

# **MCUXpresso SDK API Reference Manual**

**NXP Semiconductors**

Document Number: MCUXSDKK21FAPIRM  
Rev. 0  
Mar 2017





# Contents

**Chapter Introduction**

**Chapter Driver errors status**

**Chapter Architectural Overview**

**Chapter Trademarks**

**Chapter ADC16: 16-bit SAR Analog-to-Digital Converter Driver**

<b>5.1</b>	<b>Overview</b>	11
<b>5.2</b>	<b>Typical use case</b>	11
5.2.1	Polling Configuration	11
5.2.2	Interrupt Configuration	11
<b>5.3</b>	<b>Data Structure Documentation</b>	15
5.3.1	struct adc16_config_t	15
5.3.2	struct adc16_hardware_compare_config_t	16
5.3.3	struct adc16_channel_config_t	16
<b>5.4</b>	<b>Macro Definition Documentation</b>	17
5.4.1	FSL_ADC16_DRIVER_VERSION	17
<b>5.5</b>	<b>Enumeration Type Documentation</b>	17
5.5.1	_adc16_channel_status_flags	17
5.5.2	_adc16_status_flags	17
5.5.3	adc16_channel_mux_mode_t	17
5.5.4	adc16_clock_divider_t	18
5.5.5	adc16_resolution_t	18
5.5.6	adc16_clock_source_t	18
5.5.7	adc16_long_sample_mode_t	18
5.5.8	adc16_reference_voltage_source_t	19
5.5.9	adc16_hardware_average_mode_t	19
5.5.10	adc16_hardware_compare_mode_t	19
<b>5.6</b>	<b>Function Documentation</b>	19
5.6.1	ADC16_Init	19

# Contents

Section Number	Title	Page Number
5.6.2	ADC16_Deinit . . . . .	20
5.6.3	ADC16_GetDefaultConfig . . . . .	20
5.6.4	ADC16_DoAutoCalibration . . . . .	20
5.6.5	ADC16_SetOffsetValue . . . . .	21
5.6.6	ADC16_EnableDMA . . . . .	21
5.6.7	ADC16_EnableHardwareTrigger . . . . .	21
5.6.8	ADC16_SetChannelMuxMode . . . . .	22
5.6.9	ADC16_SetHardwareCompareConfig . . . . .	22
5.6.10	ADC16_SetHardwareAverage . . . . .	22
5.6.11	ADC16_GetStatusFlags . . . . .	23
5.6.12	ADC16_ClearStatusFlags . . . . .	23
5.6.13	ADC16_SetChannelConfig . . . . .	23
5.6.14	ADC16_GetChannelConversionValue . . . . .	25
5.6.15	ADC16_GetChannelStatusFlags . . . . .	25

## Chapter **CMP: Analog Comparator Driver**

<b>6.1</b>	<b>Overview</b> . . . . .	<b>27</b>
<b>6.2</b>	<b>Typical use case</b> . . . . .	<b>27</b>
6.2.1	Polling Configuration . . . . .	27
6.2.2	Interrupt Configuration . . . . .	28
<b>6.3</b>	<b>Data Structure Documentation</b> . . . . .	<b>30</b>
6.3.1	struct cmp_config_t . . . . .	30
6.3.2	struct cmp_filter_config_t . . . . .	31
6.3.3	struct cmp_dac_config_t . . . . .	31
<b>6.4</b>	<b>Macro Definition Documentation</b> . . . . .	<b>32</b>
6.4.1	FSL_CMP_DRIVER_VERSION . . . . .	32
<b>6.5</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>32</b>
6.5.1	_cmp_interrupt_enable . . . . .	32
6.5.2	_cmp_status_flags . . . . .	32
6.5.3	cmp_hysteresis_mode_t . . . . .	32
6.5.4	cmp_reference_voltage_source_t . . . . .	32
<b>6.6</b>	<b>Function Documentation</b> . . . . .	<b>33</b>
6.6.1	CMP_Init . . . . .	33
6.6.2	CMP_Deinit . . . . .	33
6.6.3	CMP_Enable . . . . .	33
6.6.4	CMP_GetDefaultConfig . . . . .	34
6.6.5	CMP_SetInputChannels . . . . .	34
6.6.6	CMP_EnableDMA . . . . .	34
6.6.7	CMP_EnableWindowMode . . . . .	35

# Contents

Section Number	Title	Page Number
6.6.8	CMP_EnablePassThroughMode . . . . .	35
6.6.9	CMP_SetFilterConfig . . . . .	35
6.6.10	CMP_SetDACCConfig . . . . .	35
6.6.11	CMP_EnableInterrupts . . . . .	36
6.6.12	CMP_DisableInterrupts . . . . .	36
6.6.13	CMP_GetStatusFlags . . . . .	36
6.6.14	CMP_ClearStatusFlags . . . . .	36
<b>Chapter CMT: Carrier Modulator Transmitter Driver</b>		
7.1	<b>Overview</b> . . . . .	39
7.2	<b>Clock formulas</b> . . . . .	39
7.3	<b>Typical use case</b> . . . . .	39
7.4	<b>Data Structure Documentation</b> . . . . .	42
7.4.1	struct cmt_modulate_config_t . . . . .	42
7.4.2	struct cmt_config_t . . . . .	43
7.5	<b>Macro Definition Documentation</b> . . . . .	43
7.5.1	FSL_CMT_DRIVER_VERSION . . . . .	43
7.6	<b>Enumeration Type Documentation</b> . . . . .	43
7.6.1	cmt_mode_t . . . . .	43
7.6.2	cmt_primary_clkdiv_t . . . . .	44
7.6.3	cmt_second_clkdiv_t . . . . .	44
7.6.4	cmt_infrared_output_polarity_t . . . . .	45
7.6.5	cmt_infrared_output_state_t . . . . .	45
7.6.6	_cmt_interrupt_enable . . . . .	45
7.7	<b>Function Documentation</b> . . . . .	45
7.7.1	CMT_GetDefaultConfig . . . . .	45
7.7.2	CMT_Init . . . . .	45
7.7.3	CMT_Deinit . . . . .	46
7.7.4	CMT_SetMode . . . . .	46
7.7.5	CMT_GetMode . . . . .	46
7.7.6	CMT_GetCMTFrequency . . . . .	47
7.7.7	CMT_SetCarrirGenerateCountOne . . . . .	48
7.7.8	CMT_SetCarrirGenerateCountTwo . . . . .	48
7.7.9	CMT_SetModulateMarkSpace . . . . .	49
7.7.10	CMT_EnableExtendedSpace . . . . .	49
7.7.11	CMT_SetIroState . . . . .	50
7.7.12	CMT_EnableInterrupts . . . . .	51
7.7.13	CMT_DisableInterrupts . . . . .	51
7.7.14	CMT_GetStatusFlags . . . . .	51

# Contents

Section Number	Title	Page Number
Chapter	<b>CRC: Cyclic Redundancy Check Driver</b>	
8.1	<b>Overview</b> . . . . .	53
8.2	<b>CRC Driver Initialization and Configuration</b> . . . . .	53
8.3	<b>CRC Write Data</b> . . . . .	53
8.4	<b>CRC Get Checksum</b> . . . . .	53
8.5	<b>Comments about API usage in RTOS</b> . . . . .	54
8.6	<b>Comments about API usage in interrupt handler</b> . . . . .	54
8.7	<b>CRC Driver Examples</b> . . . . .	54
8.7.1	Simple examples . . . . .	54
8.7.2	Advanced examples . . . . .	55
8.8	<b>Data Structure Documentation</b> . . . . .	58
8.8.1	struct crc_config_t . . . . .	58
8.9	<b>Macro Definition Documentation</b> . . . . .	58
8.9.1	FSL_CRC_DRIVER_VERSION . . . . .	58
8.9.2	CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT . . . . .	59
8.10	<b>Enumeration Type Documentation</b> . . . . .	59
8.10.1	crc_bits_t . . . . .	59
8.10.2	crc_result_t . . . . .	59
8.11	<b>Function Documentation</b> . . . . .	59
8.11.1	CRC_Init . . . . .	59
8.11.2	CRC_Deinit . . . . .	59
8.11.3	CRC_GetDefaultConfig . . . . .	60
8.11.4	CRC_WriteData . . . . .	60
8.11.5	CRC_Get32bitResult . . . . .	60
8.11.6	CRC_Get16bitResult . . . . .	61
Chapter	<b>DAC: Digital-to-Analog Converter Driver</b>	
9.1	<b>Overview</b> . . . . .	63
9.2	<b>Typical use case</b> . . . . .	63
9.2.1	Working as a basic DAC without the hardware buffer feature . . . . .	63
9.2.2	Working with the hardware buffer . . . . .	63
9.3	<b>Data Structure Documentation</b> . . . . .	66
9.3.1	struct dac_config_t . . . . .	66

# Contents

Section Number	Title	Page Number
9.3.2	struct dac_buffer_config_t . . . . .	66
<b>9.4</b>	<b>Macro Definition Documentation</b> . . . . .	<b>67</b>
9.4.1	FSL_DAC_DRIVER_VERSION . . . . .	67
<b>9.5</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>67</b>
9.5.1	_dac_buffer_status_flags . . . . .	67
9.5.2	_dac_buffer_interrupt_enable . . . . .	67
9.5.3	dac_reference_voltage_source_t . . . . .	68
9.5.4	dac_buffer_trigger_mode_t . . . . .	68
9.5.5	dac_buffer_watermark_t . . . . .	68
9.5.6	dac_buffer_work_mode_t . . . . .	68
<b>9.6</b>	<b>Function Documentation</b> . . . . .	<b>68</b>
9.6.1	DAC_Init . . . . .	68
9.6.2	DAC_Deinit . . . . .	69
9.6.3	DAC_GetDefaultConfig . . . . .	69
9.6.4	DAC_Enable . . . . .	69
9.6.5	DAC_EnableBuffer . . . . .	70
9.6.6	DAC_SetBufferConfig . . . . .	70
9.6.7	DAC_GetDefaultBufferConfig . . . . .	70
9.6.8	DAC_EnableBufferDMA . . . . .	70
9.6.9	DAC_SetBufferValue . . . . .	71
9.6.10	DAC_DoSoftwareTriggerBuffer . . . . .	71
9.6.11	DAC_GetBufferReadPointer . . . . .	71
9.6.12	DAC_SetBufferReadPointer . . . . .	72
9.6.13	DAC_EnableBufferInterrupts . . . . .	72
9.6.14	DAC_DisableBufferInterrupts . . . . .	72
9.6.15	DAC_GetBufferStatusFlags . . . . .	72
9.6.16	DAC_ClearBufferStatusFlags . . . . .	72

## Chapter DMAMUX: Direct Memory Access Multiplexer Driver

<b>10.1</b>	<b>Overview</b> . . . . .	<b>75</b>
<b>10.2</b>	<b>Typical use case</b> . . . . .	<b>75</b>
10.2.1	DMAMUX Operation . . . . .	75
<b>10.3</b>	<b>Macro Definition Documentation</b> . . . . .	<b>75</b>
10.3.1	FSL_DMAMUX_DRIVER_VERSION . . . . .	75
<b>10.4</b>	<b>Function Documentation</b> . . . . .	<b>76</b>
10.4.1	DMAMUX_Init . . . . .	76
10.4.2	DMAMUX_Deinit . . . . .	77
10.4.3	DMAMUX_EnableChannel . . . . .	77
10.4.4	DMAMUX_DisableChannel . . . . .	77

# Contents

Section Number	Title	Page Number
10.4.5	DMAMUX_SetSource . . . . .	78
10.4.6	DMAMUX_EnablePeriodTrigger . . . . .	78
10.4.7	DMAMUX_DisablePeriodTrigger . . . . .	78
<b>Chapter</b>	<b>DSPI: Serial Peripheral Interface Driver</b>	
<b>11.1</b>	<b>Overview</b> . . . . .	<b>79</b>
<b>11.2</b>	<b>DSPI Driver</b> . . . . .	<b>80</b>
11.2.1	Overview . . . . .	80
11.2.2	Typical use case . . . . .	80
11.2.3	Data Structure Documentation . . . . .	87
11.2.4	Macro Definition Documentation . . . . .	94
11.2.5	Typedef Documentation . . . . .	95
11.2.6	Enumeration Type Documentation . . . . .	96
11.2.7	Function Documentation . . . . .	100
<b>11.3</b>	<b>DSPI DMA Driver</b> . . . . .	<b>119</b>
11.3.1	Overview . . . . .	119
11.3.2	Data Structure Documentation . . . . .	120
11.3.3	Typedef Documentation . . . . .	123
11.3.4	Function Documentation . . . . .	124
<b>11.4</b>	<b>DSPI eDMA Driver</b> . . . . .	<b>129</b>
11.4.1	Overview . . . . .	129
11.4.2	Data Structure Documentation . . . . .	130
11.4.3	Typedef Documentation . . . . .	133
11.4.4	Function Documentation . . . . .	134
<b>11.5</b>	<b>DSPI FreeRTOS Driver</b> . . . . .	<b>139</b>
11.5.1	Overview . . . . .	139
11.5.2	Function Documentation . . . . .	139
<b>Chapter</b>	<b>eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver</b>	
<b>12.1</b>	<b>Overview</b> . . . . .	<b>141</b>
<b>12.2</b>	<b>Typical use case</b> . . . . .	<b>141</b>
12.2.1	eDMA Operation . . . . .	141
<b>12.3</b>	<b>Data Structure Documentation</b> . . . . .	<b>147</b>
12.3.1	struct edma_config_t . . . . .	147
12.3.2	struct edma_transfer_config_t . . . . .	147
12.3.3	struct edma_channel_Preemption_config_t . . . . .	148
12.3.4	struct edma_minor_offset_config_t . . . . .	149
12.3.5	struct edma_tcd_t . . . . .	149

# Contents

Section Number	Title	Page Number
12.3.6	struct edma_handle_t . . . . .	150
<b>12.4</b>	<b>Macro Definition Documentation . . . . .</b>	<b>151</b>
12.4.1	FSL_EDMA_DRIVER_VERSION . . . . .	151
<b>12.5</b>	<b>Typedef Documentation . . . . .</b>	<b>151</b>
12.5.1	edma_callback . . . . .	151
<b>12.6</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>151</b>
12.6.1	edma_transfer_size_t . . . . .	151
12.6.2	edma_modulo_t . . . . .	151
12.6.3	edma_bandwidth_t . . . . .	152
12.6.4	edma_channel_link_type_t . . . . .	152
12.6.5	_edma_channel_status_flags . . . . .	152
12.6.6	_edma_error_status_flags . . . . .	153
12.6.7	edma_interrupt_enable_t . . . . .	153
12.6.8	edma_transfer_type_t . . . . .	153
12.6.9	_edma_transfer_status . . . . .	153
<b>12.7</b>	<b>Function Documentation . . . . .</b>	<b>154</b>
12.7.1	EDMA_Init . . . . .	154
12.7.2	EDMA_Deinit . . . . .	155
12.7.3	EDMA_GetDefaultConfig . . . . .	155
12.7.4	EDMA_ResetChannel . . . . .	155
12.7.5	EDMA_SetTransferConfig . . . . .	156
12.7.6	EDMA_SetMinorOffsetConfig . . . . .	156
12.7.7	EDMA_SetChannelPreemptionConfig . . . . .	157
12.7.8	EDMA_SetChannelLink . . . . .	157
12.7.9	EDMA_SetBandWidth . . . . .	158
12.7.10	EDMA_SetModulo . . . . .	158
12.7.11	EDMA_EnableAutoStopRequest . . . . .	159
12.7.12	EDMA_EnableChannelInterrupts . . . . .	159
12.7.13	EDMA_DisableChannelInterrupts . . . . .	159
12.7.14	EDMA_TcdReset . . . . .	160
12.7.15	EDMA_TcdSetTransferConfig . . . . .	160
12.7.16	EDMA_TcdSetMinorOffsetConfig . . . . .	161
12.7.17	EDMA_TcdSetChannelLink . . . . .	162
12.7.18	EDMA_TcdSetBandWidth . . . . .	162
12.7.19	EDMA_TcdSetModulo . . . . .	163
12.7.20	EDMA_TcdEnableAutoStopRequest . . . . .	163
12.7.21	EDMA_TcdEnableInterrupts . . . . .	163
12.7.22	EDMA_TcdDisableInterrupts . . . . .	164
12.7.23	EDMA_EnableChannelRequest . . . . .	164
12.7.24	EDMA_DisableChannelRequest . . . . .	164
12.7.25	EDMA_TriggerChannelStart . . . . .	164

# Contents

Section Number	Title	Page Number
12.7.26	EDMA_GetRemainingMajorLoopCount . . . . .	165
12.7.27	EDMA_GetErrorStatusFlags . . . . .	165
12.7.28	EDMA_GetChannelStatusFlags . . . . .	166
12.7.29	EDMA_ClearChannelStatusFlags . . . . .	166
12.7.30	EDMA_CreateHandle . . . . .	166
12.7.31	EDMA_InstallTCMDMemory . . . . .	167
12.7.32	EDMA_SetCallback . . . . .	167
12.7.33	EDMA_PrepTransfer . . . . .	167
12.7.34	EDMA_SubmitTransfer . . . . .	168
12.7.35	EDMA_StartTransfer . . . . .	169
12.7.36	EDMA_StopTransfer . . . . .	169
12.7.37	EDMA_AbortTransfer . . . . .	169
12.7.38	EDMA_HandleIRQ . . . . .	169

