

8.6.2.5.2.4 int32_t nt_module_save_configuration (struct nt_module * *module,* const enum nt_module_mode *mode,* void * *config*)

Parameters

module	Pointer to the module.
mode	Mode of the module.
config	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

- NT_SUCCESS if the save operation was properly done
- NT_FAILURE if the save operation cannot be finished This is an example of saving the configuration data of the module in the NT 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(nt_module_save_configuration(&my_nt_module,
      NT_MODULE_MODE_PROXIMITY, &my_module_tsi_proximity_configuration) ==
NT_FAILURE)
   {
     printf("Saving of the current configuration for the my_nt_module failed.");
   }
   // The NT successfully saved the current configuration of the my_nt_module
```

Here is the call graph for this function:

8.7 Safety

8.7.1 Overview

Safety represents the check layer for the FMEA tests in the NXP Touch system; it is the layer that is tightly coupled with the hardware module available on the NXP MCU device and the GPIO pins. Collaboration diagram for Safety:

Modules

- General API

Data Structures

- struct nt_module_safety_gpio_params

Macros

- #define NT_MODULE_SAFETY_NAME

8.7.2 Data Structure Documentation

8.7.2.1 struct nt_module_safety_gpio_params

GPIO interface structure for the Safety tests

Collaboration diagram for nt_module_safety_gpio_params:

Data Fields

uint32_t	delay_interval	Delay interval to to apply the physical changes on the GPIO pin and equalize charge.

struct nt_module_-gpio_user_-interface *	user_interface	

8.7.3 Macro Definition Documentation

8.7.3.1 #define NT_MODULE_SAFETY_NAME

8.7.4 General API

8.7.4.1 Overview

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

Functions

- int32_t nt_module_safety_gpio_check_data (const struct nt_module_data ∗module)
  Check the safety interface.
- int32_t nt_module_safety_check_electrode_short_gnd_set (const struct nt_electrode ∗elec)
  Short to ground (GND) test.
- int32_t nt_module_safety_check_electrode_short_gnd_read (const struct nt_electrode ∗elec)
  Short to ground (GND) test.
- int32_t nt_module_safety_check_electrode_short_vdd (const struct nt_electrode ∗elec)
  Short to the power supply (VDD) test.
- int32_t nt_module_safety_check_electrode_short_adj (const struct nt_electrode ∗elec)
  Short one electrode with the other one test.
- int32_t nt_module_safety_check_baseline_low (const struct nt_electrode ∗elec)
  Electrode baseline too low level test.
- int32_t nt_module_safety_check_baseline_high (const struct nt_electrode ∗elec)
  Electrode baseline too high level test.
- int32_t nt_module_safety_check_electrodes (struct nt_module_data ∗module)
  Execute all safety tests.
- int32_t _nt_module_safety_switch_electrode_gpio (const struct nt_electrode ∗elec)
  Function change electrode pin mode from TSI to GPIO.
- int32_t _nt_module_safety_switch_electrode_tsi (const struct nt_electrode ∗elec)
  Function change electrode pin mode from GPIO to TSI.

Variables

- struct nt_module_safety_interface nt_safety_interface

8.7.4.2 Function Documentation

8.7.4.2.1 int32_t _nt_module_safety_switch_electrode_gpio (const struct nt_electrode ∗ *elec*)

Returns

The test result status (success or failure).

The function read the gpio and pin information from gpio_input member of the electrode structure and and change this pin to GPIO mode.

```
   // get module data
   struct nt_module_data *module = _nt_module_get_data(
      _nt_system_get_module((uint32_t)&nt_module_tsi_interface,
instance)); NT_ASSERT(module != NULL);
   // switch electrode 0 to GPIO
   struct nt_electrode const *electrode = module->rom->electrodes[0];
   int32_t Result = _nt_module_safety_switch_electrode_gpio(
      electrode);
```

Here is the call graph for this function:

8.7.4.2.2 int32_t _nt_module_safety_switch_electrode_tsi (const struct nt_electrode * *elec*)

Returns

The test result status (success or failure).

Function change electrode pin mode from GPIO to TSI.

```
   // get module data
   struct nt_module_data *module = _nt_module_get_data(
      _nt_system_get_module((uint32_t)&nt_module_tsi_interface,
instance)); NT_ASSERT(module != NULL);
   // switch electrode 0 to GPIO
   struct nt_electrode const *electrode = module->rom->electrodes[0];
   int32_t Result = _nt_module_safety_switch_electrode_tsi(electrode)
      ;
```

Here is the call graph for this function:

8.7.4.2.3 int32_t nt_module_safety_check_baseline_high (const struct nt_electrode * *elec*)

Returns

The test result status (success or failure).

This function reads the actual baseline level of the tested electrode and compares it to the stored baseline level. If the actual baseline value is 25% higher than the stored baseline, the function returns the failure status. If the actual baseline level is lower than 125% of the stored baseline, the function returns the success status. If all tests passed, the function returns the success status. If not, the failure status is returned.

```
// Test the baseline high level for the already defined my_electrode
int32_t testResult;
testResult = nt_module_safety_check_baseline_high(&my_electrode);
```

Here is the call graph for this function:

Here is the caller graph for this function:

8.7.4.2.4 int32_t nt_module_safety_check_baseline_low (const struct nt_electrode * *elec*)

Returns

The test result status (success or failure).

This function reads the actual baseline level of the tested electrode and compares it to the stored baseline level. If the actual baseline value is approximately 12.5% lower than the stored baseline, the function returns the failure status. If the actual baseline level is higher than 87.5% of the stored baseline, the function returns the success status.

```
// Test the baseline low level for the already defined my_electrode
int32_t testResult;
testResult = nt_module_safety_check_baseline_low(&my_electrode);
```

Here is the call graph for this function:

Here is the caller graph for this function:

8.7.4.2.5 int32_t nt_module_safety_check_electrode_short_adj (const struct nt_electrode * *elec*)

Returns

The test result status (success or failure).

This function tests a possible electrode short to another electrode. Firstly, the function changes all electrode pins (except for the tested electrode) to the GPIO mode with the logic zero output. The tested electrode is set as a GPIO input with pull-up enabled. The level of the tested electrode is then read from the port. The "logic One" value is expected for an undamaged electrode and the success result. If "logic Zero" is detected, the failure status is returned. When the mutual-cap mode is defined for an electrode, both of the associated (Rx and Tx) pins are tested for shorts.

```
// Test if "my_electrode" is shorted to another electrode
int32_t testResult;
testResult = nt_module_safety_check_electrode_short_adj(&
      my_electrode);
```

8.7.4.2.6 int32_t nt_module_safety_check_electrode_short_gnd_read (const struct nt_electrode * *elec*)

Returns

The test result status (success or failure).

This function tests the possible electrode short to the GND. Firstly, the function switches the electrode pin to the GPIO input mode with the pull-up enabled. The pin value is read from the port. The "logic One" value is expected for a good, undamaged electrode and the success result. If "logic Zero" is detected, the failure status is returned. When the mutual-cap mode is defined for the electrode, both of the associated (Rx and Tx) pin shorts are tested.

```
// Test if "my_electrode" is shorted to the GND
int32_t testResult;
testResult = nt_module_safety_check_electrode_short_gnd(&my_electrode);
```

8.7.4.2.7 int32_t nt_module_safety_check_electrode_short_gnd_set (const struct nt_electrode * *elec*)

Returns

The test result status (success or failure).

This function tests the possible electrode short to the GND. Firstly, the function switches the electrode pin to the GPIO input mode with the pull-up enabled. The pin value is read from the port. The "logic One" value is expected for a good, undamaged electrode and the success result. If "logic Zero" is detected, the failure status is returned. When the mutual-cap mode is defined for the electrode, both of the associated (Rx and Tx) pin shorts are tested.

```
// Test if "my_electrode" is shorted to the GND
int32_t testResult;
testResult = nt_module_safety_check_electrode_short_gnd(&my_electrode);
```

8.7.4.2.8 int32_t nt_module_safety_check_electrode_short_vdd (const struct nt_electrode * *elec*)

Returns

The test result status (success or failure).

This function tests the possible electrode short to the VDD. Firstly, the function switches the electrode pin to the GPIO input mode with pull-down enabled. The pin value is read from the port. The logic zero value is expected for a good, undamaged electrode and the success result. If the logic one is detected, the failure status is returned. When the mutual-cap mode is defined for the electrode, both of the associated (Rx and Tx) pin shorts are tested.

```
// Test if "my_electrode" is shorted to VDD
int32_t testResult;
testResult = nt_module_safety_check_electrode_short_vdd(&
      my_electrode);
```

8.7.4.2.9 int32_t nt_module_safety_check_electrodes (struct nt_module_data * *module*)

Returns

The test result status (success or failure).

The function is a high level safety function that checks all electrodes for shorts to the GND, VDD, or other electrodes and it executes the baseline too low and high level tests. To make all tests, the electrodes must be switched to the GPIO mode using the _nt_module_safety_switch_electrodes_gpio function. After the test, the electrodes are switched back using the _nt_module_safety_switch_electrodes_tsi function.

```
// Complete the safety test of all electrodes from the module where my_electrode is defined.
int32_t testResult;
testResult = nt_module_safety_check_electrodes (
      nt_electrode_get_module_data(&my_electrode));
```

Here is the call graph for this function:

Here is the caller graph for this function:

8.7.4.2.10 int32_t nt_module_safety_gpio_check_data (const struct nt_module_data * *module*)

Returns

the gpio user interface status

Check the status safety gpio user interface and check if the GPIO pins and the timer function are properly defined.

```
// Complete the safety test of all electrodes from the module where my_electrode is defined.
int32_t testResult;
testResult = nt_module_safety_gpio_check_data(
      nt_electrode_get_module_data(&my_electrode));
```

Here is the caller graph for this function:

8.7.4.3 Variable Documentation

8.7.4.3.1 struct nt_module_safety_interface nt_safety_interface

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

interface module safety

8.8 System

8.8.1 Overview

The System structure represents the NXP Touch Library in the user application; it is represented by the nt_system structure, which contains further references to all other application objects like: Electrodes, Key Detectors, Modules, and Controls.

The nt_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 NXP Touch library show the static allocation and initialization of nt_system along with all other related data structures. Collaboration diagram for System:

Modules

- API Functions

Data Structures

- union nt_system_event_context
- struct nt_system_event_context.sig_out_of_limit
- struct nt_system

Typedefs

- typedef void(∗ nt_system_callback)(uint32_t event, union nt_system_event_context ∗context)
- typedef void(∗ nt_error_callback)(char ∗file_name, uint32_t line)

Enumerations

- enum nt_system_event {
  NT_SYSTEM_EVENT_OVERRUN,
  NT_SYSTEM_EVENT_DATA_READY,
  NT_SYSTEM_EVENT_MODULE_DATA_READY,
  NT_SYSTEM_EVENT_DATA_OVERFLOW,
  NT_SYSTEM_EVENT_SIGNAL_LOW,
  NT_SYSTEM_EVENT_SIGNAL_HIGH,
  NT_SYSTEM_EVENT_ELEC_SHORT_VDD,
  NT_SYSTEM_EVENT_ELEC_SHORT_GND,
  NT_SYSTEM_EVENT_ELEC_SHORT_ADJ,
  NT_SYSTEM_EVENT_CRC_FAILED }

8.8.2 Data Structure Documentation

8.8.2.1 union nt_system_event_context

Collaboration diagram for nt_system_event_context:

Data Fields

struct nt_-module_data *	module_data	
struct nt_system_-event_context	sig_out_of_-limit	

8.8.2.2 struct nt_system_event_context.sig_out_of_limit

Collaboration diagram for nt_system_event_context.sig_out_of_limit:

Data Fields

struct nt_electrode *	electrode	

8.8.2.3 struct nt_system

The main structure representing the NXP 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 nt_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 NXP Touch library show the static allocation and initialization of the nt_system, along with all other related data structures.

This structure can be allocated in ROM.

Collaboration diagram for nt_system:

Data Fields

struct nt_control ∗const ∗	controls	A pointer to the list of controls. Can't be NULL.
char ∗	gui_path	GUI path to read xml configuration
uint16_t	init_time	Initialization time for the system.
struct nt_module ∗const ∗	modules	A pointer to the list of modules. Can't be NULL.
uint8_t	safety_crc_hw	true if the crc hw module present on chip will be used
uint16_t	safety_period_-multiple	Multiplier of the time period Time period for safety process function execution. If same as time period the safety process will be executed each NT system period. To avoid safety process execution set zero

uint16_t	time_period	Defined time period (triggering period). Can't be 0.

8.8.3 Typedef Documentation

8.8.3.1 typedef void(∗ nt_error_callback)(char ∗file_name, uint32_t line)

Error callback function pointer type.

Parameters

file	The name of the file where the error occurs.
line	The line index in the file where the error occurs.

Returns

None.

8.8.3.2 typedef void(∗ nt_system_callback)(uint32_t event, union nt_system_event_context ∗context)

System event callback function pointer type.

Parameters

event	Event type nt_system_event that caused the callback function call.

Returns

None.

8.8.4 Enumeration Type Documentation

8.8.4.1 enum nt_system_event

System callbacks events.

Enumerator

NT_SYSTEM_EVENT_OVERRUN Data has been overrun
NT_SYSTEM_EVENT_DATA_READY New data are available
NT_SYSTEM_EVENT_MODULE_DATA_READY New data measured
NT_SYSTEM_EVENT_DATA_OVERFLOW Measured data overflow (HW out of range)

NT_SYSTEM_EVENT_SIGNAL_LOW Measured data overflow (HW out of range)
NT_SYSTEM_EVENT_SIGNAL_HIGH Measured data overflow (HW out of range)
NT_SYSTEM_EVENT_ELEC_SHORT_VDD Possible Short to VDD on electrode pin detected
NT_SYSTEM_EVENT_ELEC_SHORT_GND Possible Short to GND on electrode pin detected
NT_SYSTEM_EVENT_ELEC_SHORT_ADJ Possible Short to Adjacent on electrode pin detected

NT_SYSTEM_EVENT_CRC_FAILED CRC failure on the system cfg data detected

8.8.5 API Functions

8.8.5.1 Overview

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

Functions

- void nt_system_register_callback (nt_system_callback callback)
  Register the system callback function.
- void nt_error_register_callback (nt_error_callback callback)
  Register the system error callback function.
- uint32_t nt_system_get_time_counter (void)
  Returns the system time counter.
- uint32_t _nt_system_get_safety_period (void)
  Returns the safety period.
- uint32_t nt_system_get_time_offset (uint32_t event_stamp)
  Returns the system time counter offset.
- uint32_t nt_mem_get_free_size (void)
  Returns the free memory size in the NT memory pool.
- int32_t nt_init (const struct nt_system *system, uint8_t *pool, const uint32_t size)
  NXP Touch Library initialization.
- int32_t nt_task (void)
  NXP Touch Main processing entry point.
- int32_t nt_trigger (void)
  Main Trigger function to acquire the touch-sensing data.

8.8.5.2 Function Documentation

8.8.5.2.1 uint32_t _nt_system_get_safety_period (void)

Returns

Returns the safety period calculated as system time period and safety period multioplier. This is an example of getting the current time of the NT library:

```
*  // Printing the current NXP Touch library safety period
*  printf("The NT library safety period is: &d ms since start.\n",
```

```
        _nt_system_get_safety_period());
*
*
```

8.8.5.2.2 void nt_error_register_callback (nt_error_callback *callback*)

Parameters

callback	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 NT library error handler:

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

8.8.5.2.3 int32_t nt_init (const struct nt_system ∗ *system,* uint8_t ∗ *pool,* const uint32_t *size*)

Parameters

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

Returns

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

This function validates the NXP Touch configuration passed within the nt_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 NT library calls if this function returns NT_FAILURE. This is an example of the NT library initialization:

```
uint8_t nt_memory_pool[512];

if(nt_init(&my_nt_system_params, nt_memory_pool, sizeof(nt_memory_pool)) ==
      NT_FAILURE)
{
  printf("Initialization of NT failed. There can be a problem with the memory size
  or invalid parameters in component parameter structures.");
}
// The NT is successfully initialized
```

Here is the call graph for this function:

8.8.5.2.4 uint32_t nt_mem_get_free_size (void)

Returns

size of unused memory in the NT memory pool

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

```
*  uint8_t nt_memory_pool[512];
*
*  if(nt_init(&my_nt_system_params, nt_memory_pool, sizeof(nt_memory_pool)) ==
      NT_FAILURE)
*  {
*    printf("Initialization of the NT failed. There may be a problem with the memory size,
*    or invalid parameters in the complemented parameter structures.");
*  }
*  // The NT is successfully initialized
*
*  printf("The unused memory size is: &d Bytes. The memory pool can be reduced
*  by this size.", nt_mem_get_free_size());
*
*
```

Here is the call graph for this function:

8.8.5.2.5 uint32_t nt_system_get_time_counter (void)

Returns

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

```
*  // Printing the current NXP Touch library time
*  printf("The current NT library time is: &d ms since start.\n",
      nt_system_get_time_counter());
*
*
```

Here is the caller graph for this function:

8.8.5.2.6 uint32_t nt_system_get_time_offset (uint32_t *event_stamp*)

Returns

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

```
*  // Printing the current NXP Touch library time offset
*  printf("The NT library time offset is: &d ms since start.\n",
      nt_system_get_time_offset());
*
*
```

Here is the call graph for this function:

Here is the caller graph for this function:

8.8.5.2.7 void nt_system_register_callback (nt_system_callback *callback*)

Parameters

callback	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 NT library system events handler:

```
*  static void my_nt_system_callback(uint32_t event);
*
*  // To catch the system events, install the system handler
*  nt_system_register_callback(my_nt_system_callback)
*
*  // The NT system events handling routine
*  static void my_nt_system_callback(uint32_t event)
*  {
*    if(event == NT_SYSTEM_EVENT_OVERRUN)
*    {
*      printf("\nThe measured data has been overrun. Call more frequently nt_task();");
*    }
*    else if(event == NT_SYSTEM_EVENT_DATA_READY)
*    {
*      printf("\nThere is new data in the NT library.);
*    }
*  }
*
*
```

8.8.5.2.8 int32_t nt_task (void)

Returns

- NT_SUCCESS when data acquired during the last trigger are now processed
- NT_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 NT_SYSTEM_MODULE_PROCESS and NT_SYSTEM_CONTROL_PROCESS command calls to each object configured in Modules and Controls. This is an example of running a task of the NT library:

```
uint8_t nt_memory_pool[512];

if(nt_init(&my_nt_system_params, nt_memory_pool, sizeof(nt_memory_pool)) ==
      NT_FAILURE)
{
  printf("Initialization of NT failed. There can be problem with memory size
  or invalid parameters in component parameter structures.");
}
// The NT is successfully initialized

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

Here is the call graph for this function:

8.8.5.2.9 int32_t nt_trigger (void)

Returns

- NT_SUCCESS when the trigger was performed without any errors or warnings.
- NT_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 NT library triggering:

```
//For example, there is a callback routine from any periodical source (for example 5 ms)
static void Timer_5msCallBack(void)
{
 if(nt_trigger() != NT_SUCCESS)
  {
    // Trigger error
  }
}
```

Here is the call graph for this function:

8.9 General Types

8.9.1 Overview

The standard types used in the whole NXP 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 NT. Collaboration diagram for General Types:

Macros

- #define NULL
  Standard NULL pointer. There is a definition in the NT, in case that NULL is not defined in the project previously.
- #define NT_FLAGS_SYSTEM_SHIFT(x)
- #define NT_FLAGS_SPECIFIC_SHIFT(x)
- #define NT_FREEMASTER_SUPPORT
  NT_FREEMASTER_SUPPORT enables the support of FreeMASTER for the NXP Touch project. When it is enabled, the NT starts using / including the freemaster.h file. NT_FREEMASTER_SUPPORT is enabled by default.
- #define NT_SAFETY_SUPPORT
  NT_SAFETY_SUPPORT enables the support of safety function for the NXP Touch project. When it is enabled, the NT starts using / including the safety files. NT_SAFETY_SUPPORT is enabled by default.
- #define NT_DEBUG
  NT_DEBUG enables the debugging code that caused the assert states in the NT. For the release compilation, this option should be disabled. NT_DEBUG is enabled by default, which enables the NT ASSERTS.
- #define NT_ASSERT(expr)

Enumerations

- enum nt_result {
  NT_SUCCESS,
  NT_FAILURE,
  NT_OUT_OF_MEMORY,
  NT_SCAN_IN_PROGRESS,
  NT_NOT_SUPPORTED,
  NT_INVALID_RESULT }

The NXP 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.