## Chapter EWM: External Watchdog Monitor Driver

13.1	Overview . . . . .	171
13.2	Typical use case . . . . .	171
13.3	Data Structure Documentation . . . . .	172
13.3.1	struct ewm_config_t . . . . .	172
13.4	Macro Definition Documentation . . . . .	172
13.4.1	FSL_EWM_DRIVER_VERSION . . . . .	172
13.5	Enumeration Type Documentation . . . . .	172
13.5.1	_ewm_interrupt_enable_t . . . . .	172
13.5.2	_ewm_status_flags_t . . . . .	172
13.6	Function Documentation . . . . .	173
13.6.1	EWM_Init . . . . .	173
13.6.2	EWM_Deinit . . . . .	173
13.6.3	EWM_GetDefaultConfig . . . . .	173
13.6.4	EWM_EnableInterrupts . . . . .	174
13.6.5	EWM_DisableInterrupts . . . . .	174
13.6.6	EWM_GetStatusFlags . . . . .	174
13.6.7	EWM_Refresh . . . . .	175

## Chapter C90TFS Flash Driver

14.1	Overview . . . . .	177
14.2	Data Structure Documentation . . . . .	185
14.2.1	struct flash_execute_in_ram_function_config_t . . . . .	185
14.2.2	struct flash_swap_state_config_t . . . . .	186

# Contents

Section Number	Title	Page Number
14.2.3	<code>struct flash_swap_ifr_field_config_t</code>	186
14.2.4	<code>union flash_swap_ifr_field_data_t</code>	187
14.2.5	<code>union pflash_protection_status_low_t</code>	187
14.2.6	<code>struct pflash_protection_status_t</code>	188
14.2.7	<code>struct flash_prefetch_speculation_status_t</code>	188
14.2.8	<code>struct flash_protection_config_t</code>	188
14.2.9	<code>struct flash_access_config_t</code>	189
14.2.10	<code>struct flash_operation_config_t</code>	189
14.2.11	<code>struct flash_config_t</code>	190
<b>14.3</b>	<b>Macro Definition Documentation</b>	<b>192</b>
14.3.1	<code>MAKE_VERSION</code>	192
14.3.2	<code>FSL_FLASH_DRIVER_VERSION</code>	192
14.3.3	<code>FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT</code>	192
14.3.4	<code>FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT</code>	192
14.3.5	<code>FLASH_DRIVER_IS_FLASH_RESIDENT</code>	192
14.3.6	<code>FLASH_DRIVER_IS_EXPORTED</code>	192
14.3.7	<code>kStatusGroupGeneric</code>	192
14.3.8	<code>MAKE_STATUS</code>	192
14.3.9	<code>FOUR_CHAR_CODE</code>	192
<b>14.4</b>	<b>Enumeration Type Documentation</b>	<b>192</b>
14.4.1	<code>_flash_driver_version_constants</code>	192
14.4.2	<code>_flash_status</code>	193
14.4.3	<code>_flash_driver_api_keys</code>	193
14.4.4	<code>flash_margin_value_t</code>	194
14.4.5	<code>flash_security_state_t</code>	194
14.4.6	<code>flash_protection_state_t</code>	194
14.4.7	<code>flash_execute_only_access_state_t</code>	194
14.4.8	<code>flash_property_tag_t</code>	194
14.4.9	<code>_flash_execute_in_ram_function_constants</code>	195
14.4.10	<code>flash_read_resource_option_t</code>	195
14.4.11	<code>_flash_read_resource_range</code>	195
14.4.12	<code>_k3_flash_read_once_index</code>	196
14.4.13	<code>flash_flexram_function_option_t</code>	196
14.4.14	<code>flash_swap_function_option_t</code>	196
14.4.15	<code>flash_swap_control_option_t</code>	196
14.4.16	<code>flash_swap_state_t</code>	197
14.4.17	<code>flash_swap_block_status_t</code>	197
14.4.18	<code>flash_partition_flexram_load_option_t</code>	197
14.4.19	<code>flash_memory_index_t</code>	197
14.4.20	<code>flash_cache_controller_index_t</code>	197
14.4.21	<code>flash_cache_clear_process_t</code>	198
<b>14.5</b>	<b>Function Documentation</b>	<b>198</b>

# Contents

Section Number	Title	Page Number
14.5.1	FLASH_Init . . . . .	198
14.5.2	FLASH_SetCallback . . . . .	198
14.5.3	FLASH_PrepareExecuteInRamFunctions . . . . .	199
14.5.4	FLASH_EraseAll . . . . .	199
14.5.5	FLASH_Erase . . . . .	200
14.5.6	FLASH_EraseAllExecuteOnlySegments . . . . .	201
14.5.7	FLASH_Program . . . . .	202
14.5.8	FLASH_ProgramOnce . . . . .	203
14.5.9	FLASH_ProgramSection . . . . .	204
14.5.10	FLASH_EepromWrite . . . . .	205
14.5.11	FLASH_ReadResource . . . . .	206
14.5.12	FLASH_ReadOnce . . . . .	207
14.5.13	FLASH_GetSecurityState . . . . .	208
14.5.14	FLASH_SecurityBypass . . . . .	208
14.5.15	FLASH_VerifyEraseAll . . . . .	209
14.5.16	FLASH_VerifyErase . . . . .	210
14.5.17	FLASH_VerifyProgram . . . . .	211
14.5.18	FLASH_VerifyEraseAllExecuteOnlySegments . . . . .	212
14.5.19	FLASH_IsProtected . . . . .	213
14.5.20	FLASH_IsExecuteOnly . . . . .	214
14.5.21	FLASHGetProperty . . . . .	215
14.5.22	FLASH_SetProperty . . . . .	215
14.5.23	FLASH_SetFlexramFunction . . . . .	216
14.5.24	FLASH_ProgramPartition . . . . .	217
14.5.25	FLASH_PflashSetProtection . . . . .	217
14.5.26	FLASH_PflashGetProtection . . . . .	218
14.5.27	FLASH_DflashSetProtection . . . . .	218
14.5.28	FLASH_DflashGetProtection . . . . .	219
14.5.29	FLASH_EepromSetProtection . . . . .	220
14.5.30	FLASH_EepromGetProtection . . . . .	220

## Chapter FlexBus: External Bus Interface Driver

15.1	Overview . . . . .	223
15.2	FlexBus functional operation . . . . .	223
15.3	Typical use case and example . . . . .	223
15.4	Data Structure Documentation . . . . .	225
15.4.1	struct flexbus_config_t . . . . .	225
15.5	Macro Definition Documentation . . . . .	226
15.5.1	FSL_FLEXBUS_DRIVER_VERSION . . . . .	226

# Contents

Section Number	Title	Page Number
<b>15.6</b>	<b>Enumeration Type Documentation</b>	<b>226</b>
15.6.1	flexbus_port_size_t . . . . .	226
15.6.2	flexbus_write_address_hold_t . . . . .	226
15.6.3	flexbus_read_address_hold_t . . . . .	227
15.6.4	flexbus_address_setup_t . . . . .	227
15.6.5	flexbus_bytelane_shift_t . . . . .	227
15.6.6	flexbus_multiplex_group1_t . . . . .	227
15.6.7	flexbus_multiplex_group2_t . . . . .	228
15.6.8	flexbus_multiplex_group3_t . . . . .	228
15.6.9	flexbus_multiplex_group4_t . . . . .	228
15.6.10	flexbus_multiplex_group5_t . . . . .	228
<b>15.7</b>	<b>Function Documentation</b>	<b>228</b>
15.7.1	FLEXBUS_Init . . . . .	228
15.7.2	FLEXBUS_Deinit . . . . .	229
15.7.3	FLEXBUS_GetDefaultConfig . . . . .	229
<b>Chapter FlexCAN: Flex Controller Area Network Driver</b>		
<b>16.1</b>	<b>Overview</b>	<b>231</b>
<b>16.2</b>	<b>FlexCAN Driver</b>	<b>232</b>
16.2.1	Overview . . . . .	232
16.2.2	Typical use case . . . . .	232
16.2.3	Data Structure Documentation . . . . .	240
16.2.4	Macro Definition Documentation . . . . .	244
16.2.5	Typedef Documentation . . . . .	249
16.2.6	Enumeration Type Documentation . . . . .	249
16.2.7	Function Documentation . . . . .	252
<b>16.3</b>	<b>FlexCAN eDMA Driver</b>	<b>266</b>
16.3.1	Overview . . . . .	266
16.3.2	Data Structure Documentation . . . . .	266
16.3.3	Typedef Documentation . . . . .	267
16.3.4	Function Documentation . . . . .	267
<b>Chapter FTM: FlexTimer Driver</b>		
<b>17.1</b>	<b>Overview</b>	<b>271</b>
<b>17.2</b>	<b>Function groups</b>	<b>271</b>
17.2.1	Initialization and deinitialization . . . . .	271
17.2.2	PWM Operations . . . . .	271
17.2.3	Input capture operations . . . . .	271
17.2.4	Output compare operations . . . . .	272

# Contents

Section Number	Title	Page Number
17.2.5	Quad decode . . . . .	272
17.2.6	Fault operation . . . . .	272
<b>17.3</b>	<b>Register Update . . . . .</b>	<b>272</b>
<b>17.4</b>	<b>Typical use case . . . . .</b>	<b>273</b>
17.4.1	PWM output . . . . .	273
<b>17.5</b>	<b>Data Structure Documentation . . . . .</b>	<b>280</b>
17.5.1	struct ftm_chnl_pwm_signal_param_t . . . . .	280
17.5.2	struct ftm_dual_edge_capture_param_t . . . . .	280
17.5.3	struct ftm_phase_params_t . . . . .	281
17.5.4	struct ftm_fault_param_t . . . . .	281
17.5.5	struct ftm_config_t . . . . .	281
<b>17.6</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>282</b>
17.6.1	ftm_chnl_t . . . . .	282
17.6.2	ftm_fault_input_t . . . . .	283
17.6.3	ftm_pwm_mode_t . . . . .	283
17.6.4	ftm_pwm_level_select_t . . . . .	283
17.6.5	ftm_output_compare_mode_t . . . . .	283
17.6.6	ftm_input_capture_edge_t . . . . .	283
17.6.7	ftm_dual_edge_capture_mode_t . . . . .	284
17.6.8	ftm_quad_decode_mode_t . . . . .	284
17.6.9	ftm_phase_polarity_t . . . . .	284
17.6.10	ftm_deadtime_prescale_t . . . . .	284
17.6.11	ftm_clock_source_t . . . . .	284
17.6.12	ftm_clock_prescale_t . . . . .	285
17.6.13	ftm_bdm_mode_t . . . . .	285
17.6.14	ftm_fault_mode_t . . . . .	285
17.6.15	ftm_external_trigger_t . . . . .	285
17.6.16	ftm_pwm_sync_method_t . . . . .	286
17.6.17	ftm_reload_point_t . . . . .	286
17.6.18	ftm_interrupt_enable_t . . . . .	287
17.6.19	ftm_status_flags_t . . . . .	287
17.6.20	_ftm_quad_decoder_flags . . . . .	288
<b>17.7</b>	<b>Function Documentation . . . . .</b>	<b>288</b>
17.7.1	FTM_Init . . . . .	288
17.7.2	FTM_Deinit . . . . .	288
17.7.3	FTM_GetDefaultConfig . . . . .	288
17.7.4	FTM_SetupPwm . . . . .	289
17.7.5	FTM_UpdatePwmDutyCycle . . . . .	289
17.7.6	FTM_UpdateChnlEdgeLevelSelect . . . . .	290
17.7.7	FTM_SetupInputCapture . . . . .	290

# Contents

Section Number	Title	Page Number
17.7.8	FTM_SetupOutputCompare . . . . .	291
17.7.9	FTM_SetupDualEdgeCapture . . . . .	291
17.7.10	FTM_SetupFault . . . . .	291
17.7.11	FTM_EnableInterrupts . . . . .	292
17.7.12	FTM_DisableInterrupts . . . . .	292
17.7.13	FTM_GetEnabledInterrupts . . . . .	292
17.7.14	FTM_GetStatusFlags . . . . .	292
17.7.15	FTM_ClearStatusFlags . . . . .	293
17.7.16	FTM_SetTimerPeriod . . . . .	293
17.7.17	FTM_GetCurrentTimerCount . . . . .	293
17.7.18	FTM_StartTimer . . . . .	294
17.7.19	FTM_StopTimer . . . . .	294
17.7.20	FTM_SetSoftwareCtrlEnable . . . . .	294
17.7.21	FTM_SetSoftwareCtrlVal . . . . .	295
17.7.22	FTM_SetGlobalTimeBaseOutputEnable . . . . .	295
17.7.23	FTM_SetOutputMask . . . . .	295
17.7.24	FTM_SetFaultControlEnable . . . . .	295
17.7.25	FTM_SetDeadTimeEnable . . . . .	296
17.7.26	FTM_SetComplementaryEnable . . . . .	296
17.7.27	FTM_SetInvertEnable . . . . .	296
17.7.28	FTM_SetupQuadDecode . . . . .	297
17.7.29	FTM_GetQuadDecoderFlags . . . . .	297
17.7.30	FTM_SetQuadDecoderModuloValue . . . . .	297
17.7.31	FTM_GetQuadDecoderCounterValue . . . . .	298
17.7.32	FTM_ClearQuadDecoderCounterValue . . . . .	298
17.7.33	FTM_SetSoftwareTrigger . . . . .	298
17.7.34	FTM_SetWriteProtection . . . . .	298

## Chapter **GPIO: General-Purpose Input/Output Driver**

<b>18.1</b>	<b>Overview</b> . . . . .	<b>301</b>
<b>18.2</b>	<b>Data Structure Documentation</b> . . . . .	<b>301</b>
18.2.1	struct gpio_pin_config_t . . . . .	301
<b>18.3</b>	<b>Macro Definition Documentation</b> . . . . .	<b>302</b>
18.3.1	FSL_GPIO_DRIVER_VERSION . . . . .	302
<b>18.4</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>302</b>
18.4.1	gpio_pin_direction_t . . . . .	302
<b>18.5</b>	<b>GPIO Driver</b> . . . . .	<b>303</b>
18.5.1	Overview . . . . .	303
18.5.2	Typical use case . . . . .	303
18.5.3	Function Documentation . . . . .	304

# Contents

Section Number	Title	Page Number
<b>18.6</b>	<b>FGPIO Driver</b>	307
18.6.1	Typical use case	307
<b>Chapter I2C: Inter-Integrated Circuit Driver</b>		
<b>19.1</b>	<b>Overview</b>	309
<b>19.2</b>	<b>I2C Driver</b>	310
19.2.1	Overview	310
19.2.2	Typical use case	310
19.2.3	Data Structure Documentation	317
19.2.4	Macro Definition Documentation	322
19.2.5	Typedef Documentation	322
19.2.6	Enumeration Type Documentation	322
19.2.7	Function Documentation	324
<b>19.3</b>	<b>I2C eDMA Driver</b>	339
19.3.1	Overview	339
19.3.2	Data Structure Documentation	339
19.3.3	Typedef Documentation	340
19.3.4	Function Documentation	340
<b>19.4</b>	<b>I2C DMA Driver</b>	343
19.4.1	Overview	343
19.4.2	Data Structure Documentation	343
19.4.3	Typedef Documentation	344
19.4.4	Function Documentation	344
<b>19.5</b>	<b>I2C FreeRTOS Driver</b>	346
19.5.1	Overview	346
19.5.2	Function Documentation	346
<b>Chapter LLWU: Low-Leakage Wakeup Unit Driver</b>		
<b>20.1</b>	<b>Overview</b>	349
<b>20.2</b>	<b>External wakeup pins configurations</b>	349
<b>20.3</b>	<b>Internal wakeup modules configurations</b>	349
<b>20.4</b>	<b>Digital pin filter for external wakeup pin configurations</b>	349
<b>20.5</b>	<b>Data Structure Documentation</b>	350
20.5.1	struct llwu_external_pin_filter_mode_t	350
<b>20.6</b>	<b>Macro Definition Documentation</b>	350

# Contents

Section Number	Title	Page Number
20.6.1	FSL_LLWU_DRIVER_VERSION . . . . .	350
<b>20.7</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>351</b>
20.7.1	llwu_external_pin_mode_t . . . . .	351
20.7.2	llwu_pin_filter_mode_t . . . . .	351
<b>20.8</b>	<b>Function Documentation . . . . .</b>	<b>351</b>
20.8.1	LLWU_SetExternalWakeUpPinMode . . . . .	351
20.8.2	LLWU_GetExternalWakeUpPinFlag . . . . .	351
20.8.3	LLWU_ClearExternalWakeUpPinFlag . . . . .	352
20.8.4	LLWU_EnableInternalModuleInterruptWakup . . . . .	352
20.8.5	LLWU_GetInternalWakeUpModuleFlag . . . . .	352
20.8.6	LLWU_SetPinFilterMode . . . . .	353
20.8.7	LLWU_GetPinFilterFlag . . . . .	353
20.8.8	LLWU_ClearPinFilterFlag . . . . .	353
20.8.9	LLWU_SetResetPinMode . . . . .	354
<b>Chapter</b>	<b>LPTMR: Low-Power Timer</b>	
<b>21.1</b>	<b>Overview . . . . .</b>	<b>355</b>
<b>21.2</b>	<b>Function groups . . . . .</b>	<b>355</b>
21.2.1	Initialization and deinitialization . . . . .	355
21.2.2	Timer period Operations . . . . .	355
21.2.3	Start and Stop timer operations . . . . .	355
21.2.4	Status . . . . .	356
21.2.5	Interrupt . . . . .	356
<b>21.3</b>	<b>Typical use case . . . . .</b>	<b>356</b>
21.3.1	LPTMR tick example . . . . .	356
<b>21.4</b>	<b>Data Structure Documentation . . . . .</b>	<b>358</b>
21.4.1	struct lptmr_config_t . . . . .	358
<b>21.5</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>359</b>
21.5.1	lptmr_pin_select_t . . . . .	359
21.5.2	lptmr_pin_polarity_t . . . . .	359
21.5.3	lptmr_timer_mode_t . . . . .	359
21.5.4	lptmr_prescaler_glitch_value_t . . . . .	360
21.5.5	lptmr_prescaler_clock_select_t . . . . .	360
21.5.6	lptmr_interrupt_enable_t . . . . .	360
21.5.7	lptmr_status_flags_t . . . . .	361
<b>21.6</b>	<b>Function Documentation . . . . .</b>	<b>361</b>
21.6.1	LPTMR_Init . . . . .	361
21.6.2	LPTMR_Deinit . . . . .	361