8.9.2 Macro Definition Documentation

8.9.2.1 #define NT_ASSERT(*expr*)

8.9.2.2 #define NT_DEBUG

8.9.2.3 #define NT_FLAGS_SPECIFIC_SHIFT(*x*)

8.9.2.4 #define NT_FLAGS_SYSTEM_SHIFT(*x*)

Generic flags for NT processing.

8.9.2.5 #define NT_FREEMASTER_SUPPORT

8.9.2.6 #define NT_SAFETY_SUPPORT

8.9.2.7 #define NULL

8.9.3 Enumeration Type Documentation

8.9.3.1 enum nt_result

Enumerator

NT_SUCCESS Successful result - Everything is all right.
NT_FAILURE Something is wrong, unspecified error.
NT_OUT_OF_MEMORY The NT does not have enough memory.
NT_SCAN_IN_PROGRESS The scan is currently in progress.
NT_NOT_SUPPORTED The feature is not supported.
NT_INVALID_RESULT The function ends with an invalid result.

8.9.4 Analog Rotary Control

8.9.4.1 Overview

Analog Rotary enables the detection of jog-dial-like finger movement using three or more electrodes; it is represented by the nt_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 8.9.1: Analog Rotary Electrodes

Collaboration diagram for Analog Rotary Control:

Data Structures

- struct nt_control_arotary_temp_data
- struct nt_control_arotary_data

Macros

- #define NT_AROTARY_INVALID_POSITION_VALUE

Enumerations

- enum nt_control_arotary_flags {
  NT_AROTARY_INVALID_POSITION_FLAG,
  NT_AROTARY_DIRECTION_FLAG,
  NT_AROTARY_MOVEMENT_FLAG,
  NT_AROTARY_TOUCH_FLAG }

8.9.4.2 Data Structure Documentation

8.9.4.2.1 struct nt_control_arotary_temp_data

Analog Rotary help structure to handle temporary values

Collaboration diagram for nt_control_arotary_temp_data:

Data Fields

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

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

8.9.4.2.2 struct nt_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 nt_control structure when it is being registered in the system.

Collaboration diagram for nt_control_arotary_data:

Data Fields

nt_control_-arotary_-callback	callback	Analog Rotary callback handler.

uint8_t	position	Position.

8.9.4.3 Macro Definition Documentation

8.9.4.3.1 #define NT_AROTARY_INVALID_POSITION_VALUE

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

8.9.4.4 Enumeration Type Documentation

8.9.4.4.1 enum nt_control_arotary_flags

Analog Rotary flags.

Enumerator

NT_AROTARY_INVALID_POSITION_FLAG Analog Rotary invalid position flag.
NT_AROTARY_DIRECTION_FLAG Analog Rotary direction flag.
NT_AROTARY_MOVEMENT_FLAG Analog Rotary movement flag.
NT_AROTARY_TOUCH_FLAG Analog Rotary touch flag.

8.9.5 Analog Slider Control

8.9.5.1 Overview

Analog Slider enables the detection of linear finger movement using two or more electrodes; it is represented by the nt_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 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 image below shows a typical two-electrode Analog Slider electrode placement.

Figure 8.9.2: Analog Slider Electrodes

Collaboration diagram for Analog Slider Control:

Data Structures

- struct nt_control_aslider_data
- struct nt_control_aslider_temp_data

Macros

- #define NT_ASLIDER_INVALID_POSITION_VALUE

Enumerations

- enum nt_control_aslider_flags {
  NT_ASLIDER_INVALID_POSITION_FLAG,
  NT_ASLIDER_DIRECTION_FLAG,
  NT_ASLIDER_MOVEMENT_FLAG,
  NT_ASLIDER_TOUCH_FLAG }

8.9.5.2 Data Structure Documentation

8.9.5.2.1 struct nt_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 nt_control_aslider structure when it is being registered in the system.

Collaboration diagram for nt_control_aslider_data:

Data Fields

nt_control_-aslider_-callback	callback	Analog Slider callback handler.
uint8_t	position	Position.

8.9.5.2.2 struct nt_control_aslider_temp_data

Analog Slider help structure to handle temporary values

Collaboration diagram for nt_control_aslider_temp_data:

Data Fields

uint32_t	active_el_ix	Index of electrode of first active electrode.
uint32_t	first_delta	Value of first delta (signal - baseline).
uint32_t	range	tmp. value of slider range

8.9.5.3 Macro Definition Documentation

8.9.5.3.1 #define NT_ASLIDER_INVALID_POSITION_VALUE

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

8.9.5.4 Enumeration Type Documentation

8.9.5.4.1 enum nt_control_aslider_flags

Analogue Slider flags.

Enumerator

NT_ASLIDER_INVALID_POSITION_FLAG Analog Slider invalid position flag.
NT_ASLIDER_DIRECTION_FLAG Analog Slider direction flag.
NT_ASLIDER_MOVEMENT_FLAG Analog Slider movement flag.
NT_ASLIDER_TOUCH_FLAG Analog Slider touch flag.

8.9.6 Keypad Control

8.9.6.1 Overview

Keypad implements the keyboard-like functionality on top of an array of electrodes; it is represented by the nt_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 8.9.3: Keypad Electrodes

Collaboration diagram for Keypad Control:

Data Structures

- struct nt_control_keypad_data

Enumerations

- enum nt_control_keypad_flags {
  NT_KEYPAD_ONLY_ONE_KEY_FLAG,
  NT_KEYPAD_MULTI_TOUCH_FLAG }

8.9.6.2 Data Structure Documentation

8.9.6.2.1 struct nt_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 nt_control_keypad structure when it is being registered in the system.

Collaboration diagram for nt_control_keypad_data:

Data Fields

uint16_t	autorepeat_rate	Autorepeat rate.
nt_control_-keypad_-callback	callback	Keypad callback handler.
int32_t	last_electrode	Last touched electrode index.
uint64_t	last_key_state	Last state of keypad keys.
uint64_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.

8.9.6.3 Enumeration Type Documentation

8.9.6.3.1 enum nt_control_keypad_flags

Keypad flags.

Enumerator

NT_KEYPAD_ONLY_ONE_KEY_FLAG Keypad only one key is valid flag.
NT_KEYPAD_MULTI_TOUCH_FLAG Multifunction keypad - multi touch recognition

8.9.7 Matrix Control

8.9.7.1 Overview

The matrix implements a similar functionality to the keypad but the matrix has the electrode surfaces placed closer to each other. Moreover, the matrix control allows processing the position tracking and gesture recognition. Usually, the matrix control is used for data processing from the touchpad hardware.

Applications may use the electrode API to determine the touch or release states of individual electrodes (digital approach) but the matrix control processes the electrodes as a two-dimensional array. The images below show a simple electrode array where the self-cap sensing is used.

Figure 8.9.4: Matrix Self-cap Sensing Electrodes

To recognize the position on the shown layout means to activate two electrodes (horizontal and vertical). When this happens, the x and y coordinates of the touch position are known and can be tracked. When the finger is moving on the surface, the actual x and y coordinates are changing in time and the position changes are stored into the buffer. At the end of the movement, the release position is stored as well and the gesture recognition algorithm can be executed.

The same processing is used for the mutual-cap-sensing-based matrix where the position is known after touching one electrode only because the mutual-cap-sensing electrode consists of the Tx and Rx electrodes which determine the position on the surface by a mutual Tx and Rx electrode hardware interconnection. The position detection is only one difference between the self-cap and mutual-cap matrices. Other matrix control processing algorithms remain the same. The mutual-cap-sensing-based matrix is enabled by defining the value of "true" to the sensing_mode matrix constant, similarly like the matrix horizontal and vertical sizes.

1	2	3	4
5	6	7	8
9	10	11	12

Figure 8.9.5: Matrix Mutual-cap Sensing Electrodes

The position recognition can be extended by a simple trick for both matrix types. The touch on the two nearest electrodes can be considered as virtual electrodes between the touched electrodes. In this way, the horizontal and vertical resolution can be extended by adding gaps between the electrodes. The new resolution is twice the original resolution minus one (extended_resolution = origin_resolution + electrode-_gaps = 2 ∗ origin_resolution - 1). If two nearest electrodes are activated at the same time, the algorithm determines that a virtual electrode is activated between the real electrodes.

The matrix control processes the position regardless of the finger size. A large finger activates more than one horizontal electrode and one vertical electrode for the self-cap mode. The algorithm ignores the other nearest touched electrodes to centralize the recognized position. The feature ensures a most precise position on the surface for next processing. The position can be determined from one, two, or three nearest electrodes. If more nearest electrodes are enabled, the position is recognized as two active figures. This rule is valid for horizontal and vertical electrodes.

The matrix control algorithm recognizes the position of not only one finger but two fingers as well. The finger position or two finger positions are stored for the touch, release, and movement (actual position) events. Also, the movement tracking and gestures can process two fingers.

The gesture feature is the most important part of the matrix control. The gestures can be divides to two groups: control gestures and unlock gestures. The control gestures are usually used to control the device, change the mode, configure a feature, or make some user action. The unlock gestures stored by the user are typically used to unlock the screen, wake up the device, or recognize the movement more precisely. The matrix control enables to recognize these control gestures:

- swipe up
- swipe down
- swipe left
- swipe right
- cover
- double tap
- zoom in
- zoom out

The conditions to recognize the swipe gestures:

- buffered gesture must be longer than three recognized positions horizontally or vertically (for one finger)
- recognized positions horizontally or vertically and the movement in dominant direction must cover twice the amount recognized positions than in insignificant direction

The double tap gesture enables to define the maximum time interval between taps. To recognize the cover gesture, each touchpad electrode must be touched. The conditions to recognize the zoom in/out gestures:

- buffered gesture must be longer than three recognized positions horizontally or vertically (for both fingers)
- square of the position distance change must be higher(lower) then 15(-15) to increment zoom in(out) counter
- the number of zoom in(out) counters must be higher then 1 and the ration between zoom in and zoom out counters must be double at least

You can store an arbitrary number of unlock gestures, limited by the free memory. The unlock gestures have higher priority than the control gestures. If both the unlock and control gestures are recognized, the control gesture is ignored. The unlock gesture means a continuous curve drawn by one finger or two continuous curves drawn by two fingers. The gesture can be stored from the drawn pattern into the RAM memory by changing the selected gesture variable (0 means no gesture to store; the number specifies which gesture number is stored). For permanent usage of the drawn gestures, the stored gestures must be copied or rewritten as matrix gesture parameters to be stored into the flash memory.

After the release event, the actual drawn gesture is buffered and compared to the stored unlock gestures (if any). The dissimilarity is a value that provides information about how much the actual gesture is similar to the stored ones. In case of a calculated dissimilarity is lower the dissimilarity_max parameter defined in matrix constants, a specified gesture is recognized. A dissimilarity value of 0 means that the gestures are the same.

When drawing the gesture, the finger (or fingers) position changes are stored to a buffer. The maximum position resolution is 16 values (hexadecimal value) for each dimension. If the matrix is too small, the resolution can be extended by creating virtual electrodes between the real ones. This feature is enabled by defining the value of "true" to the increase_resolution matrix constant. The conditions to recognize or store the unlock gestures:

- buffered gesture must be longer than four recognized positions horizontally or vertically (for one or two fingers)
- buffered gesture must be longer than four position changes to have enough recorded points for cooperation,
- buffered gesture must be shorter than sixteen position changes to not exceed the buffer size

The movement on the matrix surface is buffered in such way that each position change is stored into 64-bit x and y buffers in the right to left order as a hexadecimal value. Therefore, the same gestures drawn by a different speed are not the same because the buffer size is different. Considered this when a gesture is not recognized. The gesture-comparison algorithm is based on the summation of the square point to point distances.

The matrix control is represented by the nt_control_matrix structure. Collaboration diagram for Matrix Control:

Data Structures

- struct nt_control_matrix_data

Macros

- #define NT_MATRIX_UNLOCK_GESTURES

Enumerations

- enum nt_control_matrix_flags {
  NT_MATRIX_RELEASE_FLAG,
  NT_MATRIX_TOUCH_FLAG,
  NT_MATRIX_MOVEMENT_FLAG,
  NT_MATRIX_SET_UNLOCK_GESTURE_FLAG,
  NT_MATRIX_GET_UNLOCK_GESTURE_FLAG,
  NT_MATRIX_GET_CONTROL_GESTURE_FLAG }

8.9.7.2 Data Structure Documentation

8.9.7.2.1 struct nt_control_matrix_data

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

Allocate this structure and put a pointer into the nt_control_matrix structure when it is being registered in the system.

Collaboration diagram for nt_control_matrix_data:

Data Fields

nt_points8	actual_position	Touch-pad touch position on the x and y axes
nt_points64	buffer	Buffer to store movement on x and y axis.
uint8_t	buffer_position	Store the position into the buffer
nt_control_-matrix_-callback	callback	Matrix callback handler.

uint32_t	double_tap_-next_time	Time instant for the second tap to make a valid double tap
uint16_t	gesture_diff[NT_MATRIX_-UNLOCK_GESTURES]	Calculated differences between the stored and buffered gestures
uint8_t	gesture_-number	Number of stored gesture
uint8_t	get_position_-add_val[6]	Constants for the get_position function defined due to the increase-_resolution constant
uint64_t	last_electode_-states	Last electrodes states
uint8_t	recognized_-gesture	Number code of the recognized gesture
nt_points8	release_-position	Touch-pad release position on the x and y axes
uint8_t	select_gesture	Number code of the gesture to be stored
nt_points8	touch_position	Touch-pad touch position on the x and y axes
uint64_t	touchpad_mask	Internal mask defining the touch-pad electrodes used (horizontal and vertical electrodes must be next to each other)
uint16_t	touchpad-_mask_-horizontal	Touch-pad mask for the horizontal coordinate
uint16_t	two_points_-gesture_flag	Indicates two-point gesture

8.9.7.3 Macro Definition Documentation

8.9.7.3.1 #define NT_MATRIX_UNLOCK_GESTURES

8.9.7.4 Enumeration Type Documentation

8.9.7.4.1 enum nt_control_matrix_flags

Matrix Matrix control flags.

Enumerator

NT_MATRIX_RELEASE_FLAG Release event
NT_MATRIX_TOUCH_FLAG Key-touch event
NT_MATRIX_MOVEMENT_FLAG Movement event

NT_MATRIX_SET_UNLOCK_GESTURE_FLAG New gesture is stored
NT_MATRIX_GET_UNLOCK_GESTURE_FLAG Stored gesture is recognized
NT_MATRIX_GET_CONTROL_GESTURE_FLAG Control gesture is recognized

8.9.8 Proxi Control

8.9.8.1 Overview

The proxi control enables the detection of a finger or object presence in the near field of the electrode, so that the approaching finger/hand can be detected without a direct electrode touch. Moreover, the proximity position and direction of the movement can be evaluated for each proximity electrode. It is represented by the nt_control structure.

The proxi control uses a single electrode or a set of discrete electrodes to enable the calculation of the finger proximity position within a near field area. If more electrodes are enabled for the proximity control, the proxi detection algorithm processes each electrode separately and localizes the electrode with the highest delta signal to detect the maximum proximity electrode from the group for a single signal requirement to process the whole group. The signals from each electrode are then used for the evaluation of the proximity position, movement, and its direction. The electrode index with the maximum proximity is stored to the buffer at each maximum proximity electrode index change. The history of the maximum proximity changes consists of the same number of steps as the number of the proximity electrodes. The history stored in the buffer can be used for a simple gesture evaluation so that the buffered history is compared to the stored histories (stored gesture) and in case the same gesture is recognized. Using this feature, not only the vertical movement, but also the horizontal movement can be recognized. The gesture can be stored from the captured finger movement by modifying the gesture number variable that stores the last buffer gesture to a gesture array. Another way to store a gesture is to write the gestures directly to the proxi configuration as the array.

The proxi control provides the position, direction, and displacement values of all proximity electrodes and the buffered history of the maximum proximity changes in time. It can generate the event callbacks when the finger movement, initial-touch, or release is detected for each electrode and if the gesture is recognized. The image below shows the proxi electrode functionality.

Figure 8.9.6: Proxi Electrode

Collaboration diagram for Proxi Control:

Data Structures

- struct nt_control_proxi_data

Macros

- #define NT_PROXI_ELECTRODES
- #define NT_PROXI_GESTURES

Enumerations

- enum nt_control_prox_flags {
  NT_PROXI_DIRECTION_FLAG,
  NT_PROXI_MOVEMENT_FLAG,
  NT_PROXI_TOUCH_FLAG,
  NT_PROXI_RELEASE_FLAG }

Functions

- static void _nt_control_proxi_set_flag (uint32_t *electrode, uint32_t flags)
- static void _nt_control_proxi_clear_flag (uint32_t *electrode, uint32_t flags)
- static uint32_t _nt_control_proxi_get_flag (uint32_t *electrode, uint32_t flags)

8.9.8.2 Data Structure Documentation

8.9.8.2.1 struct nt_control_proxi_data

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

Allocate this structure and put a pointer into the nt_control_proxi structure when it is being registered in the system.

Collaboration diagram for nt_control_proxi_data:

Data Fields

uint32_t	buffer	Buffer to store index key with maximum proximity each step to hex symbol from right to left
uint32_t	buffer_mask	Buffer mask to clean longer buffer history then electrode number
nt_control_-proxi_callback	callback	Proxi callback handler
uint8_t	gesture_-number	Number of stored gesture

uint8_t	index	Proximity active key index with the maximum proximity
uint8_t	index_k_1	Previous maximum proximity key index
uint8_t	proxi_curr[NT_PROXI_ELECTRODES]	Current scaled proximity value for each electrode
uint32_t	proxi_flags[NT_PROXI_ELECTRODES]	State flags for each electrode
uint16_t	proxi_max[NT_PROXI_ELECTRODES]	Maximum proximity for each electrode
uint8_t	proxi_movement[NT_PROXI_ELECTRODES]	Proximity movement level for each electrode
uint8_t	proximity	Maximum proximity value of the index electrode
uint8_t	recognized_gesture	Recognized gesture number to indicate
uint8_t	select_gesture	Selected gesture number for storing gesture

8.9.8.3 Macro Definition Documentation

8.9.8.3.1 #define NT_PROXI_ELECTRODES

8.9.8.3.2 #define NT_PROXI_GESTURES

8.9.8.4 Enumeration Type Documentation

8.9.8.4.1 enum nt_control_prox_flags

Proxi proximity flags.

Enumerator

NT_PROXI_DIRECTION_FLAG Proxi direction flag
NT_PROXI_MOVEMENT_FLAG Proxi movement flag
NT_PROXI_TOUCH_FLAG Proxi touch flag
NT_PROXI_RELEASE_FLAG Proxi release flag

8.9.8.5 Function Documentation

8.9.8.5.1 static void _nt_control_proxi_clear_flag (uint32_t * *electrode,* uint32_t *flags*) [inline], [static]

8.9.8.5.2 static uint32_t _nt_control_proxi_get_flag (uint32_t * *electrode,* uint32_t *flags*) [inline], [static]

8.9.8.5.3 static void _nt_control_proxi_set_flag (uint32_t * *electrode,* uint32_t *flags*) [inline], [static]

8.9.9 Rotary Control

8.9.9.1 Overview

Rotary enables the detection of jog-dial-like finger movement using discrete electrodes; it is represented by the nt_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 8.9.7: Rotary Electrodes

Collaboration diagram for Rotary Control:

Data Structures

- struct nt_control_rotary_data

Enumerations

- enum nt_control_rotary_flags {
  NT_ROTARY_INVALID_POSITION_FLAG,
  NT_ROTARY_DIRECTION_FLAG,
  NT_ROTARY_MOVEMENT_FLAG,
  NT_ROTARY_TOUCH_FLAG }

8.9.9.2 Data Structure Documentation

8.9.9.2.1 struct nt_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 nt_control_rotary_control structure when it is being registered in the system.

Collaboration diagram for nt_control_rotary_data:

Data Fields

nt_control_-rotary_callback	callback	Callback handler.
uint8_t	position	Position.

8.9.9.3 Enumeration Type Documentation

8.9.9.3.1 enum nt_control_rotary_flags

Rotary flags.

Enumerator

NT_ROTARY_INVALID_POSITION_FLAG Rotary invalid position flag.
NT_ROTARY_DIRECTION_FLAG Rotary direction flag.
NT_ROTARY_MOVEMENT_FLAG Rotary movement flag.
NT_ROTARY_TOUCH_FLAG Rotary touch flag.

8.9.10 Slider Control

8.9.10.1 Overview

Slider control enables the detection of a linear finger movement using discrete electrodes; it is represented by the nt_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 8.9.8: Slider Electrodes

Collaboration diagram for Slider Control:

Data Structures

- struct nt_control_slider_data

Enumerations

- enum nt_control_slider_flags {
  NT_SLIDER_INVALID_POSITION_FLAG,
  NT_SLIDER_DIRECTION_FLAG,
  NT_SLIDER_MOVEMENT_FLAG,
  NT_SLIDER_TOUCH_FLAG }

8.9.10.2 Data Structure Documentation

8.9.10.2.1 struct nt_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 nt_control_slider_control structure when it is being registered in the system.

Collaboration diagram for nt_control_slider_data:

Data Fields

nt_control_-slider_callback	callback	Slider Callback handler.
uint8_t	position	Position.

8.9.10.3 Enumeration Type Documentation

8.9.10.3.1 enum nt_control_slider_flags

Slider flags.

Enumerator

NT_SLIDER_INVALID_POSITION_FLAG Slider invalid position flag.
NT_SLIDER_DIRECTION_FLAG Slider direction flag.
NT_SLIDER_MOVEMENT_FLAG Slider movement flag.
NT_SLIDER_TOUCH_FLAG Slider touch flag.

8.10 Controls

8.10.1 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 nt_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
- Matrix Control
- Proxi Control
- Rotary Control
- Slider Control
- General API

8.10.2 General API

8.10.2.1 Overview

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

Modules

- API Functions

Data Structures

- union nt_control_special_data
- struct nt_control_data
- struct nt_control_interface

Enumerations

- enum nt_control_flags {
  NT_CONTROL_NEW_DATA_FLAG,
  NT_CONTROL_EN_FLAG }

8.10.2.2 Data Structure Documentation

8.10.2.2.1 union nt_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.

Collaboration diagram for nt_control_special_data:

Data Fields

struct nt_control_-arotary_data *	arotary	Pointer to the Analog Rotary control special data.
struct nt_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 nt_control_-keypad_data *	keypad	Pointer to the Keypad control special data.
struct nt_control_-matrix_data *	matrix	Pointer to the Matrix control special data.
struct nt_control_-proxi_data *	proxi	Pointer to the Slider control special data.
struct nt_control_-rotary_data *	rotary	Pointer to the Rotary control special data.
struct nt_control_-slider_data *	slider	Pointer to the Slider control special data.

8.10.2.2.2 struct nt_control_data

The Control RAM structure used to store volatile parameters, flags, and other data to enable a generic behaviour of the Control. You must allocate this structure and put a pointer into the nt_control structure when the control is being registered in the system.

Collaboration diagram for nt_control_data:

Data Fields

union nt_control_-special_data	data	The pointer to the control data structure.
struct nt_-electrode_data **	electrodes	List of electrodes. Can't be NULL.
uint8_t	electrodes_size	Number of electrodes.
uint32_t	flags	Flags.
struct nt_control *	rom	The pointer to user control parameter structure.

8.10.2.2.3 struct nt_control_interface

Control interface structure; each control uses this structure to register the entry points to its algorithms. This approach enables a kind-of polymorphism in the touch System. All controls are processed in the same way from the System layer, regardless of the specific implementation. Each control type defines one static constant structure of this type to register its own initialization and processing functions.

Collaboration diagram for nt_control_interface:

Data Fields

- int32_t(∗ init)(struct nt_control_data ∗control)
- int32_t(∗ process)(struct nt_control_data ∗control)
- const char ∗ name
- const uint32_t params_size

8.10.2.2.3.1 Field Documentation

8.10.2.2.3.1.1 int32_t(∗ nt_control_interface::init)(struct nt_control_data ∗control)

The address of init function.

8.10.2.2.3.1.2 const char∗ nt_control_interface::name

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

8.10.2.2.3.1.3 const uint32_t nt_control_interface::params_size

Structure size

8.10.2.2.3.1.4 int32_t(∗ nt_control_interface::process)(struct nt_control_data ∗control)

The address of process function.

8.10.2.3 Enumeration Type Documentation

8.10.2.3.1 enum nt_control_flags

Controls flags which can be set / cleared.

Enumerator

NT_CONTROL_NEW_DATA_FLAG Indication flag that the control has new data.
NT_CONTROL_EN_FLAG Control Enable flag.

8.10.2.4 API Functions

8.10.2.4.1 Overview

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

Functions

- struct nt_control_data * _nt_control_get_data (const struct nt_control *control)
  Get control data structure pointer.
- struct nt_control_data * _nt_control_init (const struct nt_control *control)
  Initialize the control object.
- int32_t _nt_control_check_data (const struct nt_control_data *control)
  Validate control data.
- uint64_t _nt_control_get_electrodes_state (struct nt_control_data *control)
  Get the state of all control electrodes.
- uint32_t _nt_control_get_electrodes_digital_state (struct nt_control_data *control)
  Get the digital state of all control electrodes.
- void _nt_control_set_flag_all_elec (struct nt_control_data *control, uint32_t flags)
- void _nt_control_clear_flag_all_elec (struct nt_control_data *control, uint32_t flag)
- uint32_t _nt_control_get_first_elec_touched (uint32_t current_state)
- uint32_t _nt_control_get_last_elec_touched (uint32_t current_state)
- uint32_t _nt_control_get_touch_count (uint32_t current_state)
- int32_t _nt_control_check_neighbours_electrodes (struct nt_control_data *control, uint32_t first, uint32_t second, uint32_t overrun)
- int32_t _nt_control_check_edge_electrodes (struct nt_control_data *control, uint32_t electrode_ix)
- static void _nt_control_set_flag (struct nt_control_data *control, uint32_t flags)
- static void _nt_control_clear_flag (struct nt_control_data *control, uint32_t flags)
- static uint32_t _nt_control_get_flag (const struct nt_control_data *control, uint32_t flags)
- static int32_t _nt_control_overrun (struct nt_control_data *control)
- static struct nt_electrode * _nt_control_get_electrode (const struct nt_control_data *control, uint32-_t index)

8.10.2.4.2 Function Documentation

8.10.2.4.2.1 int32_t _nt_control_check_data (const struct nt_control_data * *control*)

Parameters

control	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:

8.10.2.4.2.2 int32_t _nt_control_check_edge_electrodes (struct nt_control_data ∗ *control,* uint32_t *electrode_ix*)

8.10.2.4.2.3 int32_t _nt_control_check_neighbours_electrodes (struct nt_control_data ∗ *control,* uint32_t *first,* uint32_t *second,* uint32_t *overrun*)

8.10.2.4.2.4 static void _nt_control_clear_flag (struct nt_control_data ∗ *control,* uint32_t *flags*) `[inline], [static]`

Here is the caller graph for this function:

8.10.2.4.2.5 void _nt_control_clear_flag_all_elec (struct nt_control_data ∗ *control,* uint32_t *flag*)

Here is the call graph for this function:

8.10.2.4.2.6 struct nt_control_data∗ _nt_control_get_data (const struct nt_control ∗ *control*)

Parameters

control	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:

8.10.2.4.2.7 static struct nt_electrode* _nt_control_get_electrode (const struct nt_control_data * *control,* uint32_t *index*) `[static]`

8.10.2.4.2.8 uint32_t _nt_control_get_electrodes_digital_state (struct nt_control_data * *control*)

Parameters

control	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 electrode digital result(not analog value).

Here is the call graph for this function:

8.10.2.4.2.9 uint64_t _nt_control_get_electrodes_state (struct nt_control_data ∗ *control*)

Parameters

control	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:

8.10.2.4.2.10 uint32_t _nt_control_get_first_elec_touched (uint32_t *current_state*)

8.10.2.4.2.11 static uint32_t _nt_control_get_flag (const struct nt_control_data ∗ *control,* uint32_t *flags*) [inline], [static]

Here is the caller graph for this function:

8.10.2.4.2.12 uint32_t _nt_control_get_last_elec_touched (uint32_t *current_state*)

8.10.2.4.2.13 uint32_t _nt_control_get_touch_count (uint32_t *current_state*)

8.10.2.4.2.14 struct nt_control_data∗ _nt_control_init (const struct nt_control ∗ *control*)

Parameters

control	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:

8.10.2.4.2.15 static int32_t _nt_control_overrun (struct nt_control_data * *control*) [inline], [static]

Here is the call graph for this function:

Here is the caller graph for this function:

8.10.2.4.2.16 static void _nt_control_set_flag (struct nt_control_data * *control,* uint32_t *flags*) [inline], [static]

Here is the caller graph for this function:

8.10.2.4.2.17 void _nt_control_set_flag_all_elec (struct nt_control_data * *control,* uint32_t *flags*)

Here is the call graph for this function:

8.11 CS Drivers

8.11.1 Overview

Drivers represent the lowest level of abstraction in Capacitive sense (CS) peripheral control and contain common and CS version specific files. There are functions like NT_CS_DRV_Init, NT_CS_DRV_Enable-Electrode, NT_CS_DRV_Measure, etc. Collaboration diagram for CS Drivers:

Modules

- API Functions

Data Structures

- struct nt_cs_user_config_t
- struct nt_cs_operation_mode_t
- struct nt_cs_state_t

Typedefs

- typedef void(∗ cs_callback_t)(uint32_t instance, void ∗usrData)
  Call back routine of CS driver.

Enumerations

- enum cs_status_t {
  kStatus_CS_Success,
  kStatus_CS_Busy,
  kStatus_CS_Overflow,
  kStatus_CS_Overrun,
  kStatus_CS_LowPower,
  kStatus_CS_InvalidChannel,
  kStatus_CS_InvalidMode,
  kStatus_CS_Initialized,
  kStatus_CS_Error }

- enum nt_cs_modes_t {
  cs_OpModeNormal,
  cs_OpModeLowPower,
  cs_OpModeProximity,
  cs_OpModeCnt,
  cs_OpModeNoChange }

Variables

- CS_Type *const g_csBase [ ]
- const IRQn_Type g_csIrqId [FSL_FEATURE_SOC_CS_COUNT]
- nt_cs_state_t * g_csStatePtr [FSL_FEATURE_SOC_CS_COUNT]

8.11.2 Data Structure Documentation

8.11.2.1 struct nt_cs_user_config_t

User configuration structure for CS driver.

Use an instance of this structure with NT_CS_DRV_Init(). This allows you to configure the most common settings of the CS peripheral with a single function call. Settings include:

Collaboration diagram for nt_cs_user_config_t:

Data Fields

cs_config_t *	config	A pointer to hardware configuration. Can't be NULL.
cs_callback_t	pCallBackFunc	A pointer to call back function of end of measurement.
void *	usrData	A user data of call back function.

8.11.2.2 struct nt_cs_operation_mode_t

Driver operation mode data hold structure.

This is the operation mode data hold structure. The structure is keep all needed data to be driver able to switch the operation modes and properly set up HW peripheral.

Collaboration diagram for nt_cs_operation_mode_t:

Data Fields

cs_config_t	config	A hardware configuration.
uint8_t	enabled-Electrodes	The back up of enabled electrodes for operation mode

8.11.2.3 struct nt_cs_state_t

Driver data storage place.

It must be created by the application code and the pointer is handled by the NT_CS_DRV_Init function to driver. The driver keeps all context data for itself run. Settings include:

Collaboration diagram for nt_cs_state_t:

Data Fields

uint16_t	counters[FSL_-FEATURE_C-S_CHANNEL-_COUNT]	The mirror of last state of counter registers
nt_mutex_t	lock	Used by whole driver to secure the context data integrity.
nt_mutex_t	lockChange-Mode	Used by change mode function to secure the context data integrity.
nt_cs_modes_t	opMode	Storage of current operation mode.
nt_cs_-operation_-mode_t	opModes-Data[cs_Op-ModeCnt]	Data storage of individual operational modes.
cs_callback_t	pCallBackFunc	A pointer to call back function of end of measurement.
cs_status_t	status	Current status of the driver.
void *	usrData	A user data pointer handled by call back function.

8.11.3 Typedef Documentation

8.11.3.1 typedef void(∗ cs_callback_t)(uint32_t instance, void ∗usrData)

The function is called on end of the measure of the CS driver. The function can be called from interrupt, so the code inside the callback should be short and fast.

Parameters

instance	- instance of the CS peripheral
usrData	- user data (type is void∗), the user data are specified by function NT_CS_DRV_SetCallBackFunc

Returns

- none

8.11.4 Enumeration Type Documentation

8.11.4.1 enum cs_status_t

Error codes for the CS driver.

Enumerator

kStatus_CS_Success
kStatus_CS_Busy CS still in progress
kStatus_CS_Overflow CS counter out of range
kStatus_CS_Overrun CS measurement overrun
kStatus_CS_LowPower CS is in low power mode
kStatus_CS_InvalidChannel Invalid CS channel
kStatus_CS_InvalidMode Invalid CS mode
kStatus_CS_Initialized The driver is initialized and ready to measure
kStatus_CS_Error The general driver error

8.11.4.2 enum nt_cs_modes_t

Driver operation mode definition.

The operation name definition used for CS driver.

Enumerator

cs_OpModeNormal The normal active mode of CS.
cs_OpModeLowPower The low power mode of CS.
cs_OpModeProximity The proximity sensing mode of CS.
cs_OpModeCnt Count of CS modes - for internal use.
cs_OpModeNoChange The special value of operation mode that allows call for example NT_CS_DRV_DisableLowPower function without change of operation mode.

8.11.5 Variable Documentation

8.11.5.1 CS_Type∗ const g_csBase[]

Table of base addresses for CS instances.

8.11.5.2 const IRQn_Type g_csIrqId[FSL_FEATURE_SOC_CS_COUNT]

Table to save CS IRQ enumeration numbers defined in CMSIS header file.

8.11.5.3 nt_cs_state_t∗ g_csStatePtr[FSL_FEATURE_SOC_CS_COUNT]

Table to save pointers to context data.