# Contents

Section Number	Title	Page Number
21.6.3	LPTMR_GetDefaultConfig . . . . .	361
21.6.4	LPTMR_EnableInterrupts . . . . .	362
21.6.5	LPTMR_DisableInterrupts . . . . .	362
21.6.6	LPTMR_GetEnabledInterrupts . . . . .	362
21.6.7	LPTMR_GetStatusFlags . . . . .	362
21.6.8	LPTMR_ClearStatusFlags . . . . .	363
21.6.9	LPTMR_SetTimerPeriod . . . . .	363
21.6.10	LPTMR_GetCurrentTimerCount . . . . .	363
21.6.11	LPTMR_StartTimer . . . . .	364
21.6.12	LPTMR_StopTimer . . . . .	364
<b>Chapter</b>	<b>PDB: Programmable Delay Block</b>	
<b>22.1</b>	<b>Overview</b> . . . . .	<b>365</b>
<b>22.2</b>	<b>Typical use case</b> . . . . .	<b>365</b>
22.2.1	Working as basic PDB counter with a PDB interrupt. . . . .	365
22.2.2	Working with an additional trigger. The ADC trigger is used as an example. . . . .	366
<b>22.3</b>	<b>Data Structure Documentation</b> . . . . .	<b>370</b>
22.3.1	struct pdb_config_t . . . . .	370
22.3.2	struct pdb_adc_pretrigger_config_t . . . . .	371
22.3.3	struct pdb_dac_trigger_config_t . . . . .	371
<b>22.4</b>	<b>Macro Definition Documentation</b> . . . . .	<b>372</b>
22.4.1	FSL_PDB_DRIVER_VERSION . . . . .	372
<b>22.5</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>372</b>
22.5.1	_pdb_status_flags . . . . .	372
22.5.2	_pdb_adc_pretrigger_flags . . . . .	372
22.5.3	_pdb_interrupt_enable . . . . .	372
22.5.4	pdb_load_value_mode_t . . . . .	372
22.5.5	pdb_prescaler_divider_t . . . . .	373
22.5.6	pdb_divider_multiplication_factor_t . . . . .	373
22.5.7	pdb_trigger_input_source_t . . . . .	373
<b>22.6</b>	<b>Function Documentation</b> . . . . .	<b>374</b>
22.6.1	PDB_Init . . . . .	374
22.6.2	PDB_Deinit . . . . .	374
22.6.3	PDB_GetDefaultConfig . . . . .	375
22.6.4	PDB_Enable . . . . .	375
22.6.5	PDB_DoSoftwareTrigger . . . . .	375
22.6.6	PDB_DoLoadValues . . . . .	375
22.6.7	PDB_EnableDMA . . . . .	376
22.6.8	PDB_EnableInterrupts . . . . .	376

# Contents

Section Number	Title	Page Number
22.6.9	PDB_DisableInterrupts . . . . .	376
22.6.10	PDB_GetStatusFlags . . . . .	376
22.6.11	PDB_ClearStatusFlags . . . . .	377
22.6.12	PDB_SetModulusValue . . . . .	377
22.6.13	PDB_GetCounterValue . . . . .	377
22.6.14	PDB_SetCounterDelayValue . . . . .	377
22.6.15	PDB_SetADCPreTriggerConfig . . . . .	378
22.6.16	PDB_SetADCPreTriggerDelayValue . . . . .	378
22.6.17	PDB_GetADCPreTriggerStatusFlags . . . . .	378
22.6.18	PDB_ClearADCPreTriggerStatusFlags . . . . .	379
22.6.19	PDB_SetDACTriggerConfig . . . . .	379
22.6.20	PDB_SetDACTriggerIntervalValue . . . . .	379
22.6.21	PDB_EnablePulseOutTrigger . . . . .	380
22.6.22	PDB_SetPulseOutTriggerDelayValue . . . . .	380

## Chapter **PIT: Periodic Interrupt Timer**

<b>23.1</b>	<b>Overview</b> . . . . .	<b>381</b>
<b>23.2</b>	<b>Function groups</b> . . . . .	<b>381</b>
23.2.1	Initialization and deinitialization . . . . .	381
23.2.2	Timer period Operations . . . . .	381
23.2.3	Start and Stop timer operations . . . . .	381
23.2.4	Status . . . . .	382
23.2.5	Interrupt . . . . .	382
<b>23.3</b>	<b>Typical use case</b> . . . . .	<b>382</b>
23.3.1	PIT tick example . . . . .	382
<b>23.4</b>	<b>Data Structure Documentation</b> . . . . .	<b>384</b>
23.4.1	struct pit_config_t . . . . .	384
<b>23.5</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>384</b>
23.5.1	pit_chnl_t . . . . .	384
23.5.2	pit_interrupt_enable_t . . . . .	385
23.5.3	pit_status_flags_t . . . . .	385
<b>23.6</b>	<b>Function Documentation</b> . . . . .	<b>385</b>
23.6.1	PIT_Init . . . . .	385
23.6.2	PIT_Deinit . . . . .	385
23.6.3	PIT_GetDefaultConfig . . . . .	385
23.6.4	PIT_SetTimerChainMode . . . . .	386
23.6.5	PIT_EnableInterrupts . . . . .	386
23.6.6	PIT_DisableInterrupts . . . . .	386
23.6.7	PIT_GetEnabledInterrupts . . . . .	387

# Contents

Section Number	Title	Page Number
23.6.8	PIT_GetStatusFlags . . . . .	387
23.6.9	PIT_ClearStatusFlags . . . . .	387
23.6.10	PIT_SetTimerPeriod . . . . .	388
23.6.11	PIT_GetCurrentTimerCount . . . . .	388
23.6.12	PIT_StartTimer . . . . .	389
23.6.13	PIT_StopTimer . . . . .	389

## Chapter **PMC: Power Management Controller**

<b>24.1</b>	<b>Overview</b> . . . . .	<b>391</b>
<b>24.2</b>	<b>Data Structure Documentation</b> . . . . .	<b>392</b>
24.2.1	struct pmc_low_volt_detect_config_t . . . . .	392
24.2.2	struct pmc_low_volt_warning_config_t . . . . .	392
24.2.3	struct pmc_bandgap_buffer_config_t . . . . .	392
<b>24.3</b>	<b>Macro Definition Documentation</b> . . . . .	<b>393</b>
24.3.1	FSL_PMC_DRIVER_VERSION . . . . .	393
<b>24.4</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>393</b>
24.4.1	pmc_low_volt_detect_volt_select_t . . . . .	393
24.4.2	pmc_low_volt_warning_volt_select_t . . . . .	393
<b>24.5</b>	<b>Function Documentation</b> . . . . .	<b>393</b>
24.5.1	PMC_ConfigureLowVoltDetect . . . . .	393
24.5.2	PMC_GetLowVoltDetectFlag . . . . .	394
24.5.3	PMC_ClearLowVoltDetectFlag . . . . .	394
24.5.4	PMC_ConfigureLowVoltWarning . . . . .	394
24.5.5	PMC_GetLowVoltWarningFlag . . . . .	395
24.5.6	PMC_ClearLowVoltWarningFlag . . . . .	395
24.5.7	PMC_ConfigureBandgapBuffer . . . . .	395
24.5.8	PMC_GetPeriphIOIsolationFlag . . . . .	396
24.5.9	PMC_ClearPeriphIOIsolationFlag . . . . .	396
24.5.10	PMC_IsRegulatorInRunRegulation . . . . .	396

## Chapter **PORT: Port Control and Interrupts**

<b>25.1</b>	<b>Overview</b> . . . . .	<b>399</b>
<b>25.2</b>	<b>Typical configuration use case</b> . . . . .	<b>399</b>
25.2.1	Input PORT configuration . . . . .	399
25.2.2	I2C PORT Configuration . . . . .	399
<b>25.3</b>	<b>Data Structure Documentation</b> . . . . .	<b>402</b>
25.3.1	struct port_digital_filter_config_t . . . . .	402
25.3.2	struct port_pin_config_t . . . . .	402

## Contents

Section Number	Title	Page Number
<b>25.4</b>	<b>Macro Definition Documentation</b>	<b>402</b>
25.4.1	FSL_PORT_DRIVER_VERSION	402
<b>25.5</b>	<b>Enumeration Type Documentation</b>	<b>402</b>
25.5.1	_port_pull	402
25.5.2	_port_slew_rate	403
25.5.3	_port_open_drain_enable	403
25.5.4	_port_passive_filter_enable	403
25.5.5	_port_drive_strength	403
25.5.6	_port_lock_register	403
25.5.7	port_mux_t	403
25.5.8	port_interrupt_t	404
25.5.9	port_digital_filter_clock_source_t	404
<b>25.6</b>	<b>Function Documentation</b>	<b>404</b>
25.6.1	PORT_SetPinConfig	404
25.6.2	PORT_SetMultiplePinsConfig	405
25.6.3	PORT_SetPinMux	405
25.6.4	PORT_EnablePinsDigitalFilter	406
25.6.5	PORT_SetDigitalFilterConfig	406
25.6.6	PORT_SetPinInterruptConfig	406
25.6.7	PORT_GetPinsInterruptFlags	407
25.6.8	PORT_ClearPinsInterruptFlags	408
<b>Chapter RCM: Reset Control Module Driver</b>		
<b>26.1</b>	<b>Overview</b>	<b>411</b>
<b>26.2</b>	<b>Data Structure Documentation</b>	<b>412</b>
26.2.1	struct rcm_reset_pin_filter_config_t	412
<b>26.3</b>	<b>Macro Definition Documentation</b>	<b>412</b>
26.3.1	FSL_RCM_DRIVER_VERSION	412
<b>26.4</b>	<b>Enumeration Type Documentation</b>	<b>412</b>
26.4.1	rcm_reset_source_t	412
26.4.2	rcm_run_wait_filter_mode_t	413
<b>26.5</b>	<b>Function Documentation</b>	<b>413</b>
26.5.1	RCM_GetPreviousResetSources	413
26.5.2	RCM_ConfigureResetPinFilter	413
26.5.3	RCM_GetEasyPortModePinStatus	414
<b>Chapter RNGA: Random Number Generator Accelerator Driver</b>		
<b>27.1</b>	<b>Overview</b>	<b>415</b>

# Contents

Section Number	Title	Page Number
27.2	RNGA Initialization . . . . .	415
27.3	Get random data from RNGA . . . . .	415
27.4	RNGA Set/Get Working Mode . . . . .	415
27.5	Seed RNGA . . . . .	415
27.6	Macro Definition Documentation . . . . .	417
27.6.1	FSL_RNGA_DRIVER_VERSION . . . . .	417
27.7	Enumeration Type Documentation . . . . .	417
27.7.1	rnga_mode_t . . . . .	417
27.8	Function Documentation . . . . .	417
27.8.1	RNGA_Init . . . . .	417
27.8.2	RNGA_Deinit . . . . .	417
27.8.3	RNGA_GetRandomData . . . . .	418
27.8.4	RNGA_Seed . . . . .	418
27.8.5	RNGA_SetMode . . . . .	418
27.8.6	RNGA_GetMode . . . . .	419

## Chapter RTC: Real Time Clock

28.1	Overview . . . . .	421
28.2	Function groups . . . . .	421
28.2.1	Initialization and deinitialization . . . . .	421
28.2.2	Set & Get Datetime . . . . .	421
28.2.3	Set & Get Alarm . . . . .	421
28.2.4	Start & Stop timer . . . . .	422
28.2.5	Status . . . . .	422
28.2.6	Interrupt . . . . .	422
28.2.7	RTC Oscillator . . . . .	422
28.2.8	Monotonic Counter . . . . .	422
28.3	Typical use case . . . . .	422
28.3.1	RTC tick example . . . . .	422
28.4	Data Structure Documentation . . . . .	426
28.4.1	struct rtc_datetime_t . . . . .	426
28.4.2	struct rtc_config_t . . . . .	426
28.5	Enumeration Type Documentation . . . . .	427
28.5.1	rtc_interrupt_enable_t . . . . .	427
28.5.2	rtc_status_flags_t . . . . .	427

# Contents

Section Number	Title	Page Number
28.5.3	<a href="#">rtc_osc_cap_load_t</a>	427
<b>28.6</b>	<b>Function Documentation</b>	<b>427</b>
28.6.1	<a href="#">RTC_Init</a>	427
28.6.2	<a href="#">RTC_Deinit</a>	428
28.6.3	<a href="#">RTC_GetDefaultConfig</a>	428
28.6.4	<a href="#">RTC_SetDatetime</a>	428
28.6.5	<a href="#">RTC_GetDatetime</a>	429
28.6.6	<a href="#">RTC_SetAlarm</a>	429
28.6.7	<a href="#">RTC_GetAlarm</a>	429
28.6.8	<a href="#">RTC_EnableInterrupts</a>	430
28.6.9	<a href="#">RTC_DisableInterrupts</a>	430
28.6.10	<a href="#">RTC_GetEnabledInterrupts</a>	430
28.6.11	<a href="#">RTC_GetStatusFlags</a>	430
28.6.12	<a href="#">RTC_ClearStatusFlags</a>	431
28.6.13	<a href="#">RTC_StartTimer</a>	431
28.6.14	<a href="#">RTC_StopTimer</a>	431
28.6.15	<a href="#">RTC_SetOscCapLoad</a>	431
28.6.16	<a href="#">RTC_Reset</a>	432
28.6.17	<a href="#">RTC_GetMonotonicCounter</a>	432
28.6.18	<a href="#">RTC_SetMonotonicCounter</a>	432
28.6.19	<a href="#">RTC_IncrementMonotonicCounter</a>	432

## Chapter [SAI: Serial Audio Interface](#)

<b>29.1</b>	<b>Overview</b>	<b>435</b>
<b>29.2</b>	<b>Typical use case</b>	<b>435</b>
29.2.1	<a href="#">SAI Send/receive using an interrupt method</a>	435
29.2.2	<a href="#">SAI Send/receive using a DMA method</a>	436
<b>29.3</b>	<b>Data Structure Documentation</b>	<b>441</b>
29.3.1	<a href="#">struct sai_config_t</a>	441
29.3.2	<a href="#">struct sai_transfer_format_t</a>	442
29.3.3	<a href="#">struct sai_transfer_t</a>	442
29.3.4	<a href="#">struct _sai_handle</a>	443
<b>29.4</b>	<b>Macro Definition Documentation</b>	<b>443</b>
29.4.1	<a href="#">SAI_XFER_QUEUE_SIZE</a>	443
<b>29.5</b>	<b>Enumeration Type Documentation</b>	<b>443</b>
29.5.1	<a href="#">_sai_status_t</a>	443
29.5.2	<a href="#">sai_protocol_t</a>	444
29.5.3	<a href="#">sai_master_slave_t</a>	444
29.5.4	<a href="#">sai_mono_stereo_t</a>	444

# Contents

Section Number	Title	Page Number
29.5.5	sai_sync_mode_t . . . . .	444
29.5.6	sai_mclk_source_t . . . . .	444
29.5.7	sai_bclk_source_t . . . . .	445
29.5.8	_sai_interrupt_enable_t . . . . .	445
29.5.9	_sai_dma_enable_t . . . . .	445
29.5.10	_sai_flags . . . . .	445
29.5.11	sai_reset_type_t . . . . .	445
29.5.12	sai_sample_rate_t . . . . .	446
29.5.13	sai_word_width_t . . . . .	446
<b>29.6</b>	<b>Function Documentation</b> . . . . .	<b>446</b>
29.6.1	SAI_TxInit . . . . .	446
29.6.2	SAI_RxInit . . . . .	447
29.6.3	SAI_TxGetDefaultConfig . . . . .	447
29.6.4	SAI_RxGetDefaultConfig . . . . .	447
29.6.5	SAI_Deinit . . . . .	448
29.6.6	SAI_TxReset . . . . .	448
29.6.7	SAI_RxReset . . . . .	448
29.6.8	SAI_TxEnable . . . . .	448
29.6.9	SAI_RxEnable . . . . .	449
29.6.10	SAI_TxGetStatusFlag . . . . .	449
29.6.11	SAI_TxClearStatusFlags . . . . .	449
29.6.12	SAI_RxGetStatusFlag . . . . .	449
29.6.13	SAI_RxClearStatusFlags . . . . .	450
29.6.14	SAI_TxEnableInterrupts . . . . .	450
29.6.15	SAI_RxEnableInterrupts . . . . .	450
29.6.16	SAI_TxDisableInterrupts . . . . .	451
29.6.17	SAI_RxDisableInterrupts . . . . .	452
29.6.18	SAI_TxEnableDMA . . . . .	452
29.6.19	SAI_RxEnableDMA . . . . .	452
29.6.20	SAI_TxGetDataRegisterAddress . . . . .	453
29.6.21	SAI_RxGetDataRegisterAddress . . . . .	454
29.6.22	SAI_TxSetFormat . . . . .	454
29.6.23	SAI_RxSetFormat . . . . .	455
29.6.24	SAI_WriteBlocking . . . . .	455
29.6.25	SAI_WriteData . . . . .	455
29.6.26	SAI_ReadBlocking . . . . .	456
29.6.27	SAI_ReadData . . . . .	456
29.6.28	SAI_TransferTxCreateHandle . . . . .	456
29.6.29	SAI_TransferRxCreateHandle . . . . .	457
29.6.30	SAI_TransferTxSetFormat . . . . .	457
29.6.31	SAI_TransferRxSetFormat . . . . .	458
29.6.32	SAI_TransferSendNonBlocking . . . . .	458
29.6.33	SAI_TransferReceiveNonBlocking . . . . .	459
29.6.34	SAI_TransferGetSendCount . . . . .	459