8.11.6 API Functions

8.11.6.1 Overview

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

Functions

- cs_status_t NT_CS_DRV_Init (uint32_t instance, nt_cs_state_t *csState, const nt_cs_user_config_t *csUserConfig)
  Initializes a CS instance for operation.
- cs_status_t NT_CS_DRV_DeInit (uint32_t instance)
  Shuts down the CS by disabling interrupts and the peripheral.
- cs_status_t NT_CS_DRV_EnableElectrode (uint32_t instance, const uint32_t channel, const bool enable)
  Enables/disables one electrode of the CS module.
- uint8_t NT_CS_DRV_GetEnabledElectrodes (uint32_t instance)
  Returns a mask of the enabled electrodes of the CS module.
- cs_status_t NT_CS_DRV_Measure (uint32_t instance)
  Starts the measure cycle of the enabled electrodes.
- cs_status_t NT_CS_DRV_GetCounter (uint32_t instance, const uint32_t channel, uint16_t *counter)
  Returns the last measured value.
- cs_status_t NT_CS_DRV_GetStatus (uint32_t instance)
  Returns the current status of the driver.
- cs_status_t NT_CS_DRV_EnableLowPower (uint32_t instance)
  Enters the low power mode of the CS driver.
- cs_status_t NT_CS_DRV_DisableLowPower (uint32_t instance, const nt_cs_modes_t mode)
  This function returns back the CS driver from the low power to standard operation.
- cs_status_t NT_CS_DRV_SetCallBackFunc (uint32_t instance, const cs_callback_t pFuncCallBack, void *usrData)
  Sets the callback function that is called when the measure cycle ends.
- cs_status_t NT_CS_DRV_ChangeMode (uint32_t instance, const nt_cs_modes_t mode)
  Changes the current working operation mode.
- nt_cs_modes_t NT_CS_DRV_GetMode (uint32_t instance)
  Returns the current working operation mode.
- cs_status_t NT_CS_DRV_LoadConfiguration (uint32_t instance, const nt_cs_modes_t mode, const nt_cs_operation_mode_t *operationMode)
  Loads the new configuration into a specific mode.

- cs_status_t NT_CS_DRV_SaveConfiguration (uint32_t instance, const nt_cs_modes_t mode, nt_cs_operation_mode_t *operationMode)
  Saves the CS driver configuration for a specific mode.

8.11.6.2 Function Documentation

8.11.6.2.1 cs_status_t NT_CS_DRV_ChangeMode (uint32_t *instance*, const nt_cs_modes_t *mode*)

This function changes the working operation mode of the driver.

```
// Change operation mode to low power

if(NT_CS_DRV_ChangeMode(0, cs_OpModeLowPower) !=
      kStatus_CS_Success)
{
    // Error, the CS driver can't change the operation mode into low power
}
```

Parameters

instance	The CS module instance.
mode	The requested new operation mode

Returns

An error code or kStatus_CS_Success.

8.11.6.2.2 cs_status_t NT_CS_DRV_DeInit (uint32_t *instance*)

This function disables the CS interrupts and the peripheral.

```
if(NT_CS_DRV_DeInit(0) != kStatus_CS_Success)
{
   // Error, the CS is not de-initialized
}
```

Parameters

instance	The CS module instance.

Returns

An error code or kStatus_CS_Success.

8.11.6.2.3 cs_status_t NT_CS_DRV_DisableLowPower (uint32_t *instance,* const nt_cs_modes_t *mode*)

Function switch the driver back form low power mode and it can immediately change the operation mode to any other or keep the driver in low power configuration, to be able go back to low power state.

```
// Switch the driver from the low power mode

if(NT_CS_DRV_DisableLowPower(0, cs_OpModeNormal) !=
      kStatus_CS_Success)
{
    // Error, the CS driver can't go from low power mode
}
```

Parameters

instance	The CS module instance.
mode	The new operation mode request

Returns

An error code or kStatus_CS_Success.

8.11.6.2.4 cs_status_t NT_CS_DRV_EnableElectrode (uint32_t *instance,* const uint32_t *channel,* const bool *enable*)

Function must be called for each used electrodes after initialization of CS driver.

```
    // On the CS instance 0, enable electrode with index 5
if(NT_CS_DRV_EnableElectrode(0, 5, TRUE) !=
      kStatus_CS_Success)
{
    // Error, the CS 5'th electrode is not enabled
}
```

Parameters

instance	The CS module instance.
channel	Index of CS channel.
enable	TRUE - for channel enable, FALSE for disable.

Returns

An error code or kStatus_CS_Success.

8.11.6.2.5 cs_status_t NT_CS_DRV_EnableLowPower (uint32_t *instance*)

This function switches the driver to low power mode and immediately enables the low power functionality of the CS peripheral. Before calling this function, the low power mode must be configured - Enable the right electrode and recalibrate the low power mode to get the best performance for this mode.

```
// Switch the driver to the low power mode
uint16_t signal;

// The first time is needed to configure the low power mode configuration

(void)NT_CS_DRV_ChangeMode(0, cs_OpModeLowPower); // I don't check the
       result because I believe in.
// Enable the right one electrode for low power AKE up operation
(void)NT_CS_DRV_EnableElectrode(0, 5, true);
// Recalibrate the mode to get the best performance for this one electrode
(void)NT_CS_DRV_Recalibrate(0);

if(NT_CS_DRV_EnableLowPower(0) != kStatus_CS_Success)
{
    // Error, the CS driver can't go to low power mode
}
```

Parameters

instance	The CS module instance.

Returns

An error code or kStatus_CS_Success.

8.11.6.2.6 cs_status_t NT_CS_DRV_GetCounter (uint32_t *instance,* const uint32_t *channel,* uint16_t ∗ *counter*)

This function returns the last measured value in the previous measure cycle. The data is buffered inside the driver.

```
// Get the counter value from CS instance 0 and 5th channel

uint32_t result;

if(NT_CS_DRV_GetCounter(0, 5, &result) != kStatus_CS_Success)
{
    // Error, the CS 5'th electrode is not read
}
```

Parameters

instance	The CS module instance.
channel	The CS electrode index.
counter	The pointer to 16 bit value where will be stored channel counter value.

Returns

An error code or kStatus_CS_Success.

8.11.6.2.7 uint8_t NT_CS_DRV_GetEnabledElectrodes (uint32_t *instance*)

The function returns the mask of the enabled electrodes of the current mode.

```
uint32_t enabledElectrodeMask;
enabledElectrodeMask = NT_CS_DRV_GetEnabledElectrodes(0);
```

Parameters

instance	The CS module instance.

Returns

Mask of enabled electrodes for current mode.

8.11.6.2.8 nt_cs_modes_t NT_CS_DRV_GetMode (uint32_t *instance*)

This function returns the current working operation mode of the driver.

```
// Gets current operation mode of CS driver
nt_cs_modes_t mode;

mode = NT_CS_DRV_GetMode(0);
```

Parameters

instance	The CS module instance.

Returns

An current operation mode of CS driver.

8.11.6.2.9 cs_status_t NT_CS_DRV_GetStatus (uint32_t *instance*)

This function returns the current working status of the driver.

```
// Get the current status of CS driver

cs_status_t status;

status = NT_CS_DRV_GetStatus(0);
```

Parameters

instance	The CS module instance.

Returns

An current status of the driver.

8.11.6.2.10 cs_status_t NT_CS_DRV_Init (uint32_t *instance*, nt_cs_state_t * *csState*, const nt_cs_user_config_t * *csUserConfig*)

This function initializes the run-time state structure and prepares the entire peripheral to measure the capacitances on electrodes.

```
static nt_cs_state_t myCSDriverStateStructure;

nt_cs_user_config_t myCSDriveruserConfig =
{
 .config =
    {
       ...
    },
  .pCallBackFunc = APP_myCSCallBackFunc,
  .usrData = myData,
};

if(NT_CS_DRV_Init(0, &myCSDriverStateStructure, &myCSDriveruserConfig) !=
      kStatus_CS_Success)
{
   // Error, the CS is not initialized
}
```

Parameters

instance	The CS module instance.
csState	A pointer to the CS driver state structure memory. The user is only responsible to pass in the memory for this run-time state structure where the CS driver will take care of filling out the members. This run-time state structure keeps track of the current CS peripheral and driver state.
csUserConfig	The user configuration structure of type nt_cs_user_config_t. The user populates the members of this structure and passes the pointer of this structure into the function.

Returns

An error code or kStatus_CS_Success.

8.11.6.2.11 cs_status_t NT_CS_DRV_LoadConfiguration (uint32_t *instance*, const nt_cs_modes_t *mode*, const nt_cs_operation_mode_t ∗ *operationMode*)

This function loads the new configuration into a specific mode. This can be used when the calibrated data are stored in any NVM to load after startup of the MCU to avoid run recalibration that takes more time.

```
// Load operation mode configuration

extern const nt_cs_operation_mode_t * myCSNvmLowPowerConfiguration;

if(NT_CS_DRV_LoadConfiguration(0, cs_OpModeLowPower,
      myCSNvmLowPowerConfiguration) != kStatus_CS_Success)
{
    // Error, the CS driver can't load the configuration
}
```

Parameters

instance	The CS module instance.
mode	The requested new operation mode
operationMode	The pointer to storage place of the configuration that should be loaded

Returns

An error code or kStatus_CS_Success.

8.11.6.2.12 cs_status_t NT_CS_DRV_Measure (uint32_t *instance*)

The function is non blocking. Therefore, the results can be obtained after the driver completes the measure cycle. The end of the measure cycle can be checked by pooling the NT_CS_DRV_GetStatus function or wait for registered callback function by using the NT_CS_DRV_SetCallBackFunc or NT_CS_DRV_Init.

```
// Example of the pooling style of use of NT_CS_DRV_Measure() function
if(NT_CS_DRV_Measure(0) != kStatus_CS_Success)
{
    // Error, the CS 5'th electrode is not enabled
}

while(NT_CS_DRV_GetStatus(0) != kStatus_CS_Initialized)
{
    // Do something useful - don't waste the CPU cycle time
}
```

Parameters

instance	The CS module instance.

Returns

An error code or kStatus_CS_Success.

8.11.6.2.13 cs_status_t NT_CS_DRV_SaveConfiguration (uint32_t *instance*, const nt_cs_modes_t *mode*, nt_cs_operation_mode_t * *operationMode*)

This function saves the configuration of a specific mode. This can be used when the calibrated data should be stored in any backup memory to load after the start of the MCU to avoid running the recalibration that takes more time.

```
// Save operation mode configuration

extern nt_cs_operation_mode_t  myCSNvmLowPowerConfiguration;

if(NT_CS_DRV_SaveConfiguration(0, cs_OpModeLowPower, &
      myCSNvmLowPowerConfiguration) != kStatus_CS_Success)
{
    // Error, the CS driver can't save the configuration
}
```

Parameters

instance	The CS module instance.
mode	The requested new operation mode
operationMode	The pointer to storage place of the configuration that should be save

Returns

An error code or kStatus_CS_Success.

8.11.6.2.14 cs_status_t NT_CS_DRV_SetCallBackFunc (uint32_t *instance*, const cs_callback_t *pFuncCallBack*, void * *usrData*)

This function sets up or clears, (parameter pFuncCallBack = NULL), the callback function pointer which is called after each measure cycle ends. The user can also set the custom user data, that is handled by the parameter to a call back function. One function can be called by more sources.

```
// Clear previous call back function

if(NT_CS_DRV_SetCallBackFunc(0, NULL, NULL) !=
      kStatus_CS_Success)
{
    // Error, the CS driver can't set up the call back function at the moment
}

// Set new call back function

if(NT_CS_DRV_SetCallBackFunc(0, myFunction, (void*)0x12345678) !=
      kStatus_CS_Success)
{
    // Error, the CS driver can't set up the call back function at the moment
}
```

Parameters

instance	The CS module instance.
pFuncCallBack	The pointer to application call back function
usrData	The user data pointer

Returns

An error code or kStatus_CS_Success.

8.12 TSI Drivers

8.12.1 Overview

Drivers represent the lowest level of abstraction in the TSI peripheral control;

Drivers contain common and TSI version-specific files. There are functions like NT_TSI_DRV_Init, NT_-TSI_DRV_EnableElectrode, NT_TSI_DRV_Measure, and so on. Collaboration diagram for TSI Drivers:

Modules

- API Functions

Data Structures

- struct nt_tsi_user_config_t
- struct nt_tsi_operation_mode_t
- struct nt_tsi_state_t

Macros

- #define TF_TSI_TOTAL_CHANNEL_COUNT

Typedefs

- typedef void(∗ tsi_callback_t)(uint32_t instance, void ∗usrData)
  Callback routine of the TSI driver.

Enumerations

- enum tsi_status_t {
  kStatus_TSI_Success,
  kStatus_TSI_Busy,
  kStatus_TSI_Overflow,
  kStatus_TSI_Overrun,
  kStatus_TSI_LowPower,
  kStatus_TSI_Recalibration,
  kStatus_TSI_InvalidChannel,
  kStatus_TSI_InvalidMode,
  kStatus_TSI_Initialized,
  kStatus_TSI_Error,
  kStatus_TSI_SelfUntouchRecalibError,
  kStatus_TSI_SelfSensitivRecalibError,
  kStatus_TSI_MutUntouchRecalibError,
  kStatus_TSI_MutSensitivRecalibError,
  kStatus_TSI_CofigurationIsReadOnly }
- enum nt_tsi_modes_t {
  tsi_OpModeNormal,
  tsi_OpModeProximity,
  tsi_OpModeLowPower,
  tsi_OpModeNoise,
  tsi_OpModeCnt,
  tsi_OpModeNoChange }
- enum nt_tsi_op_status_t {
  tsi_OpStatusNormal,
  tsi_OpStatusSuspend }

Variables

- TSI_Type *const g_tsiBase [ ]
- const IRQn_Type g_tsiIrqId [FSL_FEATURE_SOC_TSI_COUNT]
- nt_tsi_state_t * g_tsiStatePtr [FSL_FEATURE_SOC_TSI_COUNT]

8.12.2 Data Structure Documentation

8.12.2.1 struct nt_tsi_user_config_t

User configuration structure for the TSI driver.

Use an instance of this structure with NT_TSI_DRV_Init(). This enables you to configure the most common settings of the TSI peripheral with a single function call. Settings include:

Collaboration diagram for nt_tsi_user_config_t:

Data Fields

tsi_config_t *	config	A pointer to the hardware configuration. Can't be NULL.
tsi_callback_t	pCallBackFunc	A pointer to the callback function at the end of the measurement.
void *	usrData	The user data of the callback function.

8.12.2.2 struct nt_tsi_operation_mode_t

Driver operation mode data hold structure.

This is the operation mode data hold structure. The structure is keeping all data needed for the driver to be able to switch the operation modes and properly set up the HW peripheral.

Collaboration diagram for nt_tsi_operation_mode_t:

Data Fields

tsi_config_t	config	Hardware configuration
uint64_t	enabled-Electrodes	The backup of enabled electrodes for the operation mode

8.12.2.3 struct nt_tsi_state_t

Driver data storage place.

It must be created by the application code and the pointer is handled by the NT_TSI_DRV_Init function to the driver. The driver keeps all context data for itself running. The settings include:

Collaboration diagram for nt_tsi_state_t:

Data Fields

nt_mutex_t	lock	Used by the whole driver to secure the context data integrity.
nt_mutex_t	lockChange-Mode	Used by the change mode function to secure the context data integrity.

nt_tsi_modes_t	opMode	Storage of current operation mode.
nt_tsi_-operation_-mode_t	opModes-Data[tsi_Op-ModeCnt]	Data storage of individual operational modes.
nt_tsi_op_-status_t	opSatus	Suspended or Normal operation
tsi_callback_t	pCallBackFunc	Pointer to the callback function at the end of the measurement.
tsi_status_t	status	Current status of the driver.
void ∗	usrData	User data pointer handled by the callback function.

8.12.3 Macro Definition Documentation

8.12.3.1 #define TF_TSI_TOTAL_CHANNEL_COUNT

The total number of TSI channels corresponds to FSL_FEATURE_TSI_CHANNEL_COUNT if not defined elsewhere.

8.12.4 Typedef Documentation

8.12.4.1 typedef void(∗ tsi_callback_t)(uint32_t instance, void ∗usrData)

The function is called at the end of the measure of the TSI driver. The function can be called from an interrupt, so the code inside the callback should be short and fast.

Parameters

instance	- instance of the TSI peripheral
usrData	- user data (type is void∗), the user data are specified by function NT_TSI_DRV_Set-CallBackFunc

Returns

- none

8.12.5 Enumeration Type Documentation

8.12.5.1 enum nt_tsi_modes_t

Driver operation mode definition.

The operation name definition used for the TSI driver.

Enumerator

tsi_OpModeNormal The normal mode of TSI.
tsi_OpModeProximity The proximity-sensing mode of TSI.
tsi_OpModeLowPower The low-power mode of TSI.
tsi_OpModeNoise The noise mode of TSI. This mode is not valid with TSI HW, valid only for the TSIL HW.
tsi_OpModeCnt The count of TSI modes - for internal use.
tsi_OpModeNoChange The special value of operation mode that allows calling for example the N-T_TSI_DRV_DisableLowPower function without changing the operation mode.

8.12.5.2 enum nt_tsi_op_status_t

Driver operation status definition.

The operation status definition used for the TSI driver.

Enumerator

tsi_OpStatusNormal The normal mode of TSI.
tsi_OpStatusSuspend TSI mesurement suspended

8.12.5.3 enum tsi_status_t

Error codes for the TSI driver.

Enumerator

kStatus_TSI_Success
kStatus_TSI_Busy TSI still in progress
kStatus_TSI_Overflow TSI counter out of range
kStatus_TSI_Overrun TSI measurement overrun
kStatus_TSI_LowPower TSI is in low-power mode
kStatus_TSI_Recalibration TSI is under recalibration process
kStatus_TSI_InvalidChannel Invalid TSI channel
kStatus_TSI_InvalidMode Invalid TSI mode
kStatus_TSI_Initialized The driver is initialized and ready to measure
kStatus_TSI_Error The general driver error
kStatus_TSI_SelfUntouchRecalibError The recalibration self-cap untouched calibration error
kStatus_TSI_SelfSensitivRecalibError The recalibration self-cap sensitivity calibration error
kStatus_TSI_MutUntouchRecalibError The recalibration mutual-cap untouched calibration error
kStatus_TSI_MutSensitivRecalibError The recalibration mutual-cap sensitivity calibration error
kStatus_TSI_CofigurationIsReadOnly The recalibration error not possible to write new cofiguration

8.12.6 Variable Documentation

8.12.6.1 TSI_Type∗ const g_tsiBase[]

Table of the base addresses for TSI instances.

8.12.6.2 const IRQn_Type g_tsiIrqId[FSL_FEATURE_SOC_TSI_COUNT]

Table to save the TSI IRQ enumeration numbers defined in the CMSIS header file.

8.12.6.3 nt_tsi_state_t∗ g_tsiStatePtr[FSL_FEATURE_SOC_TSI_COUNT]

Table to save the pointers to the context data.

8.12.7 API Functions

8.12.7.1 Overview

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

Functions

- tsi_status_t NT_TSI_DRV_Init (uint32_t instance, nt_tsi_state_t *tsiState, const nt_tsi_user_-config_t *tsiUserConfig)
  Initializes a TSI instance for operation.
- void NT_TSI_DRV_InitSpecific (TSI_Type *base, const tsi_config_t *config)
  Initializes TSI v5 in both modes (self cap, mutual).
- tsi_status_t NT_TSI_DRV_DeInit (uint32_t instance)
  Shuts down the TSI by disabling the interrupts and the peripheral.
- tsi_status_t NT_TSI_DRV_EnableElectrode (uint32_t instance, const uint32_t channel, const bool enable)
  Enables/disables one electrode of the TSI module.
- uint64_t NT_TSI_DRV_GetEnabledElectrodes (uint32_t instance)
  Returns a mask of the enabled electrodes of the TSI module.
- tsi_status_t NT_TSI_DRV_Measure (uint32_t instance)
  Starts the measure cycle of the enabled electrodes.
- tsi_status_t NT_TSI_DRV_GetStatus (uint32_t instance)
  Returns the current status of the driver.
- tsi_status_t NT_TSI_DRV_EnableLowPower (uint32_t instance)
  Enters the low-power mode of the TSI driver.
- tsi_status_t NT_TSI_DRV_DisableLowPower (uint32_t instance, const nt_tsi_modes_t mode)
  This function returns back the TSI driver from the low-power to the standard operation.
- uint16_t NT_TSI_DRV_MeasureOneElect (uint32_t instance, struct nt_electrode *electrode)
  Automatically measures one electrode.
- tsi_status_t NT_TSI_DRV_MeasureAllElect (uint32_t instance, uint16_t(*counters)[ ])
  Automatically measures all used TSI electrodes.
- tsi_status_t NT_TSI_DRV_MeasureMoreElect (uint32_t instance, uint16_t(*counters)[ ], struct nt-_electrode **electrodes, uint8_t elNumber)
  Automatically measures selected TSI electrodes.
- tsi_status_t NT_TSI_DRV_Recalibrate (uint32_t instance, void *configuration)
  Automatically recalibrates all important TSI settings.
- tsi_status_t NT_TSI_DRV_SetCallBackFunc (uint32_t instance, const tsi_callback_t pFuncCall-Back, void *usrData)

Sets the callback function that is called when the measure cycle ends.

- tsi_status_t NT_TSI_DRV_ChangeMode (uint32_t instance, const nt_tsi_modes_t mode)
  Changes the current working operation mode.
- nt_tsi_modes_t NT_TSI_DRV_GetMode (uint32_t instance)
  Returns the current working operation mode.
- tsi_status_t NT_TSI_DRV_LoadConfiguration (uint32_t instance, const nt_tsi_modes_t mode, const nt_tsi_operation_mode_t *operationMode)
  Loads the new configuration into a specific mode.
- tsi_status_t NT_TSI_DRV_SaveConfiguration (uint32_t instance, const nt_tsi_modes_t mode, nt_tsi_operation_mode_t *operationMode)
  Saves the TSI driver configuration for a specific mode.
- tsi_status_t NT_TSI_DRV_Suspend (uint32_t instance)
  Temporary block of TSI measurement.
- tsi_status_t NT_TSI_DRV_Resume (uint32_t instance)
  Resumes measurement after Suspend.

8.12.7.2 Function Documentation

8.12.7.2.1 tsi_status_t NT_TSI_DRV_ChangeMode (uint32_t *instance*, const nt_tsi_modes_t *mode*)

This function changes the working operation mode of the driver.

```
// Change the operation mode to low-power

if(NT_TSI_DRV_ChangeMode(0, tsi_OpModeLowPower) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot change the operation mode into low-power
}
```

Parameters

instance	The TSI module instance.
mode	The requested new operation mode.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.2 tsi_status_t NT_TSI_DRV_DeInit (uint32_t *instance*)

This function disables the TSI interrupts and the peripheral.

```
if(NT_TSI_DRV_DeInit(0) != kStatus_TSI_Success)
{
   // Error, the TSI is not de-initialized
}
```

Parameters

instance	The TSI module instance.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.3 tsi_status_t NT_TSI_DRV_DisableLowPower (uint32_t *instance,* const nt_tsi_modes_t *mode*)

Function switches the driver back from the low-power mode and it can immediately change the operation mode to any other or keep the driver in the low-power configuration to be able go back to the low-power state.

```
// Switch the driver from the low-power mode

if(NT_TSI_DRV_DisableLowPower(0, tsi_OpModeNormal) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot go from the low-power mode
}
```

Parameters

instance	The TSI module instance.
mode	The new operation mode request.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.4 tsi_status_t NT_TSI_DRV_EnableElectrode (uint32_t *instance,* const uint32_t *channel,* const bool *enable*)

The function must be called for each used electrode after the initialization of the TSI driver.

```
    // On the TSI instance 0, enable electrode with index 5
if(NT_TSI_DRV_EnableElectrode(0, 5, TRUE) !=
      kStatus_TSI_Success)
{
    // Error, the TSI 5'th electrode is not enabled
}
```

Parameters

instance	TSI module instance.
channel	Index of the TSI channel.
enable	TRUE - for channel enable, FALSE for disable.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.5 tsi_status_t NT_TSI_DRV_EnableLowPower (uint32_t *instance*)

This function switches the driver to the low-power mode and immediately enables the low-power functionality of the TSI peripheral. Before calling this function, the low-power mode must be configured - Enable the right electrode and recalibrate the low-power mode to get the best performance for this mode.

```
// Switch the driver to the low-power mode
uint16_t signal;

// The first time is needed to configure the low-power mode configuration

(void)NT_TSI_DRV_ChangeMode(0, tsi_OpModeLowPower); // I don't check
       the result because I believe in.
// Enable the right electrode for the low-power AKE up operation
(void)NT_TSI_DRV_EnableElectrode(0, 5, true);
// Recalibrate the mode to get the best performance for this electrode
(void)NT_TSI_DRV_Recalibrate(0);

if(NT_TSI_DRV_EnableLowPower(0) != kStatus_TSI_Success)
{
    // Error, the TSI driver can't go to the low-power mode
}
```

Parameters

instance	The TSI module instance.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.6 uint64_t NT_TSI_DRV_GetEnabledElectrodes (uint32_t *instance*)

The function returns the mask of the enabled electrodes of the current mode.

```
uint32_t enabledElectrodeMask;
enabledElectrodeMask = NT_TSI_DRV_GetEnabledElectrodes(0);
```

Parameters

instance	TSI module instance.

Returns

Mask of enabled electrodes for the current mode.

8.12.7.2.7 nt_tsi_modes_t NT_TSI_DRV_GetMode (uint32_t *instance*)

This function returns the current working operation mode of the driver.

```
// Gets the current operation mode of the TSI driver
nt_tsi_modes_t mode;

mode = NT_TSI_DRV_GetMode(0);
```

Parameters

instance	The TSI module instance.

Returns

The current operation mode of the TSI driver.

8.12.7.2.8 tsi_status_t NT_TSI_DRV_GetStatus (uint32_t *instance*)

This function returns the current working status of the driver.

```
// Get the current status of the TSI driver

tsi_status_t status;

status = NT_TSI_DRV_GetStatus(0);
```

Parameters

instance	The TSI module instance.

Returns

The current status of the driver.

8.12.7.2.9 tsi_status_t NT_TSI_DRV_Init (uint32_t *instance*, nt_tsi_state_t * *tsiState*, const nt_tsi_user_config_t * *tsiUserConfig*)

This function initializes the run-time state structure and prepares the entire peripheral to measure the capacitances on the electrodes.

```
static nt_tsi_state_t myTsiDriverStateStructure;

nt_tsi_user_config_t myTsiDriveruserConfig =
{
 .config =
    {
       ...
    },
  .pCallBackFunc = APP_myTsiCallBackFunc,
  .usrData = myData,
};

if(NT_TSI_DRV_Init(0, &myTsiDriverStateStructure, &myTsiDriveruserConfig) !=
      kStatus_TSI_Success)
{
   // Error, the TSI is not initialized
}
```

Parameters

instance	The TSI module instance.
tsiState	A pointer to the TSI driver state structure memory. The user is only responsible to pass in the memory for this run-time state structure where the TSI driver takes care of filling out the members. This run-time state structure keeps track of the current TSI peripheral and driver state.
tsiUserConfig	The user configuration structure of the nt_tsi_user_config_t type. The user populates the members of this structure and passes the pointer of this structure into the function.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.10 void NT_TSI_DRV_InitSpecific (TSI_Type * *base*, const tsi_config_t * *config*)

```
NT_TSI_DRV_InitSpecific(0, &myTsiDriveruserConfig)
```

Parameters

base	The TSI module instance.
config	The user configuration structure of the tsi_config_t type. The user populates the members of this structure and passes the pointer of this structure into the function.

Returns

none.

8.12.7.2.11 tsi_status_t NT_TSI_DRV_LoadConfiguration (uint32_t *instance*, const nt_tsi_modes_t *mode*, const nt_tsi_operation_mode_t * *operationMode*)

This function loads the new configuration into a specific mode. This can be used when the calibrated data are stored in any NVM to load after the startup of the MCU to avoid the run recalibration that takes more time.

```
// Load operation mode configuration

extern const nt_tsi_operation_mode_t * myTsiNvmLowPowerConfiguration;

if(NT_TSI_DRV_LoadConfiguration(0,
      tsi_OpModeLowPower, myTsiNvmLowPowerConfiguration) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot load the configuration
}
```

Parameters

instance	The TSI module instance.
mode	The requested new operation mode.
operationMode	The pointer to the storage place of the configuration that should be loaded.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.12 tsi_status_t NT_TSI_DRV_Measure (uint32_t *instance*)

The function is non-blocking. Therefore, the results can be obtained after the driver completes the measure cycle. The end of the measure cycle can be checked by pooling the NT_TSI_DRV_GetStatus function or waiting for the registered callback function by using NT_TSI_DRV_SetCallBackFunc or NT_TSI_DRV_Init.

```
// Example of the pooling style of the use of the NT_TSI_DRV_Measure() function
if(NT_TSI_DRV_Measure(0) != kStatus_TSI_Success)
{
    // Error, the TSI 5'th electrode is not enabled
}

while(NT_TSI_DRV_GetStatus(0) != kStatus_TSI_Initialized)
{
    // Do something useful - don't waste the CPU cycle time
}
```

Parameters

instance	The TSI module instance.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.13 tsi_status_t NT_TSI_DRV_MeasureAllElect (uint32_t *instance*, uint16_t(∗) *counters[]*)

This function executes one measurement of all used electrodes. The results (counter values) are stored in te array pointed to by the pointer in the function parameters.

```
 uint16_t curr_counters[TF_TSI_TOTAL_CHANNEL_COUNT];

// Measure all electrodes at one time
if(NT_TSI_DRV_MeasureAllElect(0, &curr_counters) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot measure the electrodes in this mode
}
```

Parameters

instance	The TSI module instance.
counters	The pointer to the array where all used TSI counters are stored.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.14 tsi_status_t NT_TSI_DRV_MeasureMoreElect (uint32_t *instance*, uint16_t(∗) *counters[]*, struct nt_electrode ∗∗ *electrodes*, uint8_t *elNumber*)

This function executes one measurement of more selected electrodes by pointer array. The results (counter values) are stored in te array pointed to by the pointer in the function parameters.

```
 uint16_t curr_counters[TF_TSI_TOTAL_CHANNEL_COUNT];
 struct nt_kernel * system = _nt_system_get();
 struct nt_electrode **electrodes = (struct nt_electrode **)system->
      rom->modules[0]->electrodes;

// Measure all electrodes at one time
if(result = NT_TSI_DRV_MeasureMoreElect(0, &curr_counters, electrodes, 1);) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot measure the electrodes in this mode
}
```

Parameters

instance	The TSI module instance.
counters	The pointer to the array where all used TSI counters are stored.
electrodes	The pointer array to enabled electrodes to be measured.
elNumber	The number of enabled electrodes to be measured.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.15 uint16_t NT_TSI_DRV_MeasureOneElect (uint32_t *instance*, struct nt_electrode * *electrode*)

This function executes one measurement of the specified electrode (self or mutual capacitance) The electrode to be measured is specified by the pointer in the second function parameter. The result (counter value) is returned by this function.

```
uint32_t result;

// Measure one electrode at one time
result= NT_TSI_DRV_MeasureOneElect(0, &El_1);
```

Parameters

instance	The TSI module instance.
electrode	The pointer to electrode to be measured.

Returns

The measured electrode counter value.

8.12.7.2.16 tsi_status_t NT_TSI_DRV_Recalibrate (uint32_t *instance,* void ∗ *configuration*)

The function sets the TSI register to the default configuration and adjusts the configuration to reach the selected counter and sensitivity ranges (defined by user). Firstly, the self-capacitance TSI channels are adjusted, and then the mutual-capacitance TSI channels are adjusted. When any channel type is adjusted, the function firstly measures all TSI channels (self or mutual) and calculates the average counter value. If the average counter value is outside the selected range, the function adjusts the TSI parameter to get inside the range. The function repeats the previous steps until the average counter value is within the selected range. If all combinations are reached, the function returns an error. Then the function calibrates the sensitivity in a similar way, but the user must touch the self-cap or mutual-cap electrodes before each calibration cycle. The function makes one or more touch events by gpio method, depending on how many tuning cycles are required to reach the selected sensitivity ranges.

Warning

In some cases, the function does not finish the calibration (returns an error).

```
// define auto-calibration constant in nt_setup.c file
tsi_status_t recalib_status
struct nt_module tsi_module
struct nt_tsi_recalib_config recalib_config = {
.SelfRangeMin     = 50000,
.SelfRangeMax     = 60000,
.SelfRangeDeltaMin= 400,
.SelfRangeDeltaMax= 1300,
.MutRangeMin      = 12000,
.MutRangeMax      = 16000,
.MutRangeDeltaMin = 300,
.MutRangeDeltaMax = 600,
};

// assign auto-calibration structure to tsi module in nt_setup.c file
const struct nt_module tsi_module =
{
    ...
    .recalib_config = (void*)&recalib_config,
};

// call the nt_module_recalibrate function
recalib_status = (tsi_status_t) nt_module_recalibrate(&tsi_module);

if (recalib_status != kStatus_TSI_Success)
{
    // Error, TSI calibration  was not successful (recalib_config parameters are too strict)
}
```

Parameters

module	The pointer to the TSI module structure.
configuration	The pointer to the user recalibration configuration structure.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.17 tsi_status_t NT_TSI_DRV_Resume (uint32_t *instance*)

```
NT_TSI_DRV_Resume(0)
```

Parameters

instance	The TSI module instance.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.18 tsi_status_t NT_TSI_DRV_SaveConfiguration (uint32_t *instance,* const nt_tsi_modes_t *mode,* nt_tsi_operation_mode_t * *operationMode*)

This function saves the configuration of a specific mode. This can be used when the calibrated data should be stored in any backup memory to load after the start of the MCU to avoid running the recalibration that takes more time.

```
// Save operation mode configuration

extern nt_tsi_operation_mode_t  myTsiNvmLowPowerConfiguration;

if(NT_TSI_DRV_SaveConfiguration(0,
      tsi_OpModeLowPower, &myTsiNvmLowPowerConfiguration) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot save the configuration.
}
```

Parameters

instance	The TSI module instance.
mode	The requested new operation mode.
operationMode	The pointer to the storage place of the configuration that should be saved.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.19 tsi_status_t NT_TSI_DRV_SetCallBackFunc (uint32_t *instance,* const tsi_callback_t *pFuncCallBack,* void * *usrData*)

This function sets up or clears (parameter pFuncCallBack = NULL) the callback function pointer which is called after each measure cycle ends. The user can also set the custom user data which is handled by the parameter to a callback function. One function can be called by more sources.

```
// Clear the previous callback function

if(NT_TSI_DRV_SetCallBackFunc(0, NULL, NULL) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot set up the callback function at the moment
}

// Set new callback function

if(NT_TSI_DRV_SetCallBackFunc(0, myFunction, (void*)0x12345678) !=
      kStatus_TSI_Success)
{
    // Error, the TSI driver cannot set up the callback function at the moment
}
```

Parameters

instance	The TSI module instance.
pFuncCallBack	The pointer to the application callback function.
usrData	The user data pointer.

Returns

An error code or kStatus_TSI_Success.

8.12.7.2.20 tsi_status_t NT_TSI_DRV_Suspend (uint32_t *instance*)

```
NT_TSI_DRV_Suspend(0)
```

Parameters

instance	The TSI module instance.

Returns

An error code or kStatus_TSI_Success.

Chapter 9
NXP Touch Private API

9.1 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 NXP Touch Private API:

Modules

- Controls
- Electrodes

- Filters
- Key Detectors
- Modules
- safety
- FreeMASTER support
- Memory Management
- System

9.2 Electrodes

9.2.1 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 nt_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 NXP Touch library. Collaboration diagram for Electrodes:

Modules

- API Functions

Data Structures

- union nt_electrode_special_data
- struct nt_electrode_data

Enumerations

- enum nt_electrode_flags {
  NT_ELECTRODE_ENABLED,
  NT_ELECTRODE_LOCK_BASELINE_REQ_FLAG,
  NT_ELECTRODE_LOCK_BASELINE_FLAG,
  NT_ELECTRODE_DIGITAL_RESULT_ONLY_FLAG,
  NT_ELECTRODE_AFTER_INIT_TOUCH_FLAG }

9.2.2 Data Structure Documentation

9.2.2.1 union nt_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 nt_electrode_special_data:

Data Fields

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

9.2.2.2 struct nt_electrode_data

Electrode RAM structure used to store volatile parameters, flags, and other data, to enable a generic behaviour of the Electrode. You must allocate this structure and put a pointer into the nt_electrode structure, when the electrode is being configured and registered in the module and control.

Collaboration diagram for nt_electrode_data:

Data Fields

uint16_t	baseline	Baseline.
uint32_t	flags	Flags.
union nt_keydetector_data	keydetector_data	Pointer to the key detector data structure.
struct nt_module_data *	module_data	Pointer to the owner module data.
uint16_t	raw_signal	Raw data to be handled in the task process.
struct nt_electrode *	rom	Pointer to the electrode user parameters.
struct nt_electrode_data *	shielding_electrode	Pointer to a shielding electrode (if it is used).
uint16_t	signal	Processed signal.
union nt_electrode_special_data	special_data	Pointer to the special data (for example noise mode data for the TSI).

struct nt_electrode_status	status[NT_ELECTRODE_STATUS_HISTORY_COUNT]	Statuses.
uint8_t	status_index	Status index.

9.2.3 Enumeration Type Documentation

9.2.3.1 enum nt_electrode_flags

Electrodes flags which can be set/cleared.