# Contents

Section Number	Title	Page Number
29.6.35	SAI_TransferGetReceiveCount . . . . .	460
29.6.36	SAI_TransferAbortSend . . . . .	460
29.6.37	SAI_TransferAbortReceive . . . . .	461
29.6.38	SAI_TransferTxHandleIRQ . . . . .	461
29.6.39	SAI_TransferRxHandleIRQ . . . . .	461
<b>29.7</b>	<b>SAI DMA Driver . . . . .</b>	<b>462</b>
29.7.1	Overview . . . . .	462
29.7.2	Data Structure Documentation . . . . .	463
29.7.3	Function Documentation . . . . .	463
<b>29.8</b>	<b>SAI eDMA Driver . . . . .</b>	<b>469</b>
29.8.1	Overview . . . . .	469
29.8.2	Data Structure Documentation . . . . .	470
29.8.3	Function Documentation . . . . .	471
 <b>Chapter SDHC: Secure Digital Host Controller Driver</b>		
<b>30.1</b>	<b>Overview . . . . .</b>	<b>477</b>
<b>30.2</b>	<b>Typical use case . . . . .</b>	<b>477</b>
30.2.1	SDHC Operation . . . . .	477
<b>30.3</b>	<b>Data Structure Documentation . . . . .</b>	<b>485</b>
30.3.1	struct sdhc_adma2_descriptor_t . . . . .	485
30.3.2	struct sdhc_capability_t . . . . .	486
30.3.3	struct sdhc_transfer_config_t . . . . .	486
30.3.4	struct sdhc_boot_config_t . . . . .	486
30.3.5	struct sdhc_config_t . . . . .	487
30.3.6	struct sdhc_data_t . . . . .	488
30.3.7	struct sdhc_command_t . . . . .	488
30.3.8	struct sdhc_transfer_t . . . . .	489
30.3.9	struct sdhc_transfer_callback_t . . . . .	489
30.3.10	struct _sdhc_handle . . . . .	489
30.3.11	struct sdhc_host_t . . . . .	490
<b>30.4</b>	<b>Macro Definition Documentation . . . . .</b>	<b>490</b>
30.4.1	FSL_SDHC_DRIVER_VERSION . . . . .	490
<b>30.5</b>	<b>Typedef Documentation . . . . .</b>	<b>490</b>
30.5.1	sdhc_adma1_descriptor_t . . . . .	490
30.5.2	sdhc_transfer_function_t . . . . .	490
<b>30.6</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>490</b>
30.6.1	_sdhc_status . . . . .	490
30.6.2	_sdhc_capability_flag . . . . .	491

# Contents

Section Number	Title	Page Number
30.6.3	<a href="#">_sdhc_wakeup_event</a>	491
30.6.4	<a href="#">_sdhc_reset</a>	491
30.6.5	<a href="#">_sdhc_transfer_flag</a>	491
30.6.6	<a href="#">_sdhc_present_status_flag</a>	492
30.6.7	<a href="#">_sdhc_interrupt_status_flag</a>	492
30.6.8	<a href="#">_sdhc_auto_command12_error_status_flag</a>	493
30.6.9	<a href="#">_sdhc_adma_error_status_flag</a>	493
30.6.10	<a href="#">sdhc_adma_error_state_t</a>	493
30.6.11	<a href="#">_sdhc_force_event</a>	494
30.6.12	<a href="#">sdhc_data_bus_width_t</a>	494
30.6.13	<a href="#">sdhc_endian_mode_t</a>	494
30.6.14	<a href="#">sdhc_dma_mode_t</a>	495
30.6.15	<a href="#">_sdhc_sdio_control_flag</a>	495
30.6.16	<a href="#">sdhc_boot_mode_t</a>	495
30.6.17	<a href="#">sdhc_card_command_type_t</a>	495
30.6.18	<a href="#">sdhc_card_response_type_t</a>	496
30.6.19	<a href="#">_sdhc_adma1_descriptor_flag</a>	496
30.6.20	<a href="#">_sdhc_adma2_descriptor_flag</a>	496
<b>30.7</b>	<b><a href="#">Function Documentation</a></b>	<b>497</b>
30.7.1	<a href="#">SDHC_Init</a>	497
30.7.2	<a href="#">SDHC_Deinit</a>	497
30.7.3	<a href="#">SDHC_Reset</a>	497
30.7.4	<a href="#">SDHC_SetAdmaTableConfig</a>	498
30.7.5	<a href="#">SDHC_EnableInterruptStatus</a>	498
30.7.6	<a href="#">SDHC_DisableInterruptStatus</a>	498
30.7.7	<a href="#">SDHC_EnableInterruptSignal</a>	499
30.7.8	<a href="#">SDHC_DisableInterruptSignal</a>	499
30.7.9	<a href="#">SDHC_GetInterruptStatusFlags</a>	499
30.7.10	<a href="#">SDHC_ClearInterruptStatusFlags</a>	499
30.7.11	<a href="#">SDHC_GetAutoCommand12ErrorStatusFlags</a>	500
30.7.12	<a href="#">SDHC_GetAdmaErrorStatusFlags</a>	500
30.7.13	<a href="#">SDHC_GetPresentStatusFlags</a>	500
30.7.14	<a href="#">SDHC_GetCapability</a>	501
30.7.15	<a href="#">SDHC_EnableSdClock</a>	501
30.7.16	<a href="#">SDHC_SetSdClock</a>	501
30.7.17	<a href="#">SDHC_SetCardActive</a>	502
30.7.18	<a href="#">SDHC_SetDataBusWidth</a>	503
30.7.19	<a href="#">SDHC_SetTransferConfig</a>	503
30.7.20	<a href="#">SDHC_GetCommandResponse</a>	504
30.7.21	<a href="#">SDHC_WriteData</a>	504
30.7.22	<a href="#">SDHC_ReadData</a>	504
30.7.23	<a href="#">SDHC_EnableWakeupEvent</a>	505
30.7.24	<a href="#">SDHC_EnableCardDetectTest</a>	506
30.7.25	<a href="#">SDHC_SetCardDetectTestLevel</a>	506

# Contents

Section Number	Title	Page Number
30.7.26	SDHC_EnableSdioControl . . . . .	506
30.7.27	SDHC_SetContinueRequest . . . . .	507
30.7.28	SDHC_SetMmcBootConfig . . . . .	507
30.7.29	SDHC_SetForceEvent . . . . .	507
30.7.30	SDHC_TransferBlocking . . . . .	508
30.7.31	SDHC_TransferCreateHandle . . . . .	508
30.7.32	SDHC_TransferNonBlocking . . . . .	509
30.7.33	SDHC_TransferHandleIRQ . . . . .	510

## Chapter SIM: System Integration Module Driver

31.1	<b>Overview</b> . . . . .	511
31.2	<b>Data Structure Documentation</b> . . . . .	512
31.2.1	<b>struct sim_uid_t</b> . . . . .	512
31.3	<b>Enumeration Type Documentation</b> . . . . .	512
31.3.1	<b>_sim_usb_volt_reg_enable_mode</b> . . . . .	512
31.3.2	<b>_sim_flash_mode</b> . . . . .	512
31.4	<b>Function Documentation</b> . . . . .	512
31.4.1	<b>SIM_SetUsbVoltRegulatorEnableMode</b> . . . . .	512
31.4.2	<b>SIM_GetUniqueId</b> . . . . .	514
31.4.3	<b>SIM_SetFlashMode</b> . . . . .	514

## Chapter Smart Card

32.1	<b>Overview</b> . . . . .	515
32.2	<b>SmartCard Driver Initialization</b> . . . . .	515
32.3	<b>SmartCard Call diagram</b> . . . . .	515
32.4	<b>Data Structure Documentation</b> . . . . .	517
32.4.1	<b>struct smartcard_card_params_t</b> . . . . .	517
32.4.2	<b>struct smartcard_timers_state_t</b> . . . . .	518
32.4.3	<b>struct smartcard_interface_config_t</b> . . . . .	519
32.4.4	<b>struct smartcard_xfer_t</b> . . . . .	520
32.4.5	<b>struct smartcard_context_t</b> . . . . .	520
32.5	<b>Enumeration Type Documentation</b> . . . . .	522
32.5.1	<b>smartcard_status_t</b> . . . . .	522
32.5.2	<b>smartcard_control_t</b> . . . . .	522
32.5.3	<b>smartcard_direction_t</b> . . . . .	522
32.6	<b>Smart Card PHY TDA8035 Driver</b> . . . . .	523

# Contents

Section Number	Title	Page Number
32.6.1	Overview . . . . .	523
32.6.2	Macro Definition Documentation . . . . .	524
32.6.3	Function Documentation . . . . .	524
<b>32.7</b>	<b>Smart Card PHY EMVSIM Driver</b> . . . . .	<b>527</b>
32.7.1	Overview . . . . .	527
32.7.2	Macro Definition Documentation . . . . .	528
32.7.3	Function Documentation . . . . .	528
<b>32.8</b>	<b>Smart Card PHY GPIO Driver</b> . . . . .	<b>532</b>
32.8.1	Overview . . . . .	532
32.8.2	Macro Definition Documentation . . . . .	533
32.8.3	Function Documentation . . . . .	533
<b>32.9</b>	<b>Smart Card UART Driver</b> . . . . .	<b>537</b>
32.9.1	Overview . . . . .	537
32.9.2	Function Documentation . . . . .	538
<b>32.10</b>	<b>Smart Card EMVSIM Driver</b> . . . . .	<b>543</b>
32.10.1	Overview . . . . .	543
32.10.2	Enumeration Type Documentation . . . . .	544
32.10.3	Function Documentation . . . . .	545
<b>32.11</b>	<b>Smart Card FreeRTOS Driver</b> . . . . .	<b>549</b>
32.11.1	Overview . . . . .	549
32.11.2	Data Structure Documentation . . . . .	550
32.11.3	Macro Definition Documentation . . . . .	551
32.11.4	Function Documentation . . . . .	551
<b>Chapter</b>	<b>SMC: System Mode Controller Driver</b>	
<b>33.1</b>	<b>Overview</b> . . . . .	<b>557</b>
<b>33.2</b>	<b>Typical use case</b> . . . . .	<b>557</b>
33.2.1	Enter wait or stop modes . . . . .	557
<b>33.3</b>	<b>Data Structure Documentation</b> . . . . .	<b>559</b>
33.3.1	struct smc_power_mode_vlls_config_t . . . . .	559
<b>33.4</b>	<b>Macro Definition Documentation</b> . . . . .	<b>559</b>
33.4.1	FSL_SMC_DRIVER_VERSION . . . . .	559
<b>33.5</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>560</b>
33.5.1	smc_power_mode_protection_t . . . . .	560
33.5.2	smc_power_state_t . . . . .	560
33.5.3	smc_run_mode_t . . . . .	560

# Contents

Section Number	Title	Page Number
33.5.4	smc_stop_mode_t . . . . .	560
33.5.5	smc_stop_submode_t . . . . .	561
33.5.6	smc_partial_stop_option_t . . . . .	561
33.5.7	_smc_status . . . . .	561
<b>33.6</b>	<b>Function Documentation</b> . . . . .	<b>561</b>
33.6.1	SMC_SetPowerModeProtection . . . . .	561
33.6.2	SMC_GetPowerModeState . . . . .	562
33.6.3	SMC_PreEnterStopModes . . . . .	562
33.6.4	SMC_PostExitStopModes . . . . .	562
33.6.5	SMC_PreEnterWaitModes . . . . .	562
33.6.6	SMC_PostExitWaitModes . . . . .	562
33.6.7	SMC_SetPowerModeRun . . . . .	563
33.6.8	SMC_SetPowerModeWait . . . . .	564
33.6.9	SMC_SetPowerModeStop . . . . .	564
33.6.10	SMC_SetPowerModeVlpr . . . . .	564
33.6.11	SMC_SetPowerModeVlpw . . . . .	565
33.6.12	SMC_SetPowerModeVlps . . . . .	565
33.6.13	SMC_SetPowerModeLls . . . . .	565
33.6.14	SMC_SetPowerModeVlls . . . . .	565
<b>Chapter</b>	<b>UART: Universal Asynchronous Receiver/Transmitter Driver</b>	
<b>34.1</b>	<b>Overview</b> . . . . .	<b>567</b>
<b>34.2</b>	<b>UART Driver</b> . . . . .	<b>568</b>
34.2.1	Overview . . . . .	568
34.2.2	Typical use case . . . . .	568
34.2.3	Data Structure Documentation . . . . .	576
34.2.4	Macro Definition Documentation . . . . .	578
34.2.5	Typedef Documentation . . . . .	578
34.2.6	Enumeration Type Documentation . . . . .	578
34.2.7	Function Documentation . . . . .	580
<b>34.3</b>	<b>UART DMA Driver</b> . . . . .	<b>593</b>
34.3.1	Overview . . . . .	593
34.3.2	Data Structure Documentation . . . . .	593
34.3.3	Typedef Documentation . . . . .	594
34.3.4	Function Documentation . . . . .	594
<b>34.4</b>	<b>UART eDMA Driver</b> . . . . .	<b>598</b>
34.4.1	Overview . . . . .	598
34.4.2	Data Structure Documentation . . . . .	598
34.4.3	Typedef Documentation . . . . .	599
34.4.4	Function Documentation . . . . .	599

# Contents

Section Number	Title	Page Number
<b>34.5</b>	<b>UART FreeRTOS Driver</b>	<b>603</b>
34.5.1	Overview . . . . .	603
34.5.2	Data Structure Documentation . . . . .	603
34.5.3	Function Documentation . . . . .	604
<b>Chapter</b>	<b>VREF: Voltage Reference Driver</b>	
<b>35.1</b>	<b>Overview</b> . . . . .	<b>607</b>
<b>35.2</b>	<b>Typical use case and example</b> . . . . .	<b>607</b>
<b>35.3</b>	<b>Data Structure Documentation</b> . . . . .	<b>608</b>
35.3.1	struct vref_config_t . . . . .	608
<b>35.4</b>	<b>Macro Definition Documentation</b> . . . . .	<b>608</b>
35.4.1	FSL_VREF_DRIVER_VERSION . . . . .	608
<b>35.5</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>608</b>
35.5.1	vref_buffer_mode_t . . . . .	608
<b>35.6</b>	<b>Function Documentation</b> . . . . .	<b>608</b>
35.6.1	VREF_Init . . . . .	608
35.6.2	VREF_Deinit . . . . .	609
35.6.3	VREF_GetDefaultConfig . . . . .	609
35.6.4	VREF_SetTrimVal . . . . .	609
35.6.5	VREF_GetTrimVal . . . . .	610
<b>Chapter</b>	<b>WDOG: Watchdog Timer Driver</b>	
<b>36.1</b>	<b>Overview</b> . . . . .	<b>611</b>
<b>36.2</b>	<b>Typical use case</b> . . . . .	<b>611</b>
<b>36.3</b>	<b>Data Structure Documentation</b> . . . . .	<b>613</b>
36.3.1	struct wdog_work_mode_t . . . . .	613
36.3.2	struct wdog_config_t . . . . .	613
36.3.3	struct wdog_test_config_t . . . . .	614
<b>36.4</b>	<b>Macro Definition Documentation</b> . . . . .	<b>614</b>
36.4.1	FSL_WDOG_DRIVER_VERSION . . . . .	614
<b>36.5</b>	<b>Enumeration Type Documentation</b> . . . . .	<b>614</b>
36.5.1	wdog_clock_source_t . . . . .	614
36.5.2	wdog_clock_prescaler_t . . . . .	614
36.5.3	wdog_test_mode_t . . . . .	615
36.5.4	wdog_tested_byte_t . . . . .	615