Enumerator

NT_ELECTRODE_ENABLED This flag signals that the electrode is enabled or disabled

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

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

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

NT_ELECTRODE_AFTER_INIT_TOUCH_FLAG This flag signals that the electrode is touched after the enable/init process.

9.2.4 API Functions

9.2.4.1 Overview

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

Functions

- struct nt_electrode_data * _nt_electrode_get_data (const struct nt_electrode *electrode)
  Get electrode data structure pointer.
- int32_t _nt_electrode_get_index_from_module (const struct nt_module *module, const struct nt_-electrode *electrode)
  Get the electrode index in the module electrode array structure pointer.
- struct nt_electrode_data * _nt_electrode_init (struct nt_module_data *module, const struct nt_-electrode *electrode)
  Initialize an electrode object.
- uint32_t _nt_electrode_shielding_process (struct nt_electrode_data *electrode, uint32_t signal)
  Process shielding if it is enabled, otherwise it returns the same value.
- uint32_t _nt_electrode_normalization_process (const struct nt_electrode_data *electrode, uint32_t signal)
  Scale signal.
- void _nt_electrode_set_signal (struct nt_electrode_data *electrode, uint32_t signal)
  Set the signal for the electrode.
- void _nt_electrode_set_raw_signal (struct nt_electrode_data *electrode, uint32_t signal)
  Set the raw signal for the electrode.
- void _nt_electrode_set_status (struct nt_electrode_data *electrode, int32_t state)
  Set the status of the electrode.
- static void _nt_electrode_set_flag (struct nt_electrode_data *electrode, uint32_t flags)
- static void _nt_electrode_clear_flag (struct nt_electrode_data *electrode, uint32_t flags)
- static uint32_t _nt_electrode_get_flag (struct nt_electrode_data *electrode, uint32_t flags)
- uint32_t _nt_electrode_get_time_offset_period (const struct nt_electrode_data *electrode, uint32_t event_period)
  Determine, whether the specified time (or its multiples) has elapsed since the last electrode event.
- int32_t _nt_electrode_get_last_status (const struct nt_electrode_data *electrode)
  Get the last known electrode status.
- uint32_t _nt_electrode_get_time_offset (const struct nt_electrode_data *electrode)
  Get the time since the last electrode event.
- uint32_t _nt_electrode_get_signal (const struct nt_electrode_data *electrode)

Get the normalized and processed electrode signal.
- int32_t _nt_electrode_get_status (const struct nt_electrode_data *electrode, uint32_t index)
  Get the electrode status by specifying the history buffer index.
- uint32_t _nt_electrode_get_last_time_stamp (const struct nt_electrode_data *electrode)
  Get the last known electrode time-stamp.
- uint32_t _nt_electrode_get_time_stamp (const struct nt_electrode_data *electrode, uint32_t index)
  Get the electrode status time-stamp by specifying the history buffer index.
- uint32_t _nt_electrode_get_raw_signal (const struct nt_electrode_data *electrode)
  Get the raw electrode signal.
- int32_t _nt_electrode_get_delta (const struct nt_electrode_data *electrode)
  Return difference between the signal and its baseline.
- uint32_t _nt_electrode_is_touched (const struct nt_electrode_data *electrode)
  Get the state of the electrode.
- struct nt_electrode * _nt_electrode_get_shield (const struct nt_electrode *electrode)
  Get the shielding electrode.

9.2.4.2 Function Documentation

9.2.4.2.1 static void _nt_electrode_clear_flag (struct nt_electrode_data * *electrode,* uint32_t *flags*) [inline], [static]

Here is the caller graph for this function:

9.2.4.2.2 struct nt_electrode_data* _nt_electrode_get_data (const struct nt_electrode * *electrode*)

Parameters

electrode	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.

Here is the call graph for this function:

Here is the caller graph for this function:

9.2.4.2.3 int32_t _nt_electrode_get_delta (const struct nt_electrode_data * *electrode*)

Parameters

electrode	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:

9.2.4.2.4 static uint32_t _nt_electrode_get_flag (struct nt_electrode_data ∗ *electrode,* uint32_t *flags*) [inline], [static]

Here is the caller graph for this function:

9.2.4.2.5 int32_t _nt_electrode_get_index_from_module (const struct nt_module ∗ *module,* const struct nt_electrode ∗ *electrode*)

Parameters

electrode	Pointer to the electrode user parameter structure.
module	Pointer to the module 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:

9.2.4.2.6 int32_t _nt_electrode_get_last_status (const struct nt_electrode_data ∗ *electrode*)

Parameters

electrode	Pointer to the electrode data.

Returns

Current electrode status.

Here is the call graph for this function:

Here is the caller graph for this function:

9.2.4.2.7 uint32_t _nt_electrode_get_last_time_stamp (const struct nt_electrode_data * *electrode*)

Parameters

electrode	Pointer to the electrode data.

Returns

Current electrode status.

Here is the call graph for this function:

Here is the caller graph for this function:

9.2.4.2.8 uint32_t _nt_electrode_get_raw_signal (const struct nt_electrode_data * *electrode*)

Parameters

electrode	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:

9.2.4.2.9 struct nt_electrode∗ _nt_electrode_get_shield (const struct nt_electrode ∗ *electrode*)

Parameters

electrode	Pointer to the electrode.

Returns

Pointer to the shielding electrode, if available.

Here is the caller graph for this function:

9.2.4.2.10 uint32_t _nt_electrode_get_signal (const struct nt_electrode_data ∗ *electrode*)

Parameters

electrode	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:

9.2.4.2.11 int32_t _nt_electrode_get_status (const struct nt_electrode_data * *electrode,* uint32_t *index*)

Parameters

electrode	Pointer to the electrode data.
index	Index of the required status.

Returns

- status within the nt_electrode_state, if the index is within the range
- NT_FAILURE if the index is out of range.

Here is the caller graph for this function:

9.2.4.2.12 uint32_t _nt_electrode_get_time_offset (const struct nt_electrode_data * *electrode*)

Parameters

electrode	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:

9.2.4.2.13 uint32_t _nt_electrode_get_time_offset_period (const struct nt_electrode_data * *electrode,* uint32_t *event_period*)

Parameters

electrode	Pointer to the electrode data.
event_period	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:

9.2.4.2.14 uint32_t _nt_electrode_get_time_stamp (const struct nt_electrode_data * *electrode,* uint32_t *index*)

Parameters

electrode	Pointer to the electrode data.
index	Index of the required status.

Returns

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

Here is the caller graph for this function:

9.2.4.2.15 struct nt_electrode_data∗ _nt_electrode_init (struct nt_module_data ∗ *module,* const struct nt_electrode ∗ *electrode*)

Parameters

module	Pointer to the module data.
electrode	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:

9.2.4.2.16 uint32_t _nt_electrode_is_touched (const struct nt_electrode_data * *electrode*)

Parameters

electrode	Pointer to the electrode data.

Returns

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

Here is the call graph for this function:

Here is the caller graph for this function:

9.2.4.2.17 uint32_t _nt_electrode_normalization_process (const struct nt_electrode_data * *electrode,* uint32_t *signal*)

Parameters

electrode	A pointer to the electrode data.
signal	

Returns

signal

Normalize the signal to working values. Values from nt_electrode divider or multiplier normalize the measured signal.

9.2.4.2.18 static void _nt_electrode_set_flag (struct nt_electrode_data * *electrode,* uint32_t *flags*) [inline], [static]

Here is the caller graph for this function:

9.2.4.2.19 void _nt_electrode_set_raw_signal (struct nt_electrode_data * *electrode,* uint32_t *signal*)

Parameters

electrode	Pointer to the electrode data.
signal	

Returns

none

Here is the caller graph for this function:

9.2.4.2.20 void _nt_electrode_set_signal (struct nt_electrode_data * *electrode,* uint32_t *signal*)

Parameters

electrode	A pointer to the electrode.
signal	

Returns

none

9.2.4.2.21 void _nt_electrode_set_status (struct nt_electrode_data * *electrode,* int32_t *state*)

Parameters

electrode	A pointer to the electrode data.
state	nt_electrode_state

Returns

none

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

Here is the call graph for this function:

Here is the caller graph for this function:

9.2.4.2.22 uint32_t _nt_electrode_shielding_process (struct nt_electrode_data * *electrode,* uint32_t *signal*)

Parameters

electrode	A pointer to the electrode data.
signal	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.

9.3 Filters

9.3.1 Overview

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

Modules

- API Functions

Data Structures

- struct nt_filter_fbutt_data
- struct nt_filter_moving_average_data

Enumerations

- enum nt_filter_state {
  NT_FILTER_STATE_INIT,
  NT_FILTER_STATE_RUN }

9.3.2 Data Structure Documentation

9.3.2.1 struct nt_filter_fbutt_data

The butterworth filter context data.

Collaboration diagram for nt_filter_fbutt_data:

Data Fields

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

9.3.2.2 struct nt_filter_moving_average_data

The moving average filter context data.

Collaboration diagram for nt_filter_moving_average_data:

Data Fields

int32_t	sum	The sum of the filter data

9.3.3 Enumeration Type Documentation

9.3.3.1 enum nt_filter_state

The filter state definition.

Enumerator

NT_FILTER_STATE_INIT The filter is initialized.
NT_FILTER_STATE_RUN The filter is running correctly.

9.3.4 API Functions

9.3.4.1 Overview

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

Functions

- uint32_t _nt_abs_int32 (int32_t lsrc)
  Gets the absolute value.
- void _nt_filter_fbutt_init (const struct nt_filter_fbutt ∗rom, struct nt_filter_fbutt_data ∗ram, uint32_t signal)
  Initialize the ButterWorth filter for the first use.
- uint16_t _nt_filter_fbutt_process (struct nt_filter_fbutt_data ∗ram, uint16_t signal)
  Process signal fbutt filter.
- uint32_t _nt_filter_iir_process (const struct nt_filter_iir ∗rom, uint32_t signal, uint32_t previous_-signal)
  Process signal IIR filter.
- int32_t _nt_filter_moving_average_init (const struct nt_filter_moving_average ∗rom, struct nt_-filter_moving_average_data ∗ram, uint16_t value)
  This function initialize moving average filter.
- uint32_t _nt_filter_moving_average_process (const struct nt_filter_moving_average ∗rom, struct nt-_filter_moving_average_data ∗ram, uint16_t value)
  This function compute moving average filter.
- uint16_t _nt_filter_abs (int16_t value)
  This function compute absolute value (16-bit version).
- uint16_t _nt_filter_pos (int16_t value)
  This function returns pos value (16-bit version).
- uint16_t _nt_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).
- int32_t _nt_filter_range_s (int32_t value, uint32_t limit)
  This function limit the input value in allowed range (16-bit version).
- uint16_t _nt_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).
- uint16_t _nt_filter_deadrange_p (uint16_t value, uint16_t base, uint16_t range)
  This function make positive dead range for input value out of the allowed range (16-bit version).
- int32_t _nt_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).

9.3.4.2 Function Documentation

9.3.4.2.1 uint32_t _nt_abs_int32 (int32_t *lsrc*)

Parameters

lsrc	Input signed 32-bit number.

Returns

Unsigned 32-bit absolute value of the input number.

9.3.4.2.2 uint16_t _nt_filter_abs (int16_t *value*)

Parameters

value	Input signed value.

Returns

Absolute unsigned value of input.

9.3.4.2.3 uint16_t _nt_filter_deadrange_p (uint16_t *value,* uint16_t *base,* uint16_t *range*)

Parameters

value	Input value.
base	Base value of deadband range.
range	Range of the deadband range.

Returns

Result value out of the deadband range.

Here is the call graph for this function:

9.3.4.2.4 uint16_t _nt_filter_deadrange_u (uint16_t *value,* uint16_t *base,* uint16_t *range*)

Parameters

value	Input value.
base	Base value of deadband range.
range	Range of the deadband range (one half).

Returns

Result value out of the deadband range.

Here is the call graph for this function:

9.3.4.2.5 void _nt_filter_fbutt_init (const struct nt_filter_fbutt ∗ *rom,* struct nt_filter_fbutt_data ∗ *ram,* uint32_t *signal*)

Parameters

rom	Pointer to the nt_filter_fbutt structure.
ram	Pointer to the nt_filter_fbutt_data.
signal	Input signal.

Returns

none

Here is the call graph for this function:

9.3.4.2.6 uint16_t _nt_filter_fbutt_process (struct nt_filter_fbutt_data ∗ *ram,* uint16_t *signal*)

Parameters

ram	Pointer to the nt_filter_fbutt_data structure.
signal	Input signal.

Returns

Filtered signal.

Returns signal equal

9.3.4.2.7 uint32_t _nt_filter_iir_process (const struct nt_filter_iir ∗ *rom,* uint32_t *signal,* uint32_t *previous_signal*)

Parameters

rom	Pointer to nt_filter_iir
signal	Current signal.
previous_-signal	Previous signal

Returns

signal

Process the signal, using the following equation: y(n)=(1 / (coef + 1) ∗ current signal + (coef / (coef + 1) ∗ previous_signal)

9.3.4.2.8 int32_t _nt_filter_is_deadrange_u (uint16_t *value,* uint16_t *base,* uint16_t *range*)

Parameters

value	Input value.
base	Base value of deadband range.
range	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:

9.3.4.2.9 uint16_t _nt_filter_limit_u (int32_t *value,* uint16_t *limit_l,* uint16_t *limit_h*)

Parameters

value	Input value.
limit_l	Limitation low range border.
limit_h	Limitation high range border.

Returns

Result value.

Here is the caller graph for this function:

9.3.4.2.10 int32_t _nt_filter_moving_average_init (const struct nt_filter_moving_average * *rom*, struct nt_filter_moving_average_data * *ram*, uint16_t *value*)

Parameters

rom	Pointer to nt_filter_moving_average structure.
ram	Pointer to nt_filter_moving_average_data structure.
value	Input initial value.

Returns

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

9.3.4.2.11 uint32_t _nt_filter_moving_average_process (const struct nt_filter_moving_average * *rom*, struct nt_filter_moving_average_data * *ram*, uint16_t *value*)

Parameters

rom	Pointer to nt_filter_moving_average structure.

ram	Pointer to nt_filter_moving_average_data structure.
value	Input new value.

Returns

Current value of the moving average filter.

9.3.4.2.12 uint16_t _nt_filter_pos (int16_t *value*)

Parameters

value	Input signed value.

Returns

value of input,if positive, else returns 0

9.3.4.2.13 int32_t _nt_filter_range_s (int32_t *value,* uint32_t *limit*)

Parameters

value	Input value.
limit	Limitation range.

Returns

Result value.

9.3.5 Advanced Filtering and Integrating Detection

9.3.5.1 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 nt_keydetector_afid_asc_data
- struct nt_keydetector_afid_data

Macros

- #define NT_KEYDETECTOR_AFID_INITIAL_INTEGRATOR_VALUE
- #define NT_KEYDETECTOR_AFID_INITIAL_RESET_TOUCH_COUNTER_VALUE
- #define NT_KEYDETECTOR_AFID_INITIAL_RESET_RELEASE_COUNTER_VALUE

9.3.5.2 Data Structure Documentation

9.3.5.2.1 struct nt_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 nt_keydetector_afid_asc_data:

Data Fields

int32_t	max_resets	

9.3.5.2.2 struct nt_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.

Collaboration diagram for nt_keydetector_afid_data:

Data Fields

struct nt_keydetector-_afid_asc_data	asc	Storage of ASC (Automatic sensitive calibration) data for AFID
struct nt_filter_-moving_-average_data	base_avrg	Base line moving average filter data.
struct nt_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 nt_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.

9.3.5.3 Macro Definition Documentation

9.3.5.3.1 #define NT_KEYDETECTOR_AFID_INITIAL_INTEGRATOR_VALUE

The initial integration value of the AFID.

9.3.5.3.2 #define NT_KEYDETECTOR_AFID_INITIAL_RESET_RELEASE_COUNTER_VALUE

The reset threshold value of the AFID.

9.3.5.3.3 #define NT_KEYDETECTOR_AFID_INITIAL_RESET_TOUCH_COUNTER_VALUE

The initial reset counter value of the AFID.

9.4 Key Detectors

9.4.1 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
- Safa Detector
- uSafa Detector

Data Structures

- union nt_keydetector_data

9.4.2 Data Structure Documentation

9.4.2.1 union nt_keydetector_data

The key detector optional run-time data.

Collaboration diagram for nt_keydetector_data:

Data Fields

struct nt_keydetector-_afid_data *	afid	AFID electrode run-time data
struct nt_keydetector-_safa_data *	safa	SAFA electrode run-time data
struct nt_keydetector-_usafa_data *	usafa	USAFA electrode run-time data

9.4.3 Safa Detector

9.4.3.1 Overview

Safa key detector is a method for recognition of touch or release states. It can be used for each type of control.

If the measured sample is reported as a valid sample, the module calculates the delta value from the actual signal value and the baseline value. The delta value is compared to the threshold value computed from the expected signal and baseline values. Based on the result, it determines the electrode state, which can be released, touched, changing from released to touched, and changing from touched to released. The method is using moving average filters to determine the baseline and expected signal values with a different depth of the filter, depending on the state of the electrode. The deadband filters in the horizontal and vertical directions are also implemented. Collaboration diagram for Safa Detector:

Data Structures

- struct nt_keydetector_safa_data

9.4.3.2 Data Structure Documentation

9.4.3.2.1 struct nt_keydetector_safa_data

Safa RAM structure. This structure is used for internal algorithms to store data, while evaluating Safa. Contains the data of result calculation and auxiliary variables.

This structure only manages and uses internal methods.

Collaboration diagram for nt_keydetector_safa_data:

Data Fields

struct nt_filter_-moving_-average_data	base_avrg	Baseline moving average filter data.
struct nt_filter-_moving_-average	base_avrg_init	Baseline moving average filter settings.

int32_t	deadband_cnt	Deadband event counter.
uint32_t	deadband_h	Deadband high watermark
uint32_t	deadband_l	Deadband low watermark
int32_t	entry_event_-cnt	Event counter value.
uint32_t	f_noise	Fast Noise value.
struct nt_filter_-moving_-average_data	f_noise_avrg	Fast Noise signal moving average filter data.
struct nt_filter-_moving_-average	f_noise_avrg_-init	Fast Noise moving average filter settings.
enum nt_filter_state	filter_state	Input filter state.
uint32_t	noise	Noise value.
struct nt_filter_-moving_-average_data	noise_avrg	Noise signal moving average filter data.
struct nt_filter-_moving_-average	noise_avrg_init	Noise moving average filter settings.
uint32_t	predicted_-signal	Predicted signal value.
struct nt_filter_-moving_-average_data	predicted_-signal_avrg	Predicted signal line moving average filter data.
struct nt_filter-_moving_-average	predicted_-signal_avrg_-init	Predicted signal moving average filter settings.

int32_t	recovery_cnt	Recovery counter.

9.4.4 uSafa Detector

9.4.4.1 Overview

The uSafa key detector is a method for recognition of the touch or release states. It can be used for each type of control.

If the measured sample is reported as a valid sample, the module calculates the delta value from the actual signal value and the baseline value. The delta value is compared to the threshold value computed from the expected signal and baseline values. Based on the result, it determines the electrode state, which can be released, touched, changing from released to touched, and changing from touched to released. The method uses moving average filters to determine the baseline and expected signal values with a different depth of the filter, depending on the state of the electrode. The deadband filters in the horizontal and vertical directions are also implemented. Collaboration diagram for uSafa Detector:

Data Structures

- struct nt_keydetector_usafa_data

9.4.4.2 Data Structure Documentation

9.4.4.2.1 struct nt_keydetector_usafa_data

uSafa RAM structure. This structure is used for internal algorithms to store data, while evaluating uSafa. Contains the data of the resulting calculation and auxiliary variables.

This structure manages and uses internal methods only.

Collaboration diagram for nt_keydetector_usafa_data:

Data Fields

struct nt_filter_-moving_-average_data	base_avrg	Baseline moving average filter data.
struct nt_filter-_moving_-average	base_avrg_init	Baseline moving average filter settings.

int32_t	dc_track_cnt	DC tracker counter
int32_t	deadband_cnt	Deadband event counter.
uint32_t	deadband_h	Deadband high watermark
uint32_t	deadband_l	Deadband low watermark
int32_t	entry_event_-cnt	Event counter value.
uint32_t	f_noise	Fast Noise value.
struct nt_filter_-moving_-average_data	f_noise_avrg	Fast Noise signal moving average filter data.
struct nt_filter-_moving_-average	f_noise_avrg_-init	Fast Noise moving average filter settings.
enum nt_filter_state	filter_state	Input filter state.
uint32_t	noise	Noise value.
struct nt_filter_-moving_-average_data	noise_avrg	Noise signal moving average filter data.
struct nt_filter-_moving_-average	noise_avrg_init	Noise moving average filter settings.
uint32_t	predicted_-signal	Predicted signal value.
struct nt_filter_-moving_-average_data	predicted_-signal_avrg	Predicted signal line moving average filter data.
struct nt_filter-_moving_-average	predicted_-signal_avrg_-init	Predicted signal moving average filter settings.

int32_t	recovery_cnt	Recovery counter.

9.4.5 CS module

9.4.5.1 Overview

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

The CS module is designed for processors that have a hardware CS module. Collaboration diagram for CS module:

Data Structures

- struct nt_module_cs_data

9.4.5.2 Data Structure Documentation

9.4.5.2.1 struct nt_module_cs_data

Collaboration diagram for nt_module_cs_data:

Data Fields

nt_cs_state_t	cs_state	main NT driver data structure with state variables

9.4.6 GPIO module

9.4.6.1 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 nt_module_gpio_data

9.4.6.2 Data Structure Documentation

9.4.6.2.1 struct nt_module_gpio_data

GPIO module's RAM. This structure contains

Collaboration diagram for nt_module_gpio_data:

Data Fields

uint32_t	pen	PEN - enablement of all modules electrodes

9.4.7 GPIO interrupt module

9.4.7.1 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 nt_module_gpioint_data

9.4.7.2 Data Structure Documentation

9.4.7.2.1 struct nt_module_gpioint_data

GPIO interrupt module's RAM. This structure contains

Collaboration diagram for nt_module_gpioint_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 nt_gpio_scan_states

9.4.8 TSI module

9.4.8.1 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 nt_module_tsi_noise_data
- struct nt_module_tsi_data

Macros

- #define NT_TSI_NOISE_INITIAL_TOUCH_THRESHOLD
- #define NT_TSI_NOISE_TOUCH_RANGE

Enumerations

- enum nt_module_tsi_flags {
  NT_MODULE_IN_NOISE_MODE_FLAG,
  NT_MODULE_HAS_NOISE_MODE_FLAG,
  NT_MODULE_NOISE_MODE_REQ_FLAG }

9.4.8.2 Data Structure Documentation

9.4.8.2.1 struct nt_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 nt_module_tsi_noise_data:

Data Fields

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

9.4.8.2.2 struct nt_module_tsi_data

Collaboration diagram for nt_module_tsi_data:

Data Fields

uint32_t	noise_-timestamp	Noise mode switch event time-stamp
nt_tsi_state_t	tsi_state	main NT driver data structure with state variables

9.4.8.3 Macro Definition Documentation

9.4.8.3.1 #define NT_TSI_NOISE_INITIAL_TOUCH_THRESHOLD

The TSI module noise mode initial touch threshold value.

9.4.8.3.2 #define NT_TSI_NOISE_TOUCH_RANGE

The TSI module noise mode touch range value.

9.4.8.4 Enumeration Type Documentation

9.4.8.4.1 enum nt_module_tsi_flags

The TSI module's noise mode flags definition.

Enumerator

NT_MODULE_IN_NOISE_MODE_FLAG This flag signalizes that the module is currently in the noise mode.

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

NT_MODULE_NOISE_MODE_REQ_FLAG This flag signalizes that the module wants to switch to the noise mode.

9.5 Modules

9.5.1 Overview

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

Each Module implements a set of private functions contained in the nt_modules_prv.h file. Collaboration diagram for Modules:

Modules

- CS module
- GPIO module
- GPIO interrupt module
- TSI module
- General API

9.5.2 General API

9.5.2.1 Overview

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

Modules

- API functions

Data Structures

- union nt_module_special_data
- struct nt_module_data
- struct nt_module_interface

9.5.2.2 Data Structure Documentation

9.5.2.2.1 union nt_module_special_data

The module optional run-time data.

Collaboration diagram for nt_module_special_data:

Data Fields

struct nt_module_cs-_data *	cs	CS module run-time data

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

9.5.2.2.2 struct nt_module_data

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

Collaboration diagram for nt_module_data:

Data Fields

enum nt_-module_mode	active_mode	Active mode of the module.
uint8_t	electrode_last	Last assigned electrode to be measured.
struct nt_-electrode_data **	electrodes	Pointer to the list of electrodes. Can't be NULL.
uint8_t	electrodes_cnt	Electrode's count.
uint32_t	flags	Module's symptoms.
struct nt_module *	rom	Pointer to the module parameters defined by the user.
uint32_t	safety_delay_-last_time	System time of last safety delay function execution
uint8_t	safety_last_-adj_test	Numer of the last executed adj safety test
uint8_t	safety_last_test	Numer of the last executed safety test
uint32_t	safety_last_-time	System time of last safety process function execution
union nt_module_-special_data	special_data	Pointer to the special data (for example run-time data for the GPI-O).

9.5.2.2.3 struct nt_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 nt_module_interface:

Data Fields

- int32_t(∗ init)(struct nt_module_data ∗module)
- int32_t(∗ trigger)(struct nt_module_data ∗module)
- int32_t(∗ process)(struct nt_module_data ∗module)
- int32_t(∗ recalibrate)(struct nt_module_data ∗module, void ∗configuration)
- int32_t(∗ electrode_enable)(struct nt_module_data ∗module, const uint32_t elec_index)
- int32_t(∗ electrode_disable)(struct nt_module_data ∗module, const uint32_t elec_index)
- int32_t(∗ change_mode)(struct nt_module_data ∗module, const enum nt_module_mode mode, const struct nt_electrode ∗electrode)
- int32_t(∗ load_configuration)(struct nt_module_data ∗module, const enum nt_module_mode mode, const void ∗config)
- int32_t(∗ save_configuration)(struct nt_module_data ∗module, const enum nt_module_mode mode, void ∗config)
- const char ∗ name
- const uint32_t params_size

9.5.2.2.3.1 Field Documentation

9.5.2.2.3.1.1 int32_t(∗ nt_module_interface::change_mode)(struct nt_module_data ∗module, const enum nt_module_mode mode, const struct nt_electrode ∗electrode)

Change the the mode of the module.

9.5.2.2.3.1.2 int32_t(∗ nt_module_interface::electrode_disable)(struct nt_module_data ∗module, const uint32_t elec_index)

Disable the module electrode in hardware.

9.5.2.2.3.1.3 int32_t(∗ nt_module_interface::electrode_enable)(struct nt_module_data ∗module, const uint32_t elec_index)

Enable the module electrode in hardware.

9.5.2.2.3.1.4 int32_t(∗ nt_module_interface::init)(struct nt_module_data ∗module)

The initialization of the module.

9.5.2.2.3.1.5 int32_t(∗ nt_module_interface::load_configuration)(struct nt_module_data ∗module, const enum nt_module_mode mode, const void ∗config)

Load the configuration for the selected mode.

9.5.2.2.3.1.6 const char∗ nt_module_interface::name

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

9.5.2.2.3.1.7 const uint32_t nt_module_interface::params_size

Structure size

9.5.2.2.3.1.8 int32_t(∗ nt_module_interface::process)(struct nt_module_data ∗module)

Process the read data from the trigger event.

9.5.2.2.3.1.9 int32_t(∗ nt_module_interface::recalibrate)(struct nt_module_data ∗module, void ∗configuration)

Force recalibration of the module in the current mode.

9.5.2.2.3.1.10 int32_t(∗ nt_module_interface::save_configuration)(struct nt_module_data ∗module, const enum nt_module_mode mode, void ∗config)

Save the configuration of the selected mode.

9.5.2.2.3.1.11 int32_t(∗ nt_module_interface::trigger)(struct nt_module_data ∗module)

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

9.5.2.3 API functions

9.5.2.3.1 Overview

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

Functions

- struct nt_module_data * _nt_module_get_data (const struct nt_module *module)
  Get the module data structure pointer.
- struct nt_module_data * _nt_module_init (const struct nt_module *module)
  Init module.
- int32_t _nt_module_trigger (struct nt_module_data *module)
  Trigger the start of measure event of the module.
- int32_t _nt_module_process (struct nt_module_data *module)
  Process the module.
- int32_t _nt_module_process_safety (struct nt_module_data *module)
  Process module Safety tests.
- uint32_t _nt_module_get_electrodes_state (struct nt_module_data *module)
  Get the module electrodes state.
- int32_t _nt_module_check_signal_levels (struct nt_module_data *module)
  Check the module electrodes signal levels.
- uint32_t _nt_module_count_electrodes (const struct nt_module *module)
- static void _nt_module_set_flag (struct nt_module_data *module, uint32_t flags)
  Set the flag of the module.
- static void _nt_module_clear_flag (struct nt_module_data *module, uint32_t flags)
  Reset the flag of the module.
- static uint32_t _nt_module_get_flag (struct nt_module_data *module, uint32_t flags)
  Return the flag of the module.
- static uint32_t _nt_module_get_instance (const struct nt_module_data *module)
  Return the instance of the module.
- static void _nt_module_set_mode (struct nt_module_data *module, uint32_t mode)
  Set the module's mode.
- static uint32_t _nt_module_get_mode (struct nt_module_data *module)
  Get the module's mode.

9.5.2.3.2 Function Documentation

9.5.2.3.2.1 int32_t _nt_module_check_signal_levels (struct nt_module_data * *module*)

Parameters

module	Pointer to the NT module_data.

Returns

mode.

Here is the call graph for this function:

9.5.2.3.2.2 static void _nt_module_clear_flag (struct nt_module_data * *module*, uint32_t *flags*) [inline], [static]

Parameters

module	Pointer to the NT module.
flags	The flags to be cleared.

Returns

void

Here is the caller graph for this function:

9.5.2.3.2.3 uint32_t _nt_module_count_electrodes (const struct nt_module * *module*)

Here is the caller graph for this function:

9.5.2.3.2.4 struct nt_module_data* _nt_module_get_data (const struct nt_module * *module*)

Parameters

module	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:

9.5.2.3.2.5 uint32_t _nt_module_get_electrodes_state (struct nt_module_data * *module*)

Parameters

module	Pointer to the NT module_data.

Returns

mode.

Here is the call graph for this function:

9.5.2.3.2.6 static uint32_t _nt_module_get_flag (struct nt_module_data * *module,* uint32_t *flags*) `[inline]`, `[static]`

Parameters

module	Pointer to the NT module.
flags	The flags to be tested

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:

9.5.2.3.2.7 static uint32_t _nt_module_get_instance (const struct nt_module_data ∗ *module*) [inline], [static]

Parameters

module	Pointer to the NT module.

Returns

instance

9.5.2.3.2.8 static uint32_t _nt_module_get_mode (struct nt_module_data ∗ *module*) [inline], [static]

Parameters

module	Pointer to the NT module_data.

Returns

mode.

9.5.2.3.2.9 struct nt_module_data∗ _nt_module_init (const struct nt_module ∗ *module*)

Parameters

module	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:

9.5.2.3.2.10 int32_t _nt_module_process (struct nt_module_data * *module*)

Parameters

module	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:

9.5.2.3.2.11 int32_t _nt_module_process_safety (struct nt_module_data * *module*)

Parameters

module	Pointer to the module under test.

Returns

The result of the operation.

Here is the caller graph for this function:

9.5.2.3.2.12 static void _nt_module_set_flag (struct nt_module_data * *module,* uint32_t *flags*) [inline], [static]

Parameters

module	Pointer to the NT module.
flags	The flags to be set.

Returns

void

Here is the caller graph for this function:

9.5.2.3.2.13 static void _nt_module_set_mode (struct nt_module_data * *module,* uint32_t *mode*) [inline], [static]

Parameters

module	Pointer to the NT module.
mode	

Returns

None.

Here is the caller graph for this function:

9.5.2.3.2.14 int32_t _nt_module_trigger (struct nt_module_data ∗ *module*)

Parameters

module	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:

Chapter 10
General API

10.1 Overview

General Function definition of the safety_crc.

Functions

- int32_t nt_safety_calc_cfg_crc (const struct nt_system ∗system, uint16_t ∗crc_val)
  Calculate CRC checksum on the System configuration data.
- int32_t nt_safety_check_cfg_data (const struct nt_system ∗system, uint16_t crc_val)
  Check the System configuration data.

10.2 Function Documentation

10.2.1 int32_t nt_safety_calc_cfg_crc (const struct nt_system ∗ *system,* uint16_t ∗ *crc_val*)

Returns

the status of the calculation
- NT_SUCCESS if the calculation was properly done
- NT_FAILURE if the calculation cannot be finished

Calculate the CRC on the System configuration data and returns the pointer to the CRC checksum

```
// Calc CRC on system configuration data
int32_t testResult;
testResult = nt_safety_calc_cfg_crc(const struct nt_system *system, uint16_t
       *crc_val);
```

Here is the call graph for this function:

Here is the caller graph for this function:

10.2.2 int32_t nt_safety_check_cfg_data (const struct nt_system * *system*, uint16_t *crc_val*)

Returns

the status of the operation

- NT_SUCCESS if the check was properly done
- NT_FAILURE if the check cannot be finished Check the system data .

```
// Check the system configuration data, calculate CRC and compare with the  pre-calculated CRC
int32_t testResult;
testResult = nt_safety_check_cfg_data(const struct
      nt_system *system, uint16_t crc_val);
```

Here is the call graph for this function:

10.3 SafetyGPIO

10.3.1 Overview

SafetyGPIO represents the functions to support GPIO or timer periferal configuration. Collaboration diagram for SafetyGPIO:

Functions

- void set_pin_ouput (uint32_t port, uint32_t pin)
  Configure pin as gpio output.
- void set_pin_input (uint32_t port, uint32_t pin)
  Configure pin as gpio input.
- void set_pin_low (uint32_t port, uint32_t pin)
  Configure pin value as logic 0.
- void set_pin_high (uint32_t port, uint32_t pin)
  Configure pin value as logic 1.
- uint32_t get_pin_value (uint32_t port, uint32_t pin)
  Reads the pin value.
- void set_pin_default_state (uint32_t port, uint32_t pin)
  Configure pin as gpio output and set output to logic 1.
- void init_timer (void)
  Initialize the FlexTimer.
- void start_timer (void)
  Start the FlexTimer.
- void stop_timer (void)
  Stop the FlexTimer.
- void timer_reset_counter (void)
  Reset the FlexTimer.
- uint32_t timer_get_counter (void)
  Reads FlexTimer counted value.
- uint32_t timer_get_overrun (void)
  Reads FlexTimer overflow.
- void set_pin_pull_up (uint32_t port, uint32_t pin)
  Configures pin to have pull up resistor.
- void set_pin_pull_down (uint32_t port, uint32_t pin)
  Configures pin to have pull down resistor.
- void dis_pin_pull (uint32_t port, uint32_t pin)
  Disable pin pull resistor.

- void set_pin_tsi_mode (uint32_t port, uint32_t pin)
  Configure pin as TSI peripheral pin.
- void set_pin_gpio_mode (uint32_t port, uint32_t pin)
  Configure pin as GPIO pin.
- void configure_gpio_touch_sensing_pins (uint32_t instance)
  Configure pin as gpio output.

10.3.2 Function Documentation

10.3.2.1 void configure_gpio_touch_sensing_pins (uint32_t *instance*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as gpio input.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as gpio output
uint32_t port = 2;
uint32_t pin = 5;
testResult = set_pin_ouput(port, pin);
```

10.3.2.2 void dis_pin_pull (uint32_t *port,* uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function disable selected pin pull resistor.