# Contents

Section Number	Title	Page Number
36.5.5	_wdog_interrupt_enable_t . . . . .	615
36.5.6	_wdog_status_flags_t . . . . .	615
<b>36.6</b>	<b>Function Documentation</b> . . . . .	<b>615</b>
36.6.1	WDOG_GetDefaultConfig . . . . .	615
36.6.2	WDOG_Init . . . . .	616
36.6.3	WDOG_Deinit . . . . .	616
36.6.4	WDOG_SetTestModeConfig . . . . .	617
36.6.5	WDOG_Enable . . . . .	617
36.6.6	WDOG_Disable . . . . .	617
36.6.7	WDOG_EnableInterrupts . . . . .	618
36.6.8	WDOG_DisableInterrupts . . . . .	618
36.6.9	WDOG_GetStatusFlags . . . . .	618
36.6.10	WDOG_ClearStatusFlags . . . . .	619
36.6.11	WDOG_SetTimeoutValue . . . . .	619
36.6.12	WDOG_SetWindowValue . . . . .	620
36.6.13	WDOG_Unlock . . . . .	620
36.6.14	WDOG_Refresh . . . . .	620
36.6.15	WDOG_GetResetCount . . . . .	621
36.6.16	WDOG_ClearResetCount . . . . .	622
<b>Chapter</b>	<b>Clock Driver</b>	
<b>37.1</b>	<b>Overview</b> . . . . .	<b>623</b>
<b>37.2</b>	<b>Get frequency</b> . . . . .	<b>623</b>
<b>37.3</b>	<b>External clock frequency</b> . . . . .	<b>623</b>
<b>37.4</b>	<b>Data Structure Documentation</b> . . . . .	<b>631</b>
37.4.1	struct sim_clock_config_t . . . . .	631
37.4.2	struct oscer_config_t . . . . .	631
37.4.3	struct osc_config_t . . . . .	632
37.4.4	struct mcg_pll_config_t . . . . .	632
37.4.5	struct mcg_config_t . . . . .	633
<b>37.5</b>	<b>Macro Definition Documentation</b> . . . . .	<b>634</b>
37.5.1	MCG_CONFIG_CHECK_PARAM . . . . .	634
37.5.2	FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL . . . . .	634
37.5.3	FSL_CLOCK_DRIVER_VERSION . . . . .	635
37.5.4	DMAMUX_CLOCKS . . . . .	635
37.5.5	RTC_CLOCKS . . . . .	635
37.5.6	SAI_CLOCKS . . . . .	635
37.5.7	PORT_CLOCKS . . . . .	635
37.5.8	FLEXBUS_CLOCKS . . . . .	635

# Contents

Section Number	Title	Page Number
37.5.9	EWM_CLOCKS . . . . .	636
37.5.10	PIT_CLOCKS . . . . .	636
37.5.11	DSPI_CLOCKS . . . . .	636
37.5.12	LPTMR_CLOCKS . . . . .	636
37.5.13	SDHC_CLOCKS . . . . .	636
37.5.14	FTM_CLOCKS . . . . .	637
37.5.15	EDMA_CLOCKS . . . . .	637
37.5.16	FLEXCAN_CLOCKS . . . . .	637
37.5.17	DAC_CLOCKS . . . . .	637
37.5.18	ADC16_CLOCKS . . . . .	637
37.5.19	SYSMPU_CLOCKS . . . . .	638
37.5.20	VREF_CLOCKS . . . . .	638
37.5.21	CMT_CLOCKS . . . . .	638
37.5.22	UART_CLOCKS . . . . .	638
37.5.23	RNGA_CLOCKS . . . . .	638
37.5.24	CRC_CLOCKS . . . . .	639
37.5.25	I2C_CLOCKS . . . . .	639
37.5.26	PDB_CLOCKS . . . . .	639
37.5.27	FTF_CLOCKS . . . . .	639
37.5.28	CMP_CLOCKS . . . . .	639
37.5.29	SYS_CLK . . . . .	640
<b>37.6</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>640</b>
37.6.1	clock_name_t . . . . .	640
37.6.2	clock_usb_src_t . . . . .	640
37.6.3	clock_ip_name_t . . . . .	640
37.6.4	osc_mode_t . . . . .	640
37.6.5	_osc_cap_load . . . . .	641
37.6.6	_oscer_enable_mode . . . . .	641
37.6.7	mcg_fll_src_t . . . . .	641
37.6.8	mcg_irc_mode_t . . . . .	641
37.6.9	mcg_dmx32_t . . . . .	641
37.6.10	mcg_drs_t . . . . .	642
37.6.11	mcg_pll_ref_src_t . . . . .	642
37.6.12	mcg_clkout_src_t . . . . .	642
37.6.13	mcg_atm_select_t . . . . .	642
37.6.14	mcg_oscsel_t . . . . .	642
37.6.15	mcg_pll_clk_select_t . . . . .	643
37.6.16	mcg_monitor_mode_t . . . . .	643
37.6.17	_mcg_status . . . . .	643
37.6.18	_mcg_status_flags_t . . . . .	643
37.6.19	_mcg_irclk_enable_mode . . . . .	643
37.6.20	_mcg_pll_enable_mode . . . . .	644
37.6.21	mcg_mode_t . . . . .	644

# Contents

Section Number	Title	Page Number
<b>37.7</b>	<b>Function Documentation</b>	<b>644</b>
37.7.1	CLOCK_EnableClock . . . . .	644
37.7.2	CLOCK_DisableClock . . . . .	644
37.7.3	CLOCK_SetSdhc0Clock . . . . .	645
37.7.4	CLOCK_SetEr32kClock . . . . .	645
37.7.5	CLOCK_SetTraceClock . . . . .	645
37.7.6	CLOCK_SetPllFllSelClock . . . . .	645
37.7.7	CLOCK_SetClkOutClock . . . . .	645
37.7.8	CLOCK_SetRtcClkOutClock . . . . .	646
37.7.9	CLOCK_EnableUsbfs0Clock . . . . .	646
37.7.10	CLOCK_DisableUsbfs0Clock . . . . .	646
37.7.11	CLOCK_SetOutDiv . . . . .	646
37.7.12	CLOCK_GetFreq . . . . .	647
37.7.13	CLOCK_GetCoreSysClkFreq . . . . .	647
37.7.14	CLOCK_GetPlatClkFreq . . . . .	647
37.7.15	CLOCK_GetBusClkFreq . . . . .	647
37.7.16	CLOCK_GetFlexBusClkFreq . . . . .	648
37.7.17	CLOCK_GetFlashClkFreq . . . . .	648
37.7.18	CLOCK_GetPllFllSelClkFreq . . . . .	648
37.7.19	CLOCK_GetEr32kClkFreq . . . . .	648
37.7.20	CLOCK_GetOsc0ErClkFreq . . . . .	648
37.7.21	CLOCK_SetSimConfig . . . . .	648
37.7.22	CLOCK_SetSimSafeDivs . . . . .	649
37.7.23	CLOCK_GetOutClkFreq . . . . .	649
37.7.24	CLOCK_GetFllFreq . . . . .	649
37.7.25	CLOCK_GetInternalRefClkFreq . . . . .	649
37.7.26	CLOCK_GetFixedFreqClkFreq . . . . .	650
37.7.27	CLOCK_GetPll0Freq . . . . .	650
37.7.28	CLOCK_SetLowPowerEnable . . . . .	650
37.7.29	CLOCK_SetInternalRefClkConfig . . . . .	650
37.7.30	CLOCK_SetExternalRefClkConfig . . . . .	651
37.7.31	CLOCK_SetFllExtRefDiv . . . . .	651
37.7.32	CLOCK_EnablePll0 . . . . .	652
37.7.33	CLOCK_DisablePll0 . . . . .	652
37.7.34	CLOCK_CalcPllDiv . . . . .	652
37.7.35	CLOCK_SetOsc0MonitorMode . . . . .	652
37.7.36	CLOCK_SetRtcOscMonitorMode . . . . .	653
37.7.37	CLOCK_SetPll0MonitorMode . . . . .	653
37.7.38	CLOCK_GetStatusFlags . . . . .	653
37.7.39	CLOCK_ClearStatusFlags . . . . .	654
37.7.40	OSC_SetExtRefClkConfig . . . . .	654
37.7.41	OSC_SetCapLoad . . . . .	654
37.7.42	CLOCK_InitOsc0 . . . . .	655
37.7.43	CLOCK_DeinitOsc0 . . . . .	655
37.7.44	CLOCK_SetXtal0Freq . . . . .	655

# Contents

<b>Section Number</b>	<b>Title</b>	<b>Page Number</b>
37.7.45	CLOCK_SetXtal32Freq . . . . .	655
37.7.46	CLOCK_TrimInternalRefClk . . . . .	655
37.7.47	CLOCK_GetMode . . . . .	657
37.7.48	CLOCK_SetFeiMode . . . . .	657
37.7.49	CLOCK_SetFeeMode . . . . .	658
37.7.50	CLOCK_SetFbiMode . . . . .	658
37.7.51	CLOCK_SetFbeMode . . . . .	659
37.7.52	CLOCK_SetBlpiMode . . . . .	660
37.7.53	CLOCK_SetBlpeMode . . . . .	660
37.7.54	CLOCK_SetPbeMode . . . . .	660
37.7.55	CLOCK_SetPeeMode . . . . .	661
37.7.56	CLOCK_ExternalModeToFbeModeQuick . . . . .	661
37.7.57	CLOCK_InternalModeToFbiModeQuick . . . . .	662
37.7.58	CLOCK_BootToFeiMode . . . . .	662
37.7.59	CLOCK_BootToFeeMode . . . . .	663
37.7.60	CLOCK_BootToBlpiMode . . . . .	663
37.7.61	CLOCK_BootToBlpeMode . . . . .	664
37.7.62	CLOCK_BootToPeeMode . . . . .	664
37.7.63	CLOCK_SetMcgConfig . . . . .	665
<b>37.8</b>	<b>Variable Documentation</b> . . . . .	<b>665</b>
37.8.1	g_xtal0Freq . . . . .	665
37.8.2	g_xtal32Freq . . . . .	666
<b>37.9</b>	<b>Multipurpose Clock Generator (MCG)</b> . . . . .	<b>667</b>
37.9.1	Function description . . . . .	667
37.9.2	Typical use case . . . . .	669
37.9.3	Code Configuration Option . . . . .	672
<b>Chapter</b>	<b>DMA Manager</b>	
<b>38.1</b>	<b>Overview</b> . . . . .	<b>675</b>
<b>38.2</b>	<b>Function groups</b> . . . . .	<b>675</b>
38.2.1	DMAMGR Initialization and De-initialization . . . . .	675
38.2.2	DMAMGR Operation . . . . .	675
<b>38.3</b>	<b>Typical use case</b> . . . . .	<b>675</b>
38.3.1	DMAMGR static channel allocation . . . . .	675
38.3.2	DMAMGR dynamic channel allocation . . . . .	675
<b>38.4</b>	<b>Data Structure Documentation</b> . . . . .	<b>676</b>
38.4.1	struct dmamanager_handle_t . . . . .	676
<b>38.5</b>	<b>Macro Definition Documentation</b> . . . . .	<b>677</b>
38.5.1	DMAMGR_DYNAMIC_ALLOCATE . . . . .	677

# Contents

Section Number	Title	Page Number
<b>38.6</b>	<b>Enumeration Type Documentation</b>	<b>677</b>
38.6.1	_dma_manager_status	677
<b>38.7</b>	<b>Function Documentation</b>	<b>677</b>
38.7.1	DMAMGR_Init	677
38.7.2	DMAMGR_Deinit	678
38.7.3	DMAMGR_RequestChannel	678
38.7.4	DMAMGR_ReleaseChannel	679
38.7.5	DMAMGR_IsChannelOccupied	680
<b>Chapter</b>	<b>Memory-Mapped Cryptographic Acceleration Unit (MMCAU)</b>	
<b>39.1</b>	<b>Overview</b>	<b>681</b>
<b>39.2</b>	<b>Purpose</b>	<b>681</b>
<b>39.3</b>	<b>Library Features</b>	<b>681</b>
<b>39.4</b>	<b>CAU and mmCAU software library overview</b>	<b>681</b>
<b>39.5</b>	<b>mmCAU software library usage</b>	<b>682</b>
<b>39.6</b>	<b>Function Documentation</b>	<b>684</b>
39.6.1	cau_aes_set_key	684
39.6.2	cau_aes_encrypt	685
39.6.3	cau_aes_decrypt	685
39.6.4	cau_des_chk_parity	686
39.6.5	cau_des_encrypt	686
39.6.6	cau_des_decrypt	687
39.6.7	cau_md5_initialize_output	687
39.6.8	cau_md5_hash_n	688
39.6.9	cau_md5_update	689
39.6.10	cau_md5_hash	689
39.6.11	cau_sha1_initialize_output	690
39.6.12	cau_sha1_hash_n	690
39.6.13	cau_sha1_update	691
39.6.14	cau_sha1_hash	692
39.6.15	cau_sha256_initialize_output	692
39.6.16	cau_sha256_hash_n	693
39.6.17	cau_sha256_update	693
39.6.18	cau_sha256_hash	694
39.6.19	MMCAU_AES_SetKey	694
39.6.20	MMCAU_AES_EncryptEcb	695
39.6.21	MMCAU_AES_DecryptEcb	696
39.6.22	MMCAU_DES_ChkParity	696
39.6.23	MMCAU_DES_EncryptEcb	697

# Contents

Section Number	Title	Page Number
39.6.24	MMCAU_DES_DecryptEcb . . . . .	697
39.6.25	MMCAU_MD5_InitializeOutput . . . . .	698
39.6.26	MMCAU_MD5_HashN . . . . .	698
39.6.27	MMCAU_MD5_Update . . . . .	699
39.6.28	MMCAU_SHA1_InitializeOutput . . . . .	700
39.6.29	MMCAU_SHA1_HashN . . . . .	700
39.6.30	MMCAU_SHA1_Update . . . . .	701
39.6.31	MMCAU_SHA256_InitializeOutput . . . . .	702
39.6.32	MMCAU_SHA256_HashN . . . . .	702
39.6.33	MMCAU_SHA256_Update . . . . .	703

## **Chapter Secure Digital Card/Embedded MultiMedia Card (CARD)**

<b>40.1</b>	<b>Overview . . . . .</b>	<b>705</b>
<b>40.2</b>	<b>Data Structure Documentation . . . . .</b>	<b>710</b>
40.2.1	struct sd_card_t . . . . .	710
40.2.2	struct sdio_card_t . . . . .	711
40.2.3	struct mmc_card_t . . . . .	712
40.2.4	struct mmc_boot_config_t . . . . .	713
<b>40.3</b>	<b>Macro Definition Documentation . . . . .</b>	<b>713</b>
40.3.1	FSL_SDMMC_DRIVER_VERSION . . . . .	713
<b>40.4</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>714</b>
40.4.1	_sdmmc_status . . . . .	714
40.4.2	_sd_card_flag . . . . .	715
40.4.3	_mmc_card_flag . . . . .	715
40.4.4	card_operation_voltage_t . . . . .	715
40.4.5	_host_endian_mode . . . . .	716
<b>40.5</b>	<b>Function Documentation . . . . .</b>	<b>716</b>
40.5.1	SD_Init . . . . .	716
40.5.2	SD_Deinit . . . . .	717
40.5.3	SD_CheckReadOnly . . . . .	717
40.5.4	SD_ReadBlocks . . . . .	717
40.5.5	SD_WriteBlocks . . . . .	718
40.5.6	SD_EraseBlocks . . . . .	719
40.5.7	MMC_Init . . . . .	720
40.5.8	MMC_Deinit . . . . .	721
40.5.9	MMC_CheckReadOnly . . . . .	721
40.5.10	MMC_ReadBlocks . . . . .	721
40.5.11	MMC_WriteBlocks . . . . .	722
40.5.12	MMC_EraseGroups . . . . .	723
40.5.13	MMC_SelectPartition . . . . .	723

# Contents

Section Number	Title	Page Number
40.5.14	MMC_SetBootConfig . . . . .	724
40.5.15	SDIO_CardInActive . . . . .	724
40.5.16	SDIO_IO_Write_Direct . . . . .	725
40.5.17	SDIO_IO_Read_Direct . . . . .	725
40.5.18	SDIO_IO_Write_Extended . . . . .	726
40.5.19	SDIO_IO_Read_Extended . . . . .	726
40.5.20	SDIO_GetCardCapability . . . . .	727
40.5.21	SDIO_SetBlockSize . . . . .	727
40.5.22	SDIO_CardReset . . . . .	728
40.5.23	SDIO_SetDataBusWidth . . . . .	728
40.5.24	SDIO_SwitchToHighSpeed . . . . .	729
40.5.25	SDIO_ReadCIS . . . . .	729
40.5.26	SDIO_Init . . . . .	730
40.5.27	SDIO_EnableIOInterrupt . . . . .	731
40.5.28	SDIO_EnableIO . . . . .	731
40.5.29	SDIO_SelectIO . . . . .	731
40.5.30	SDIO_AbortIO . . . . .	732
40.5.31	SDIO_DeInit . . . . .	732
40.5.32	HOST_NotSupport . . . . .	732
40.5.33	CardInsertDetect . . . . .	733
40.5.34	HOST_Init . . . . .	733
40.5.35	HOST_Deinit . . . . .	733

## Chapter **SPI based Secure Digital Card (SDSPI)**

<b>41.1</b>	<b>Overview . . . . .</b>	<b>735</b>
<b>41.2</b>	<b>Data Structure Documentation . . . . .</b>	<b>737</b>
41.2.1	struct sdspi_command_t . . . . .	737
41.2.2	struct sdspi_host_t . . . . .	737
41.2.3	struct sdspi_card_t . . . . .	737
<b>41.3</b>	<b>Enumeration Type Documentation . . . . .</b>	<b>738</b>
41.3.1	_sdspi_status . . . . .	738
41.3.2	_sdspi_card_flag . . . . .	739
41.3.3	sdspi_response_type_t . . . . .	739
<b>41.4</b>	<b>Function Documentation . . . . .</b>	<b>739</b>
41.4.1	SDSPI_Init . . . . .	739
41.4.2	SDSPI_Deinit . . . . .	740
41.4.3	SDSPI_CheckReadOnly . . . . .	740
41.4.4	SDSPI_ReadBlocks . . . . .	741
41.4.5	SDSPI_WriteBlocks . . . . .	741

# Contents

Section Number	Title	Page Number
Chapter	<b>Debug Console</b>	
42.1	<b>Overview</b> . . . . .	743
42.2	<b>Function groups</b> . . . . .	743
42.2.1	Initialization . . . . .	743
42.2.2	Advanced Feature . . . . .	744
42.3	<b>Typical use case</b> . . . . .	747
42.4	<b>Semihosting</b> . . . . .	749
42.4.1	Guide Semihosting for IAR . . . . .	749
42.4.2	Guide Semihosting for Keil µVision . . . . .	749
42.4.3	Guide Semihosting for KDS . . . . .	751
42.4.4	Guide Semihosting for ATL . . . . .	751
42.4.5	Guide Semihosting for ARMGCC . . . . .	752
Chapter	<b>Notification Framework</b>	
43.1	<b>Overview</b> . . . . .	755
43.2	<b>Notifier Overview</b> . . . . .	755
43.3	<b>Data Structure Documentation</b> . . . . .	757
43.3.1	struct notifier_notification_block_t . . . . .	757
43.3.2	struct notifier_callback_config_t . . . . .	758
43.3.3	struct notifier_handle_t . . . . .	758
43.4	<b>Typedef Documentation</b> . . . . .	759
43.4.1	notifier_user_config_t . . . . .	759
43.4.2	notifier_user_function_t . . . . .	759
43.4.3	notifier_callback_t . . . . .	760
43.5	<b>Enumeration Type Documentation</b> . . . . .	760
43.5.1	_notifier_status . . . . .	760
43.5.2	notifier_policy_t . . . . .	761
43.5.3	notifier_notification_type_t . . . . .	761
43.5.4	notifier_callback_type_t . . . . .	761
43.6	<b>Function Documentation</b> . . . . .	762
43.6.1	NOTIFIER_CreateHandle . . . . .	762
43.6.2	NOTIFIER_SwitchConfig . . . . .	763
43.6.3	NOTIFIER_GetErrorCallbackIndex . . . . .	764

# Contents

Section Number	Title	Page Number
<b>Chapter</b>	<b>Shell</b>	
<b>44.1</b>	<b>Overview</b>	<b>765</b>
<b>44.2</b>	<b>Function groups</b>	<b>765</b>
44.2.1	Initialization	765
44.2.2	Advanced Feature	765
44.2.3	Shell Operation	766
<b>44.3</b>	<b>Data Structure Documentation</b>	<b>767</b>
44.3.1	struct shell_context_struct	767
44.3.2	struct shell_command_context_t	768
44.3.3	struct shell_command_context_list_t	768
<b>44.4</b>	<b>Macro Definition Documentation</b>	<b>769</b>
44.4.1	SHELL_USE_HISTORY	769
44.4.2	SHELL_SEARCH_IN_HIST	769
44.4.3	SHELL_USE_FILE_STREAM	769
44.4.4	SHELL_AUTO_COMPLETE	769
44.4.5	SHELL_BUFFER_SIZE	769
44.4.6	SHELL_MAX_ARGS	769
44.4.7	SHELL_HIST_MAX	769
44.4.8	SHELL_MAX_CMD	769
<b>44.5</b>	<b>Typedef Documentation</b>	<b>769</b>
44.5.1	send_data_cb_t	769
44.5.2	recv_data_cb_t	769
44.5.3	printf_data_t	769
44.5.4	cmd_function_t	769
<b>44.6</b>	<b>Enumeration Type Documentation</b>	<b>769</b>
44.6.1	fun_key_status_t	769
<b>44.7</b>	<b>Function Documentation</b>	<b>770</b>
44.7.1	SHELL_Init	770
44.7.2	SHELL_RegisterCommand	770
44.7.3	SHELL_Main	770
<b>Chapter</b>	<b>Smartcard_phy_ncn8025_driver</b>	
<b>45.1</b>	<b>Overview</b>	<b>773</b>
<b>45.2</b>	<b>Macro Definition Documentation</b>	<b>774</b>
45.2.1	SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT	774
45.2.2	SMARTCARD_NCN8025_STATUS_PRES	774

## Contents

Section Number	Title	Page Number
<b>45.3</b>	<b>Function Documentation</b>	<b>774</b>
45.3.1	SMARTCARD_PHY_NCN8025_GetDefaultConfig . . . . .	774
45.3.2	SMARTCARD_PHY_NCN8025_Init . . . . .	774
45.3.3	SMARTCARD_PHY_NCN8025_Deinit . . . . .	774
45.3.4	SMARTCARD_PHY_NCN8025_Activate . . . . .	775
45.3.5	SMARTCARD_PHY_NCN8025_Deactivate . . . . .	775
45.3.6	SMARTCARD_PHY_NCN8025_Control . . . . .	775
45.3.7	SMARTCARD_PHY_NCN8025_IRQHandler . . . . .	776

# Chapter 1

## Introduction

The MCUXpresso Software Development Kit (MCUXpresso SDK) is a collection of software enablement for NXP Microcontrollers that includes peripheral drivers, multicore support, USB stack, and integrated RTOS support for FreeRTOS<sup>TM</sup>. In addition to the base enablement, the MCUXpresso SDK is augmented with demo applications, driver example projects, and API documentation to help users quickly leverage the support provided by MCUXpresso SDK. The KEx Web UI is available to provide access to all MCUXpresso SDK packages. See the *MCUXpresso Software Development Kit (SDK) Release Notes* (document MCUXSDKRNN) in the Supported Devices section at [MCUXpresso-SDK: Software Development Kit for MCUXpresso](#) for details.

The MCUXpresso SDK is built with the following runtime software components:

- ARM<sup>®</sup> and DSP standard libraries, and CMSIS-compliant device header files which provide direct access to the peripheral registers.
- Peripheral drivers that provide stateless, high-performance, ease-of-use APIs. Communication drivers provide higher-level transactional APIs for a higher-performance option.
- RTOS wrapper driver built on top of MCUXpresso SDK peripheral drivers and leverage native RTOS services to better comply to the RTOS cases.
- Real time operation systems (RTOS) for FreeRTOS OS.
- Stacks and middleware in source or object formats including:
  - A USB device, host, and OTG stack with comprehensive USB class support.
  - CMSIS-DSP, a suite of common signal processing functions.
  - The MCUXpresso SDK comes complete with software examples demonstrating the usage of the peripheral drivers, RTOS wrapper drivers, middleware, and RTOSes.

All demo applications and driver examples are provided with projects for the following toolchains:

- IAR Embedded Workbench
- Keil MDK
- MCUXpresso IDE

The peripheral drivers and RTOS driver wrappers can be used across multiple devices within the product family without modification. The configuration items for each driver are encapsulated into C language data structures. Device-specific configuration information is provided as part of the MCUXpresso SDK and need not be modified by the user. If necessary, the user is able to modify the peripheral driver and RTOS wrapper driver configuration during runtime. The driver examples demonstrate how to configure the drivers by passing the proper configuration data to the APIs. The folder structure is organized to reduce the total number of includes required to compile a project.

The rest of this document describes the API references in detail for the peripheral drivers and RTOS wrapper drivers. For the latest version of this and other MCUXpresso SDK documents, see the [kex.-nxp.com/apidoc](#).

<b>Deliverable</b>	<b>Location</b>
Demo Applications	<install_dir>/boards/<board_name>/demo_apps
Driver Examples	<install_dir>/boards/<board_name>/driver_examples
Documentation	<install_dir>/docs
Middleware	<install_dir>/middleware
Drivers	<install_dir>/<device_name>/drivers/
CMSIS Standard ARM Cortex-M Headers, math and DSP Libraries	<install_dir>/CMSIS
Device Startup and Linker	<install_dir>/<device_name>/<toolchain>/
MCUXpresso SDK Utilities	<install_dir>/devices/<device_name>/utilities
RTOS Kernel Code	<install_dir>/rtos

Table 2: MCUXpresso SDK Folder Structure

## Chapter 2

### Driver errors status

- `kStatus_DSPI_Error` = 601
- `kStatus_EDMA_QueueFull` = 5100
- `kStatus_EDMA_Busy` = 5101
- `kStatus_SAI_TxBusy` = 1900
- `kStatus_SAI_RxBusy` = 1901
- `kStatus_SAI_TxError` = 1902
- `kStatus_SAI_RxError` = 1903
- `kStatus_SAI_QueueFull` = 1904
- `kStatus_SAI_TxIdle` = 1905
- `kStatus_SAI_RxIdle` = 1906
- `kStatus_SMARTCARD_Success` = 4300
- `kStatus_SMARTCARD_TxBusy` = 4301
- `kStatus_SMARTCARD_RxBusy` = 4302
- `kStatus_SMARTCARD_NoTransferInProgress` = 4303
- `kStatus_SMARTCARD_Timeout` = 4304
- `kStatus_SMARTCARD_Initialized` = 4305
- `kStatus_SMARTCARD_PhysicalInitialized` = 4306
- `kStatus_SMARTCARD_CardNotActivated` = 4307
- `kStatus_SMARTCARD_InvalidInput` = 4308
- `kStatus_SMARTCARD_OtherError` = 4309
- `kStatus_SMC_StopAbort` = 3900
- `kStatus_DMAMGR_ChannelOccupied` = 5200
- `kStatus_DMAMGR_ChannelNotUsed` = 5201
- `kStatus_DMAMGR_NoFreeChannel` = 5202
- `kStatus_NOTIFIER_ErrorNotificationBefore` = 9800
- `kStatus_NOTIFIER_ErrorNotificationAfter` = 9801



# Chapter 3

## Architectural Overview

This chapter provides the architectural overview for the MCUXpresso Software Development Kit (MCUXpresso SDK). It describes each layer within the architecture and its associated components.

### Overview

The MCUXpresso SDK architecture consists of five key components listed below.

1. The ARM Cortex Microcontroller Software Interface Standard (CMSIS) CORE compliance device-specific header files, SOC Header, and CMSIS math/DSP libraries.
2. Peripheral Drivers
3. Real-time Operating Systems (RTOS)
4. Stacks and Middleware that integrate with the MCUXpresso SDK
5. Demo Applications based on the MCUXpresso SDK

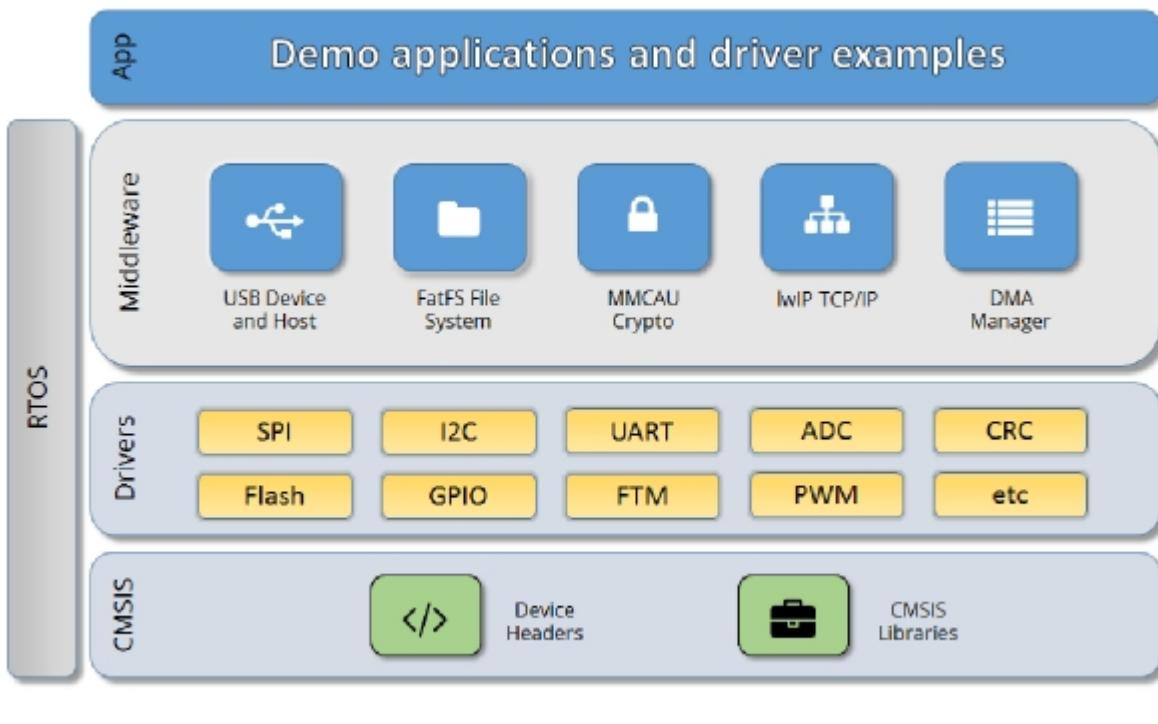


Figure 1: MCUXpresso SDK Block Diagram

### MCU header files

Each supported MCU device in the MCUXpresso SDK has an overall System-on Chip (SoC) memory-

mapped header file. This header file contains the memory map and register base address for each peripheral and the IRQ vector table with associated vector numbers. The overall SoC header file provides access to the peripheral registers through pointers and predefined bit masks. In addition to the overall SoC memory-mapped header file, the MCUXpresso SDK includes a feature header file for each device. The feature header file allows NXP to deliver a single software driver for a given peripheral. The feature file ensures that the driver is properly compiled for the target SOC.

## CMSIS Support

Along with the SoC header files and peripheral extension header files, the MCUXpresso SDK also includes common CMSIS header files for the ARM Cortex-M core and the math and DSP libraries from the latest CMSIS release. The CMSIS DSP library source code is also included for reference.

## MCUXpresso SDK Peripheral Drivers

The MCUXpresso SDK peripheral drivers mainly consist of low-level functional APIs for the MCU product family on-chip peripherals and also of high-level transactional APIs for some bus drivers/DM-A driver/eDMA driver to quickly enable the peripherals and perform transfers.

All MCUXpresso SDK peripheral drivers only depend on the CMSIS headers, device feature files, fsl\_common.h, and fsl\_clock.h files so that users can easily pull selected drivers and their dependencies into projects. With the exception of the clock/power-relevant peripherals, each peripheral has its own driver. Peripheral drivers handle the peripheral clock gating/ungating inside the drivers during initialization and deinitialization respectively.

Low-level functional APIs provide common peripheral functionality, abstracting the hardware peripheral register accesses into a set of stateless basic functional operations. These APIs primarily focus on the control, configuration, and function of basic peripheral operations. The APIs hide the register access details and various MCU peripheral instantiation differences so that the application can be abstracted from the low-level hardware details. The API prototypes are intentionally similar to help ensure easy portability across supported MCUXpresso SDK devices.

Transactional APIs provide a quick method for customers to utilize higher-level functionality of the peripherals. The transactional APIs utilize interrupts and perform asynchronous operations without user intervention. Transactional APIs operate on high-level logic that requires data storage for internal operation context handling. However, the Peripheral Drivers do not allocate this memory space. Rather, the user passes in the memory to the driver for internal driver operation. Transactional APIs ensure the NVIC is enabled properly inside the drivers. The transactional APIs do not meet all customer needs, but provide a baseline for development of custom user APIs.

Note that the transactional drivers never disable an NVIC after use. This is due to the shared nature of interrupt vectors on devices. It is up to the user to ensure that NVIC interrupts are properly disabled after usage is complete.

## Interrupt handling for transactional APIs

A double weak mechanism is introduced for drivers with transactional API. The double weak indicates two levels of weak vector entries. See the examples below:

```
PUBWEAK SPI0_IRQHandler  
PUBWEAK SPI0_DriverIRQHandler  
SPI0_IRQHandler
```

```
LDR      R0, =SPI0_DriverIRQHandler  
BX      R0
```

The first level of the weak implementation are the functions defined in the vector table. In the devices/<DEVICE\_NAME>/<TOOLCHAIN>/startup\_<DEVICE\_NAME>.S file, the implementation of the first layer weak function calls the second layer of weak function. The implementation of the second layer weak function (ex. SPI0\_DriverIRQHandler) jumps to itself (BX). The MCUXpresso SDK drivers with transactional APIs provide the reimplementation of the second layer function inside of the peripheral driver. If the MCUXpresso SDK drivers with transactional APIs are linked into the image, the SPI0\_DriverIRQHandler is replaced with the function implemented in the MCUXpresso SDK SPI driver.

The reason for implementing the double weak functions is to provide a better user experience when using the transactional APIs. For drivers with a transactional function, call the transactional APIs and the drivers complete the interrupt-driven flow. Users are not required to redefine the vector entries out of the box. At the same time, if users are not satisfied by the second layer weak function implemented in the MCUXpresso SDK drivers, users can redefine the first layer weak function and implement their own interrupt handler functions to suit their implementation.

The limitation of the double weak mechanism is that it cannot be used for peripherals that share the same vector entry. For this use case, redefine the first layer weak function to enable the desired peripheral interrupt functionality. For example, if the MCU's UART0 and UART1 share the same vector entry, redefine the UART0\_UART1\_IRQHandler according to the use case requirements.

## Feature Header Files

The peripheral drivers are designed to be reusable regardless of the peripheral functional differences from one MCU device to another. An overall Peripheral Feature Header File is provided for the MCUXpresso SDK-supported MCU device to define the features or configuration differences for each sub-family device.

## Application

See the *Getting Started with MCUXpresso SDK* document (MCUXSDKGSUG).



## **Chapter 4**

## **Trademarks**

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.

How to Reach Us:

Home Page: [nxp.com](http://nxp.com)

Web Support: [nxp.com/support](http://nxp.com/support)

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/Sales-TermsandConditions](http://nxp.com/Sales-TermsandConditions)

NXP, the NXP logo, Freescale, the Freescale logo, Kinetis, and Processor Expert are trademarks of NXP B.V. Tower is a trademark of NXP B.V. All other product or service names are the property of their respective owners. ARM, ARM powered logo, Keil, and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2017 NXP B.V.



# Chapter 5

## ADC16: 16-bit SAR Analog-to-Digital Converter Driver

### 5.1 Overview

The MCUXpresso SDK provides a peripheral driver for the 16-bit SAR Analog-to-Digital Converter (ADC16) module of MCUXpresso SDK devices.