```
#include "gpio.h"
// Disable pin 5 of the port 2 (PTC) pull resistor.
uint32_t port = 2;
uint32_t pin = 5;
dis_pin_pull(port, pin);
```

10.3.2.3 uint32_t get_pin_value (uint32_t *port,* uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

pin value in uint32_t data type

Function reads selected pin value of the selected port.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as gpio output
uint32_t port = 2;
uint32_t pin = 5;
uint32_t testResult;
testResult = get_pin_value(port, pin);
```

10.3.2.4 void init_timer (void)

Returns

void

Function Initialize the FlexTimer to be in free-running mode.

```
#include "gpio.h"
// Initialize the FlexTimer.
init_timer();
```

10.3.2.5 void set_pin_default_state (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as gpio output and set output to logic 1.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as gpio output
uint32_t port = 2;
uint32_t pin = 5;
set_pin_default_state(port, pin);
```

Here is the call graph for this function:

10.3.2.6 void set_pin_gpio_mode (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as GPIO pin.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as gpio output
uint32_t port = 2;
uint32_t pin = 5;
set_pin_gpio_mode(port, pin);
```

Here is the caller graph for this function:

10.3.2.7 void set_pin_high (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as logic 1.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as logic 1
uint32_t port = 2;
uint32_t pin = 5;
set_pin_high(port, pin);
```

Here is the caller graph for this function:

10.3.2.8 void set_pin_input (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as gpio input.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as gpio input
uint32_t port = 2;
uint32_t pin = 5;
set_pin_input(port, pin);
```

10.3.2.9 void set_pin_low (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as logic 0.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as logic 0
uint32_t port = 2;
uint32_t pin = 5;
set_pin_low(port, pin);
```

10.3.2.10 void set_pin_ouput (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as gpio output.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as gpio output
uint32_t port = 2;
uint32_t pin = 5;
set_pin_ouput(port, pin);
```

Here is the caller graph for this function:

10.3.2.11 void set_pin_pull_down (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configure selected pin to have pull down resistor.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) to have pull down resistor.
uint32_t port = 2;
uint32_t pin = 5;
set_pin_pull_down(port, pin);
```

10.3.2.12 void set_pin_pull_up (uint32_t *port,* uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configure selected pin to have pull up resistor.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) to have pull up resistor.
uint32_t port = 2;
uint32_t pin = 5;
set_pin_pull_up(port, pin);
```

10.3.2.13 void set_pin_tsi_mode (uint32_t *port*, uint32_t *pin*)

Parameters

port	GPIO port where pin is located.
pin	GPIO pin.

Returns

void

Function configures selected pin of the selected port as TSI peripheral pin.

```
#include "gpio.h"
// Set pin 5 of the port 2 (PTC) as gpio output
uint32_t port = 2;
uint32_t pin = 5;
set_pin_tsi_mode(port, pin);
```

Here is the caller graph for this function:

10.3.2.14 void start_timer (void)

Returns

void

Function Start the FlexTimer to count.

```
#include "gpio.h"
// Start the FlexTimer.
start_timer();
```

10.3.2.15 void stop_timer (void)

Returns

void

Function Stop the FlexTimer to count.

```
#include "gpio.h"
// Stop the FlexTimer.
stop_timer();
```

10.3.2.16 uint32_t timer_get_counter (void)

Returns

uint32_t FlexTimer counter value

Function reads FlexTimer counted value.

```
#include "gpio.h"
// Reads FlexTimer counted value.
uint32_t testResult;
testResult = timer_get_counter();
```

10.3.2.17 uint32_t timer_get_overrun (void)

Returns

1 if there was FlexTimer overrun, 0 if not

Function check and return TOF flag.

```
#include "gpio.h"
// Get FlexTimer overrun state.
uint32_t testResult;
testResult = timer_get_overrun();
```

10.3.2.18 void timer_reset_counter (void)

Returns

void

Function resets the FlexTimer counter.

```
#include "gpio.h"
// Reset the FlexTimer.
timer_reset_counter();
```

10.4 safety

10.4.1 Overview

Safety represent the safety check layer in the NXP Touch system, it is the layer that is tightly coupled to hardware available on the NXP MCU device.

Each safety implements a set of private functions contained in the nt_safety_prv.h file. Collaboration diagram for safety:

Modules

- General API

10.4.2 General API

10.4.2.1 Overview

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

Data Structures

- struct nt_module_safety_interface
- struct nt_safety_data

Functions

- int32_t _nt_module_safety_init (struct nt_module_data ∗module)
  Init the safety functions for the module.
- int32_t _nt_module_safety_process (struct nt_module_data ∗module)
  Process the safety on the module.

10.4.2.2 Data Structure Documentation

10.4.2.2.1 struct nt_module_safety_interface

safety interface structure; safety uses this structure to register the entry points to its algorithms. This approach enables a kind-of polymorphism in the touch System. Safety functions are processed the same way from the System layer, regardless of the specific implementation.

Collaboration diagram for nt_module_safety_interface:

Data Fields

- int32_t(∗ init)(struct nt_module_data ∗module)
- int32_t(∗ process)(struct nt_module_data ∗module)
- const char ∗ name
- const uint32_t params_size

10.4.2.2.1.1 Field Documentation

10.4.2.2.1.1.1 int32_t(∗ nt_module_safety_interface::init)(struct nt_module_data ∗module)

The initialization of the safety functions

10.4.2.2.1.1.2 const char∗ nt_module_safety_interface::name

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

10.4.2.2.1.1.3 const uint32_t nt_module_safety_interface::params_size

Structure size

10.4.2.2.1.1.4 int32_t(∗ nt_module_safety_interface::process)(struct nt_module_data ∗module)

Process the safety functions

10.4.2.2.2 struct nt_safety_data

The matrix RAM structure is used to store volatile parameters of the safety interface.

Allocate this structure and put a pointer into the nt_safety_data structure when it is being registered in the system.

Collaboration diagram for nt_safety_data:

Data Fields

uint32_t	safety_delay_-next_time	Time instant for the delay interval end

10.4.2.3 Function Documentation

10.4.2.3.1 int32_t _nt_module_safety_init (struct nt_module_data ∗ *module*)

Parameters

module	Pointer to the where the module safety to be initialized.

Returns

The result of the operation.

Here is the call graph for this function:

10.4.2.3.2 int32_t _nt_module_safety_process (struct nt_module_data * *module*)

Parameters

module	Pointer to the where the module safety to be processed.

Returns

The result of the operation.

Here is the call graph for this function:

10.5 FreeMASTER support

10.5.1 Overview

Collaboration diagram for FreeMASTER support:

Modules

- API functions

10.5.2 API functions

10.5.2.1 Overview

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

Functions

- int32_t _nt_freemaster_init (void)
  Initialized the NXP touch FreeMASTER support system.
- int32_t _nt_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.

10.5.2.2 Function Documentation

10.5.2.2.1 int32_t _nt_freemaster_add_variable (const char ∗ *name,* const char ∗ *type_name,* void ∗ *address,* uint32_t *size*)

Parameters

name	- pointer to the string with the name of the variable
type_name	- pointer to the string with the name of the variable type
address	- address of the variable
size	- size of the variable

Returns

The result of the operation.

Here is the caller graph for this function:

10.5.2.2.2 int32_t _nt_freemaster_init (void)

Returns

The result of operation.

Here is the caller graph for this function:

10.6 Memory Management

10.6.1 Overview

Collaboration diagram for Memory Management:

Modules

- API functions

Data Structures

- struct nt_mem

10.6.2 Data Structure Documentation

10.6.2.1 struct nt_mem

This structure contains the memory pool for all RAM data of the NXP touch volatile data structures. This structure can be allocated in RAM.

Collaboration diagram for nt_mem:

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 NXP touch.
uint32_t	pool_size	Size of the allocated memory pool for the NXP touch.

10.6.3 API functions

10.6.3.1 Overview

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

Functions

- int32_t _nt_mem_init (uint8_t ∗pool, const uint32_t size)
  Initialized the NXP touch memory management system.
- void ∗ _nt_mem_alloc (const uint32_t size)
  Allocation of memory from the memory pool.
- int32_t _nt_mem_deinit (void)
  De-initialized the NXP touch memory management system.

10.6.3.2 Function Documentation

10.6.3.2.1 void∗ _nt_mem_alloc (const uint32_t *size*)

Parameters

size	- 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:

Here is the caller graph for this function:

10.6.3.2.2 int32_t _nt_mem_deinit (void)

Returns

The result of the operation.

Here is the call graph for this function:

10.6.3.2.3 int32_t _nt_mem_init (uint8_t * *pool,* const uint32_t *size*)

Parameters

pool	- pointer to the allocated memory place to be used by the system (the size must be aligned by 4).
size	- 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:

10.7 System

10.7.1 Overview

/

The system private API and definitions. Collaboration diagram for System:

Modules

- API Functions

Data Structures

- struct nt_kernel

Macros

- #define OSA_WAIT_FOREVER
  Constant to pass as time out value in order to wait indefinitely.

Typedefs

- typedef void ∗ nt_mutex_t

Enumerations

- enum nt_osa_status_t {
  knt_Status_OSA_Success,
  knt_Status_OSA_Error,
  knt_Status_OSA_Timeout,
  knt_Status_OSA_Idle }
  Defines the return status of OSA's functions.

- enum nt_system_module_call {
  NT_SYSTEM_MODULE_INIT,
  NT_SYSTEM_MODULE_TRIGGER,
  NT_SYSTEM_MODULE_PROCESS,
  NT_SYSTEM_MODULE_CHECK_DATA }
- enum nt_system_control_call {
  NT_SYSTEM_CONTROL_INIT,
  NT_SYSTEM_CONTROL_PROCESS,
  NT_SYSTEM_CONTROL_OVERRUN,
  NT_SYSTEM_CONTROL_DATA_READY }

10.7.2 Data Structure Documentation

10.7.2.1 struct nt_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 NXP Touch library.

Collaboration diagram for nt_kernel:

Data Fields

nt_system_-callback	callback	System event handler.

struct nt_control_data **	controls	Pointer to the list of controls. Can't be NULL.
uint8_t	controls_cnt	Count of the controls.
struct nt_mem	memory	System Memory handler
struct nt_-module_data **	modules	Pointer to the list of modules. Can't be NULL.
uint8_t	modules_cnt	Count of the modules.
struct nt_system *	rom	Pointer to the system parameters.
uint16_t	safety_crc	container for CRC result used by safety check
uint32_t	time_counter	Time counter.

10.7.3 Macro Definition Documentation

10.7.3.1 #define OSA_WAIT_FOREVER

10.7.4 Typedef Documentation

10.7.4.1 typedef void* nt_mutex_t

10.7.5 Enumeration Type Documentation

10.7.5.1 enum nt_osa_status_t

Enumerator

knt_Status_OSA_Success Success
knt_Status_OSA_Error Failed
knt_Status_OSA_Timeout Time out occurs while waiting
knt_Status_OSA_Idle Used for bare metal only, the wait object is not ready and time out still not occur

10.7.5.2 enum nt_system_control_call

Internal Controls function call identifier

Enumerator

NT_SYSTEM_CONTROL_INIT Do control initialization

NT_SYSTEM_CONTROL_PROCESS Process the new data of control
NT_SYSTEM_CONTROL_OVERRUN Control data are overrun
NT_SYSTEM_CONTROL_DATA_READY Control data are ready

10.7.5.3 enum nt_system_module_call

Internal Module function call identifier

Enumerator

NT_SYSTEM_MODULE_INIT Do module initialization
NT_SYSTEM_MODULE_TRIGGER Send trigger event to module
NT_SYSTEM_MODULE_PROCESS Do process data in the module
NT_SYSTEM_MODULE_CHECK_DATA Check the module data

10.7.6 API Functions

10.7.6.1 Overview

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

Modules

- NT_OSA

Functions

- struct nt_kernel ∗ _nt_system_get (void)
  Obtain a pointer to the system.
- void _nt_system_increment_time_counter (void)
  Increments the system counter.
- uint32_t _nt_system_get_time_period (void)
  Get time period.
- uint32_t _nt_system_get_time_offset_from_period (uint32_t event_period)
  Time offset by a defined period.
- int32_t _nt_system_module_function (uint32_t option)
  Invoke the module function based on the option parameter.
- int32_t _nt_system_control_function (uint32_t option)
  Invoke the control function based on the option parameter.
- int32_t _nt_system_init (const struct nt_system ∗system)
  Initialize system.
- void _nt_system_invoke_callback (uint32_t event, union nt_system_event_context ∗context)
  System callback invocation.
- void _nt_system_modules_data_ready (struct nt_module_data ∗module)
  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 nt_module ∗ _nt_system_get_module (uint32_t interface_address, uint32_t instance)
  Find the n-th instance of a module of a specified type.
- void nt_error (char ∗file_name, uint32_t line)
  The NT error function that is invoked from NT asserts.
- uint32_t _nt_system_count_pointer_array (const void ∗∗pointer_array)
- int32_t _nt_system_check_data (const struct nt_system ∗system)

10.7.6.2 Function Documentation

10.7.6.2.1 int32_t _nt_system_check_data (const struct nt_system * *system*)

Here is the caller graph for this function:

10.7.6.2.2 int32_t _nt_system_control_function (uint32_t *option*)

Parameters

option	One of the options from nt_system_control_call enum

Returns

- NT_SUCCESS if the control's action was carried out successfully,
- NT_FAILURE if the control's action failed.

Here is the call graph for this function:

Here is the caller graph for this function:

10.7.6.2.3 uint32_t _nt_system_count_pointer_array (const void ∗∗ *pointer_array*)

Here is the caller graph for this function:

10.7.6.2.4 struct nt_kernel∗ _nt_system_get (void)

Returns

A pointer to the system kernel structure.

Here is the caller graph for this function:

10.7.6.2.5 struct nt_module* _nt_system_get_module (uint32_t *interface_address,* uint32_t *instance*)

Parameters

interface_-address	Address to the module's interface (uniquely identifies module type).
instance	Zero-based module instance index.

Returns

- valid pointer to module.
- NULL if the module was not found

10.7.6.2.6 uint32_t _nt_system_get_time_offset_from_period (uint32_t *event_period*)

Parameters

event_period	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:

10.7.6.2.7 uint32_t _nt_system_get_time_period (void)

Returns

Time period.

Here is the caller graph for this function:

10.7.6.2.8 void _nt_system_increment_time_counter (void)

Returns

None.

Here is the caller graph for this function:

10.7.6.2.9 int32_t _nt_system_init (const struct nt_system * *system*)

Parameters

system	Pointer to the user system parameters structure.

Returns

- NT_SUCCESS if the system data are set correctly,
- NT_FAILURE if the system data check failed.

Here is the call graph for this function:

Here is the caller graph for this function:

10.7.6.2.10 void _nt_system_invoke_callback (uint32_t *event,* union nt_system_event_context * *context*)

Parameters

event	Callback event.

Returns

None.

Here is the call graph for this function:

Here is the caller graph for this function:

10.7.6.2.11 int32_t _nt_system_module_function (uint32_t *option*)

Parameters

option	One of the options from the nt_system_module_call enum

Returns

- NT_SUCCESS if the module's action was carried out successfully,
- NT_FAILURE if the module's action failed.

Here is the call graph for this function:

Here is the caller graph for this function:

10.7.6.2.12 void _nt_system_modules_data_ready (struct nt_module_data * *module*)

Returns

None.

Here is the call graph for this function:

Here is the caller graph for this function:

10.7.6.2.13 void nt_error (char * *file_name,* uint32_t *line*)

Parameters

file_name	Pointer to the file name.
line	Number of the line which was asserted.

Returns

none

Here is the call graph for this function:

10.7.6.3 NT_OSA

10.7.6.3.1 Overview

/

NT_OSA Collaboration diagram for NT_OSA:

Functions

- int32_t NT_OSA_Init ()
- int32_t NT_OSA_EnterCritical ()
- int32_t NT_OSA_ExitCritical ()
- int32_t NT_OSA_MutexCreate (nt_mutex_t *ptr_mtx)
- int32_t NT_OSA_MutexLock (nt_mutex_t mutex, uint32_t timeout)
- int32_t NT_OSA_MutexUnlock (nt_mutex_t mutex)

10.7.6.3.2 Function Documentation

10.7.6.3.2.1 int32_t NT_OSA_EnterCritical ( )

Start of critical subsection, critical code will not be pre-empted, user must define the function body according his OS.

Returns

none

10.7.6.3.2.2 int32_t NT_OSA_ExitCritical ( )

End of critical subsection, critical code will not be pre-empted, user must define the function body according his OS.

Returns

none

10.7.6.3.2.3 int32_t NT_OSA_Init ( )

Initializes NT own OSA, user must define the function body according his OS.

Returns

none

10.7.6.3.2.4 int32_t NT_OSA_MutexCreate (nt_mutex_t * *ptr_mtx*)

Create mutex that handles shared resources, user must define the function body according his OS.

Parameters

ptr_mtx	Pointer to mutex.

Returns

none

10.7.6.3.2.5 int32_t NT_OSA_MutexLock (nt_mutex_t *mutex,* uint32_t *timeout*)

Lock mutex that handles shared resources, user must define the function body according his OS.

Parameters

mutex	mutex structure.
timeout	time out for locking the mutex.

Returns

none

10.7.6.3.2.6 int32_t NT_OSA_MutexUnlock (nt_mutex_t *mutex*)

Unlock mutex that handles shared resources, user must define the function body according his OS.

Parameters

mutex	mutex structure.

Returns

none

Chapter 11
Data Structure Documentation

11.0.7 nt_keydetector_interface Struct Reference

```
#include <nt_keydetectors.h>
```

Collaboration diagram for nt_keydetector_interface:

Data Fields

- int32_t(∗ nt_keydetector_init)(struct nt_electrode_data ∗electrode)
- void(∗ nt_keydetector_enable)(struct nt_electrode_data ∗electrode, uint32_t touch)
- void(∗ nt_keydetector_process)(struct nt_electrode_data ∗electrode)
- void(∗ nt_keydetector_measure)(struct nt_electrode_data ∗electrode, uint32_t signal)
- void(∗ nt_keydetector_reset)(struct nt_electrode_data ∗electrode)
- const char ∗ name
- const uint32_t params_size

11.0.7.1 Detailed Description

The key detector interface structure represents the NXP Touch library Key Detector algorithm interface. The context data of the key detectors are stored in the Electrodes application objects.

11.0.7.2 Field Documentation

11.0.7.2.1 const char∗ nt_keydetector_interface::name

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

11.0.7.2.2 void(∗ nt_keydetector_interface::nt_keydetector_enable)(struct nt_electrode_data ∗electrode, uint32_t touch)

Key Detector enable function pointer

11.0.7.2.3 int32_t(∗ nt_keydetector_interface::nt_keydetector_init)(struct nt_electrode_data ∗electrode)

Key Detector initialization function pointer

11.0.7.2.4 void(∗ nt_keydetector_interface::nt_keydetector_measure)(struct nt_electrode_data ∗electrode, uint32_t signal)

Key Detector measure function pointer

11.0.7.2.5 void(∗ nt_keydetector_interface::nt_keydetector_process)(struct nt_electrode_data ∗electrode)

Key Detector process function pointer

11.0.7.2.6 void(∗ nt_keydetector_interface::nt_keydetector_reset)(struct nt_electrode_data ∗electrode)

Key Detector reset function pointer

11.0.7.2.7 const uint32_t nt_keydetector_interface::params_size

Structure size

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP 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. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP 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 NXP 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. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

NXP, the NXP logo, Freescale, the Freescale logo and Kinetis are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2018 NXP B.V.