### 5.2 Typical use case

#### 5.2.1 Polling Configuration

```
adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

ADC16_Init(DEMO_ADC16_INSTANCE);
ADC16_SetDefaultConfig(&adc16ConfigStruct);
ADC16_Configure(DEMO_ADC16_INSTANCE, &adc16ConfigStruct);
ADC16_EnableHardwareTrigger(DEMO_ADC16_INSTANCE, false);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
    if (kStatus_Success == ADC16_DoAutoCalibration(DEMO_ADC16_INSTANCE))
    {
        PRINTF("ADC16_DoAutoCalibration() Done.\r\n");
    }
    else
    {
        PRINTF("ADC16_DoAutoCalibration() Failed.\r\n");
    }
#endif // FSL_FEATURE_ADC16_HAS_CALIBRATION

adc16ChannelConfigStruct.channelNumber = DEMO_ADC16_USER_CHANNEL;
adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
    false;
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif // FSL_FEATURE_ADC16_HAS_DIFF_MODE

while(1)
{
    GETCHAR(); // Input any key in terminal console.
    ADC16_ChannelConfigure(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP, &adc16ChannelConfigStruct);
    while (kADC16_ChannelConversionDoneFlag !=
    ADC16_ChannelGetStatusFlags(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP))
    {
    }
    PRINTF("ADC Value: %d\r\n", ADC16_ChannelGetConversionValue(DEMO_ADC16_INSTANCE,
    DEMO_ADC16_CHANNEL_GROUP));
}
```

#### 5.2.2 Interrupt Configuration

```
volatile bool g_Adcl6ConversionDoneFlag = false;
volatile uint32_t g_Adcl6ConversionValue;
volatile uint32_t g_Adcl6InterruptCount = 0U;
```

## Typical use case

```
// ...

adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

ADC16_Init(DEMO_ADC16_INSTANCE);
ADC16_GetDefaultConfig(&adc16ConfigStruct);
ADC16_Configure(DEMO_ADC16_INSTANCE, &adc16ConfigStruct);
ADC16_EnableHardwareTrigger(DEMO_ADC16_INSTANCE, false);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
if (ADC16_DoAutoCalibration(DEMO_ADC16_INSTANCE))
{
    PRINTF("ADC16_DoAutoCalibration() Done.\r\n");
}
else
{
    PRINTF("ADC16_DoAutoCalibration() Failed.\r\n");
}
#endif // FSL_FEATURE_ADC16_HAS_CALIBRATION

adc16ChannelConfigStruct.channelNumber = DEMO_ADC16_USER_CHANNEL;
adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
    true; // Enable the interrupt.
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif // FSL_FEATURE_ADC16_HAS_DIFF_MODE

while(1)
{
    GETCHAR(); // Input a key in the terminal console.
    g_Adc16ConversionDoneFlag = false;
    ADC16_ChannelConfigure(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP, &adc16ChannelConfigStruct);
    while (!g_Adc16ConversionDoneFlag)
    {
    }
    PRINTF("ADC Value: %d\r\n", g_Adc16ConversionValue);
    PRINTF("ADC Interrupt Count: %d\r\n", g_Adc16InterruptCount);
}

// ...

void DEMO_ADC16_IRQHandler(void)
{
    g_Adc16ConversionDoneFlag = true;
    // Read the conversion result to clear the conversion completed flag.
    g_Adc16ConversionValue = ADC16_ChannelConversionValue(DEMO_ADC16_INSTANCE, DEMO_ADC16_CHANNEL_GROUP);
    g_Adc16InterruptCount++;
}
```

## Data Structures

- struct `adc16_config_t`  
*ADC16 converter configuration.* [More...](#)
- struct `adc16_hardware_compare_config_t`  
*ADC16 Hardware comparison configuration.* [More...](#)
- struct `adc16_channel_config_t`  
*ADC16 channel conversion configuration.* [More...](#)

## Enumerations

- enum `_adc16_channel_status_flags` { `kADC16_ChannelConversionDoneFlag` = ADC\_SC1\_COC\_O\_MASK }

- *Channel status flags.*
- enum `_adc16_status_flags` {
   
    `kADC16_ActiveFlag` = ADC\_SC2\_ADACT\_MASK,
   
    `kADC16_CalibrationFailedFlag` = ADC\_SC3\_CALF\_MASK }
- Converter status flags.*
- enum `adc16_channel_mux_mode_t` {
   
    `kADC16_ChannelMuxA` = 0U,
   
    `kADC16_ChannelMuxB` = 1U }
- Channel multiplexer mode for each channel.*
- enum `adc16_clock_divider_t` {
   
    `kADC16_ClockDivider1` = 0U,
   
    `kADC16_ClockDivider2` = 1U,
   
    `kADC16_ClockDivider4` = 2U,
   
    `kADC16_ClockDivider8` = 3U }
- Clock divider for the converter.*
- enum `adc16_resolution_t` {
   
    `kADC16_Resolution8or9Bit` = 0U,
   
    `kADC16_Resolution12or13Bit` = 1U,
   
    `kADC16_Resolution10or11Bit` = 2U,
   
    `kADC16_ResolutionSE8Bit` = `kADC16_Resolution8or9Bit`,
   
    `kADC16_ResolutionSE12Bit` = `kADC16_Resolution12or13Bit`,
   
    `kADC16_ResolutionSE10Bit` = `kADC16_Resolution10or11Bit`,
   
    `kADC16_ResolutionDF9Bit` = `kADC16_Resolution8or9Bit`,
   
    `kADC16_ResolutionDF13Bit` = `kADC16_Resolution12or13Bit`,
   
    `kADC16_ResolutionDF11Bit` = `kADC16_Resolution10or11Bit`,
   
    `kADC16_Resolution16Bit` = 3U,
   
    `kADC16_ResolutionSE16Bit` = `kADC16_Resolution16Bit`,
   
    `kADC16_ResolutionDF16Bit` = `kADC16_Resolution16Bit` }
- Converter's resolution.*
- enum `adc16_clock_source_t` {
   
    `kADC16_ClockSourceAlt0` = 0U,
   
    `kADC16_ClockSourceAlt1` = 1U,
   
    `kADC16_ClockSourceAlt2` = 2U,
   
    `kADC16_ClockSourceAlt3` = 3U,
   
    `kADC16_ClockSourceAsynchronousClock` = `kADC16_ClockSourceAlt3` }
- Clock source.*
- enum `adc16_long_sample_mode_t` {
   
    `kADC16_LongSampleCycle24` = 0U,
   
    `kADC16_LongSampleCycle16` = 1U,
   
    `kADC16_LongSampleCycle10` = 2U,
   
    `kADC16_LongSampleCycle6` = 3U,
   
    `kADC16_LongSampleDisabled` = 4U }
- Long sample mode.*
- enum `adc16_reference_voltage_source_t` {
   
    `kADC16_ReferenceVoltageSourceVref` = 0U,
   
    `kADC16_ReferenceVoltageSourceValt` = 1U }

## Typical use case

- Reference voltage source.
  - enum `adc16_hardware_average_mode_t` {  
    kADC16\_HardwareAverageCount4 = 0U,  
    kADC16\_HardwareAverageCount8 = 1U,  
    kADC16\_HardwareAverageCount16 = 2U,  
    kADC16\_HardwareAverageCount32 = 3U,  
    kADC16\_HardwareAverageDisabled = 4U }
- Hardware average mode.
  - enum `adc16_hardware_compare_mode_t` {  
    kADC16\_HardwareCompareMode0 = 0U,  
    kADC16\_HardwareCompareMode1 = 1U,  
    kADC16\_HardwareCompareMode2 = 2U,  
    kADC16\_HardwareCompareMode3 = 3U }
- Hardware compare mode.

## Driver version

- #define `FSL_ADC16_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
*ADC16 driver version 2.0.0.*

## Initialization

- void `ADC16_Init` (ADC\_Type \*base, const `adc16_config_t` \*config)  
*Initializes the ADC16 module.*
- void `ADC16_Deinit` (ADC\_Type \*base)  
*De-initializes the ADC16 module.*
- void `ADC16_GetDefaultConfig` (`adc16_config_t` \*config)  
*Gets an available pre-defined settings for the converter's configuration.*
- status\_t `ADC16_DoAutoCalibration` (ADC\_Type \*base)  
*Automates the hardware calibration.*
- static void `ADC16_SetOffsetValue` (ADC\_Type \*base, int16\_t value)  
*Sets the offset value for the conversion result.*

## Advanced Features

- static void `ADC16_EnableDMA` (ADC\_Type \*base, bool enable)  
*Enables generating the DMA trigger when the conversion is complete.*
- static void `ADC16_EnableHardwareTrigger` (ADC\_Type \*base, bool enable)  
*Enables the hardware trigger mode.*
- void `ADC16_SetChannelMuxMode` (ADC\_Type \*base, `adc16_channel_mux_mode_t` mode)  
*Sets the channel mux mode.*
- void `ADC16_SetHardwareCompareConfig` (ADC\_Type \*base, const `adc16_hardware_compare_config_t` \*config)  
*Configures the hardware compare mode.*
- void `ADC16_SetHardwareAverage` (ADC\_Type \*base, `adc16_hardware_average_mode_t` mode)  
*Sets the hardware average mode.*
- uint32\_t `ADC16_GetStatusFlags` (ADC\_Type \*base)  
*Gets the status flags of the converter.*
- void `ADC16_ClearStatusFlags` (ADC\_Type \*base, uint32\_t mask)  
*Clears the status flags of the converter.*

## Conversion Channel

- void [ADC16\\_SetChannelConfig](#) (ADC\_Type \*base, uint32\_t channelGroup, const [adc16\\_channel\\_config\\_t](#) \*config)  
*Configures the conversion channel.*
- static uint32\_t [ADC16\\_GetChannelConversionValue](#) (ADC\_Type \*base, uint32\_t channelGroup)  
*Gets the conversion value.*
- uint32\_t [ADC16\\_GetChannelStatusFlags](#) (ADC\_Type \*base, uint32\_t channelGroup)  
*Gets the status flags of channel.*

## 5.3 Data Structure Documentation

### 5.3.1 struct adc16\_config\_t

#### Data Fields

- [adc16\\_reference\\_voltage\\_source\\_t](#) referenceVoltageSource  
*Select the reference voltage source.*
- [adc16\\_clock\\_source\\_t](#) clockSource  
*Select the input clock source to converter.*
- bool enableAsynchronousClock  
*Enable the asynchronous clock output.*
- [adc16\\_clock\\_divider\\_t](#) clockDivider  
*Select the divider of input clock source.*
- [adc16\\_resolution\\_t](#) resolution  
*Select the sample resolution mode.*
- [adc16\\_long\\_sample\\_mode\\_t](#) longSampleMode  
*Select the long sample mode.*
- bool enableHighSpeed  
*Enable the high-speed mode.*
- bool enableLowPower  
*Enable low power.*
- bool enableContinuousConversion  
*Enable continuous conversion mode.*

## Data Structure Documentation

### 5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 `adc16_reference_voltage_source_t adc16_config_t::referenceVoltageSource`

5.3.1.0.0.1.2 `adc16_clock_source_t adc16_config_t::clockSource`

5.3.1.0.0.1.3 `bool adc16_config_t::enableAsynchronousClock`

5.3.1.0.0.1.4 `adc16_clock_divider_t adc16_config_t::clockDivider`

5.3.1.0.0.1.5 `adc16_resolution_t adc16_config_t::resolution`

5.3.1.0.0.1.6 `adc16_long_sample_mode_t adc16_config_t::longSampleMode`

5.3.1.0.0.1.7 `bool adc16_config_t::enableHighSpeed`

5.3.1.0.0.1.8 `bool adc16_config_t::enableLowPower`

5.3.1.0.0.1.9 `bool adc16_config_t::enableContinuousConversion`

### 5.3.2 struct `adc16_hardware_compare_config_t`

#### Data Fields

- `adc16_hardware_compare_mode_t hardwareCompareMode`  
*Select the hardware compare mode.*
- `int16_t value1`  
*Setting value1 for hardware compare mode.*
- `int16_t value2`  
*Setting value2 for hardware compare mode.*

### 5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 `adc16_hardware_compare_mode_t adc16_hardware_compare_config_t::hardwareCompareMode`

See "adc16\_hardware\_compare\_mode\_t".

5.3.2.0.0.2.2 `int16_t adc16_hardware_compare_config_t::value1`

5.3.2.0.0.2.3 `int16_t adc16_hardware_compare_config_t::value2`

### 5.3.3 struct `adc16_channel_config_t`

#### Data Fields

- `uint32_t channelNumber`  
*Setting the conversion channel number.*
- `bool enableInterruptOnConversionCompleted`

- **bool enableDifferentialConversion**  
*Using Differential sample mode.*

### 5.3.3.0.0.3 Field Documentation

#### 5.3.3.0.0.3.1 `uint32_t adc16_channel_config_t::channelNumber`

The available range is 0-31. See channel connection information for each chip in Reference Manual document.

#### 5.3.3.0.0.3.2 `bool adc16_channel_config_t::enableInterruptOnConversionCompleted`

#### 5.3.3.0.0.3.3 `bool adc16_channel_config_t::enableDifferentialConversion`

## 5.4 Macro Definition Documentation

### 5.4.1 `#define FSL_ADC16_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

## 5.5 Enumeration Type Documentation

### 5.5.1 `enum _adc16_channel_status_flags`

Enumerator

***kADC16\_ChannelConversionDoneFlag*** Conversion done.

### 5.5.2 `enum _adc16_status_flags`

Enumerator

***kADC16\_ActiveFlag*** Converter is active.

***kADC16\_CalibrationFailedFlag*** Calibration is failed.

### 5.5.3 `enum adc16_channel_mux_mode_t`

For some ADC16 channels, there are two pin selections in channel multiplexer. For example, ADC0\_SE4a and ADC0\_SE4b are the different channels that share the same channel number.

Enumerator

***kADC16\_ChannelMuxA*** For channel with channel mux a.

***kADC16\_ChannelMuxB*** For channel with channel mux b.

## Enumeration Type Documentation

### 5.5.4 enum adc16\_clock\_divider\_t

Enumerator

- kADC16\_ClockDivider1* For divider 1 from the input clock to the module.
- kADC16\_ClockDivider2* For divider 2 from the input clock to the module.
- kADC16\_ClockDivider4* For divider 4 from the input clock to the module.
- kADC16\_ClockDivider8* For divider 8 from the input clock to the module.

### 5.5.5 enum adc16\_resolution\_t

Enumerator

- kADC16\_Resolution8or9Bit* Single End 8-bit or Differential Sample 9-bit.
- kADC16\_Resolution12or13Bit* Single End 12-bit or Differential Sample 13-bit.
- kADC16\_Resolution10or11Bit* Single End 10-bit or Differential Sample 11-bit.
- kADC16\_ResolutionSE8Bit* Single End 8-bit.
- kADC16\_ResolutionSE12Bit* Single End 12-bit.
- kADC16\_ResolutionSE10Bit* Single End 10-bit.
- kADC16\_ResolutionDF9Bit* Differential Sample 9-bit.
- kADC16\_ResolutionDF13Bit* Differential Sample 13-bit.
- kADC16\_ResolutionDF11Bit* Differential Sample 11-bit.
- kADC16\_Resolution16Bit* Single End 16-bit or Differential Sample 16-bit.
- kADC16\_ResolutionSE16Bit* Single End 16-bit.
- kADC16\_ResolutionDF16Bit* Differential Sample 16-bit.

### 5.5.6 enum adc16\_clock\_source\_t

Enumerator

- kADC16\_ClockSourceAlt0* Selection 0 of the clock source.
- kADC16\_ClockSourceAlt1* Selection 1 of the clock source.
- kADC16\_ClockSourceAlt2* Selection 2 of the clock source.
- kADC16\_ClockSourceAlt3* Selection 3 of the clock source.
- kADC16\_ClockSourceAsynchronousClock* Using internal asynchronous clock.

### 5.5.7 enum adc16\_long\_sample\_mode\_t

Enumerator

- kADC16\_LongSampleCycle24* 20 extra ADCK cycles, 24 ADCK cycles total.
- kADC16\_LongSampleCycle16* 12 extra ADCK cycles, 16 ADCK cycles total.

*kADC16\_LongSampleCycle10* 6 extra ADCK cycles, 10 ADCK cycles total.

*kADC16\_LongSampleCycle6* 2 extra ADCK cycles, 6 ADCK cycles total.

*kADC16\_LongSampleDisabled* Disable the long sample feature.

### 5.5.8 enum adc16\_reference\_voltage\_source\_t

Enumerator

*kADC16\_ReferenceVoltageSourceVref* For external pins pair of VrefH and VrefL.

*kADC16\_ReferenceVoltageSourceValt* For alternate reference pair of ValtH and ValtL.

### 5.5.9 enum adc16\_hardware\_average\_mode\_t

Enumerator

*kADC16\_HardwareAverageCount4* For hardware average with 4 samples.

*kADC16\_HardwareAverageCount8* For hardware average with 8 samples.

*kADC16\_HardwareAverageCount16* For hardware average with 16 samples.

*kADC16\_HardwareAverageCount32* For hardware average with 32 samples.

*kADC16\_HardwareAverageDisabled* Disable the hardware average feature.

### 5.5.10 enum adc16\_hardware\_compare\_mode\_t

Enumerator

*kADC16\_HardwareCompareMode0*  $x < \text{value1}$ .

*kADC16\_HardwareCompareMode1*  $x > \text{value1}$ .

*kADC16\_HardwareCompareMode2* if  $\text{value1} \leq \text{value2}$ , then  $x < \text{value1} \parallel x > \text{value2}$ ; else,  
 $\text{value1} > x > \text{value2}$ .

*kADC16\_HardwareCompareMode3* if  $\text{value1} \leq \text{value2}$ , then  $\text{value1} \leq x \leq \text{value2}$ ; else  $x \geq \text{value1} \parallel x \leq \text{value2}$ .

## 5.6 Function Documentation

### 5.6.1 void ADC16\_Init( ADC\_Type \* base, const adc16\_config\_t \* config )

## Function Documentation

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>config</i>	Pointer to configuration structure. See "adc16_config_t".

### 5.6.2 void ADC16\_Deinit ( ADC\_Type \* *base* )

Parameters

<i>base</i>	ADC16 peripheral base address.
-------------	--------------------------------

### 5.6.3 void ADC16\_GetDefaultConfig ( adc16\_config\_t \* *config* )

This function initializes the converter configuration structure with available settings. The default values are as follows.

```
* config->referenceVoltageSource      = kADC16_ReferenceVoltageSourceVref
*           ;
* config->clockSource               = kADC16_ClockSourceAsynchronousClock
*           ;
* config->enableAsynchronousClock   = true;
* config->clockDivider              = kADC16_ClockDivider8;
* config->resolution                = kADC16_ResolutionSE12Bit;
* config->longSampleMode            = kADC16_LongSampleDisabled;
* config->enableHighSpeed           = false;
* config->enableLowPower             = false;
* config->enableContinuousConversion = false;
*
```

Parameters

<i>config</i>	Pointer to the configuration structure.
---------------	---

### 5.6.4 status\_t ADC16\_DoAutoCalibration ( ADC\_Type \* *base* )

This auto calibration helps to adjust the plus/minus side gain automatically. Execute the calibration before using the converter. Note that the hardware trigger should be used during the calibration.

Parameters

<i>base</i>	ADC16 peripheral base address.
-------------	--------------------------------

Returns

Execution status.

Return values

<i>kStatus_Success</i>	Calibration is done successfully.
<i>kStatus_Fail</i>	Calibration has failed.

### 5.6.5 static void ADC16\_SetOffsetValue ( ADC\_Type \* *base*, int16\_t *value* ) [inline], [static]

This offset value takes effect on the conversion result. If the offset value is not zero, the reading result is subtracted by it. Note, the hardware calibration fills the offset value automatically.

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>value</i>	Setting offset value.

### 5.6.6 static void ADC16\_EnableDMA ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>enable</i>	Switcher of the DMA feature. "true" means enabled, "false" means not enabled.

### 5.6.7 static void ADC16\_EnableHardwareTrigger ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>enable</i>	Switcher of the hardware trigger feature. "true" means enabled, "false" means not enabled.

### 5.6.8 void ADC16\_SetChannelMuxMode ( ADC\_Type \* *base*, adc16\_channel\_mux\_mode\_t *mode* )

Some sample pins share the same channel index. The channel mux mode decides which pin is used for an indicated channel.

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>mode</i>	Setting channel mux mode. See "adc16_channel_mux_mode_t".

### 5.6.9 void ADC16\_SetHardwareCompareConfig ( ADC\_Type \* *base*, const adc16\_hardware\_compare\_config\_t \* *config* )

The hardware compare mode provides a way to process the conversion result automatically by using hardware. Only the result in the compare range is available. To compare the range, see "adc16\_hardware\_compare\_mode\_t" or the appropriate reference manual for more information.

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>config</i>	Pointer to the "adc16_hardware_compare_config_t" structure. Passing "NULL" disables the feature.

### 5.6.10 void ADC16\_SetHardwareAverage ( ADC\_Type \* *base*, adc16\_hardware\_average\_mode\_t *mode* )

The hardware average mode provides a way to process the conversion result automatically by using hardware. The multiple conversion results are accumulated and averaged internally making them easier to read.

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>mode</i>	Setting the hardware average mode. See "adc16_hardware_average_mode_t".

### 5.6.11 **uint32\_t ADC16\_GetStatusFlags ( ADC\_Type \* *base* )**

Parameters

<i>base</i>	ADC16 peripheral base address.
-------------	--------------------------------

Returns

Flags' mask if indicated flags are asserted. See "\_adc16\_status\_flags".

### 5.6.12 **void ADC16\_ClearStatusFlags ( ADC\_Type \* *base*, uint32\_t *mask* )**

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>mask</i>	Mask value for the cleared flags. See "_adc16_status_flags".

### 5.6.13 **void ADC16\_SetChannelConfig ( ADC\_Type \* *base*, uint32\_t *channelGroup*, const adc16\_channel\_config\_t \* *config* )**

This operation triggers the conversion when in software trigger mode. When in hardware trigger mode, this API configures the channel while the external trigger source helps to trigger the conversion.

Note that the "Channel Group" has a detailed description. To allow sequential conversions of the ADC to be triggered by internal peripherals, the ADC has more than one group of status and control registers, one for each conversion. The channel group parameter indicates which group of registers are used, for example, channel group 0 is for Group A registers and channel group 1 is for Group B registers. The channel groups are used in a "ping-pong" approach to control the ADC operation. At any point, only one of the channel groups is actively controlling ADC conversions. The channel group 0 is used for both software and hardware trigger modes. Channel group 1 and greater indicates multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the appropriate MCU reference manual for the number of SC1n registers (channel groups) specific to this device. Channel group 1 or greater are not used for software trigger operation. Therefore, writing to these channel groups does not initiate a new conversion. Updating the channel group 0 while a different channel group is

## Function Documentation

actively controlling a conversion is allowed and vice versa. Writing any of the channel group registers while that specific channel group is actively controlling a conversion aborts the current conversion.

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>channelGroup</i>	Channel group index.
<i>config</i>	Pointer to the "adc16_channel_config_t" structure for the conversion channel.

#### 5.6.14 static uint32\_t ADC16\_GetChannelConversionValue ( ADC\_Type \* *base*, uint32\_t *channelGroup* ) [inline], [static]

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>channelGroup</i>	Channel group index.

Returns

Conversion value.

#### 5.6.15 uint32\_t ADC16\_GetChannelStatusFlags ( ADC\_Type \* *base*, uint32\_t *channelGroup* )

Parameters

<i>base</i>	ADC16 peripheral base address.
<i>channelGroup</i>	Channel group index.

Returns

Flags' mask if indicated flags are asserted. See "\_adc16\_channel\_status\_flags".

## Function Documentation

# Chapter 6

## CMP: Analog Comparator Driver

### 6.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Analog Comparator (CMP) module of MCUXpresso SDK devices.

The CMP driver is a basic comparator with advanced features. The APIs for the basic comparator enable the CMP to compare the two voltages of the two input channels and create the output of the comparator result. The APIs for advanced features can be used as the plug-in functions based on the basic comparator. They can process the comparator's output with hardware support.

### 6.2 Typical use case

#### 6.2.1 Polling Configuration

```
int main(void)
{
    cmp_config_t mCmpConfigStruct;
    cmp_dac_config_t mCmpDacConfigStruct;

    // ...

    // Configures the comparator.
    CMP_Init(DEMO_CMP_INSTANCE);
    CMP_GetDefaultConfig(&mCmpConfigStruct);
    CMP_Configure(DEMO_CMP_INSTANCE, &mCmpConfigStruct);

    // Configures the DAC channel.
    mCmpDacConfigStruct.referenceVoltageSource =
        kCMP_VrefSourceVin2; // VCC.
    mCmpDacConfigStruct.DACValue = 32U; // Half voltage of logic high-level.
    CMP_SetDACConfig(DEMO_CMP_INSTANCE, &mCmpDacConfigStruct);
    CMP_SetInputChannels(DEMO_CMP_INSTANCE, DEMO_CMP_USER_CHANNEL, DEMO_CMP_DAC_CHANNEL
        );

    while (1)
    {
        if (0U != (kCMP_OutputAssertEventFlag &
        CMP_GetStatusFlags(DEMO_CMP_INSTANCE)))
        {
            // Do something.
        }
        else
        {
            // Do something.
        }
    }
}
```

## Typical use case

### 6.2.2 Interrupt Configuration

```
volatile uint32_t g_CmpFlags = 0U;  
// ...  
  
void DEMO_CMP_IRQ_HANDLER_FUNC(void)  
{  
    g_CmpFlags = CMP_GetStatusFlags(DEMO_CMP_INSTANCE);  
    CMP_ClearStatusFlags(DEMO_CMP_INSTANCE, kCMP_OutputRisingEventFlag |  
        kCMP_OutputFallingEventFlag);  
    if (0U != (g_CmpFlags & kCMP_OutputRisingEventFlag))  
    {  
        // Do something.  
    }  
    else if (0U != (g_CmpFlags & kCMP_OutputFallingEventFlag))  
    {  
        // Do something.  
    }  
}  
  
int main(void)  
{  
    cmp_config_t mCmpConfigStruct;  
    cmp_dac_config_t mCmpDacConfigStruct;  
  
    // ...  
    EnableIRQ(DEMO_CMP_IRQ_ID);  
    // ...  
  
    // Configures the comparator.  
    CMP_Init(DEMO_CMP_INSTANCE);  
    CMP_SetDefaultConfig(&mCmpConfigStruct);  
    CMP_Configure(DEMO_CMP_INSTANCE, &mCmpConfigStruct);  
  
    // Configures the DAC channel.  
    mCmpDacConfigStruct.referenceVoltageSource =  
        kCMP_VrefSourceVIn2; // VCC.  
    mCmpDacConfigStruct.DACValue = 32U; // Half voltage of logic high-level.  
    CMP_SetDACConfig(DEMO_CMP_INSTANCE, &mCmpDacConfigStruct);  
    CMP_SetInputChannels(DEMO_CMP_INSTANCE, DEMO_CMP_USER_CHANNEL, DEMO_CMP_DAC_CHANNEL  
        );  
  
    // Enables the output rising and falling interrupts.  
    CMP_EnableInterrupts(DEMO_CMP_INSTANCE,  
        kCMP_OutputRisingInterruptEnable |  
        kCMP_OutputFallingInterruptEnable);  
  
    while (1)  
    {  
    }  
}
```

## Data Structures

- struct [cmp\\_config\\_t](#)  
*Configures the comparator.* [More...](#)
- struct [cmp\\_filter\\_config\\_t](#)  
*Configures the filter.* [More...](#)
- struct [cmp\\_dac\\_config\\_t](#)  
*Configures the internal DAC.* [More...](#)

## Enumerations

- enum `_cmp_interrupt_enable` {
   
    `kCMP_OutputRisingInterruptEnable` = CMP\_SCR\_IER\_MASK,  
`kCMP_OutputFallingInterruptEnable` = CMP\_SCR\_IEF\_MASK }
   
*Interrupt enable/disable mask.*
- enum `_cmp_status_flags` {
   
    `kCMP_OutputRisingEventFlag` = CMP\_SCR\_CFR\_MASK,  
`kCMP_OutputFallingEventFlag` = CMP\_SCR\_CFF\_MASK,  
`kCMP_OutputAssertEventFlag` = CMP\_SCR\_COUT\_MASK }
   
*Status flags' mask.*
- enum `cmp_hysteresis_mode_t` {
   
    `kCMP_HysteresisLevel0` = 0U,  
`kCMP_HysteresisLevel1` = 1U,  
`kCMP_HysteresisLevel2` = 2U,  
`kCMP_HysteresisLevel3` = 3U }
   
*CMP Hysteresis mode.*
- enum `cmp_reference_voltage_source_t` {
   
    `kCMP_VrefSourceVin1` = 0U,  
`kCMP_VrefSourceVin2` = 1U }
   
*CMP Voltage Reference source.*

## Driver version

- #define `FSL_CMP_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 0))
   
*CMP driver version 2.0.0.*

## Initialization

- void `CMP_Init` (CMP\_Type \*base, const `cmp_config_t` \*config)
   
*Initializes the CMP.*
- void `CMP_Deinit` (CMP\_Type \*base)
   
*De-initializes the CMP module.*
- static void `CMP_Enable` (CMP\_Type \*base, bool enable)
   
*Enables/disables the CMP module.*
- void `CMP_GetDefaultConfig` (`cmp_config_t` \*config)
   
*Initializes the CMP user configuration structure.*
- void `CMP_SetInputChannels` (CMP\_Type \*base, uint8\_t positiveChannel, uint8\_t negativeChannel)
   
*Sets the input channels for the comparator.*

## Advanced Features

- void `CMP_EnableDMA` (CMP\_Type \*base, bool enable)
   
*Enables/disables the DMA request for rising/falling events.*
- static void `CMP_EnableWindowMode` (CMP\_Type \*base, bool enable)
   
*Enables/disables the window mode.*
- static void `CMP_EnablePassThroughMode` (CMP\_Type \*base, bool enable)
   
*Enables/disables the pass through mode.*
- void `CMP_SetFilterConfig` (CMP\_Type \*base, const `cmp_filter_config_t` \*config)
   
*Configures the filter.*

## Data Structure Documentation

- void [CMP\\_SetDACConfig](#) (CMP\_Type \*base, const cmp\_dac\_config\_t \*config)  
*Configures the internal DAC.*
- void [CMP\\_EnableInterrupts](#) (CMP\_Type \*base, uint32\_t mask)  
*Enables the interrupts.*
- void [CMP\\_DisableInterrupts](#) (CMP\_Type \*base, uint32\_t mask)  
*Disables the interrupts.*

## Results

- uint32\_t [CMP\\_GetStatusFlags](#) (CMP\_Type \*base)  
*Gets the status flags.*
- void [CMP\\_ClearStatusFlags](#) (CMP\_Type \*base, uint32\_t mask)  
*Clears the status flags.*

## 6.3 Data Structure Documentation

### 6.3.1 struct cmp\_config\_t

#### Data Fields

- bool [enableCmp](#)  
*Enable the CMP module.*
- [cmp\\_hysteresis\\_mode\\_t](#) [hysteresisMode](#)  
*CMP Hysteresis mode.*
- bool [enableHighSpeed](#)  
*Enable High-speed (HS) comparison mode.*
- bool [enableInvertOutput](#)  
*Enable the inverted comparator output.*
- bool [useUnfilteredOutput](#)  
*Set the compare output(COUT) to equal COUTA(true) or COUT(false).*
- bool [enablePinOut](#)  
*The comparator output is available on the associated pin.*

#### 6.3.1.0.0.4 Field Documentation

6.3.1.0.0.4.1 `bool cmp_config_t::enableCmp`

6.3.1.0.0.4.2 `cmp_hysteresis_mode_t cmp_config_t::hysteresisMode`

6.3.1.0.0.4.3 `bool cmp_config_t::enableHighSpeed`

6.3.1.0.0.4.4 `bool cmp_config_t::enableInvertOutput`

6.3.1.0.0.4.5 `bool cmp_config_t::useUnfilteredOutput`

6.3.1.0.0.4.6 `bool cmp_config_t::enablePinOut`

### 6.3.2 `struct cmp_filter_config_t`

#### Data Fields

- `bool enableSample`  
*Using the external SAMPLE as a sampling clock input or using a divided bus clock.*
- `uint8_t filterCount`  
*Filter Sample Count.*
- `uint8_t filterPeriod`  
*Filter Sample Period.*

#### 6.3.2.0.0.5 Field Documentation

6.3.2.0.0.5.1 `bool cmp_filter_config_t::enableSample`

6.3.2.0.0.5.2 `uint8_t cmp_filter_config_t::filterCount`

Available range is 1-7; 0 disables the filter.

6.3.2.0.0.5.3 `uint8_t cmp_filter_config_t::filterPeriod`

The divider to the bus clock. Available range is 0-255.

### 6.3.3 `struct cmp_dac_config_t`

#### Data Fields

- `cmp_reference_voltage_source_t referenceVoltageSource`  
*Supply voltage reference source.*
- `uint8_t DACValue`  
*Value for the DAC Output Voltage.*

## Enumeration Type Documentation

### 6.3.3.0.0.6 Field Documentation

6.3.3.0.0.6.1 `cmp_reference_voltage_source_t cmp_dac_config_t::referenceVoltageSource`

6.3.3.0.0.6.2 `uint8_t cmp_dac_config_t::DACValue`

Available range is 0-63.

## 6.4 Macro Definition Documentation

6.4.1 `#define FSL_CMP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

## 6.5 Enumeration Type Documentation

### 6.5.1 enum \_cmp\_interrupt\_enable

Enumerator

*kCMP\_OutputRisingInterruptEnable* Comparator interrupt enable rising.

*kCMP\_OutputFallingInterruptEnable* Comparator interrupt enable falling.

### 6.5.2 enum \_cmp\_status\_flags

Enumerator

*kCMP\_OutputRisingEventFlag* Rising-edge on the comparison output has occurred.

*kCMP\_OutputFallingEventFlag* Falling-edge on the comparison output has occurred.

*kCMP\_OutputAssertEventFlag* Return the current value of the analog comparator output.

### 6.5.3 enum cmp\_hysteresis\_mode\_t

Enumerator

*kCMP\_HysteresisLevel0* Hysteresis level 0.

*kCMP\_HysteresisLevel1* Hysteresis level 1.

*kCMP\_HysteresisLevel2* Hysteresis level 2.

*kCMP\_HysteresisLevel3* Hysteresis level 3.

### 6.5.4 enum cmp\_reference\_voltage\_source\_t

Enumerator

*kCMP\_VrefSourceVin1* Vin1 is selected as a resistor ladder network supply reference Vin.

*kCMP\_VrefSourceVin2* Vin2 is selected as a resistor ladder network supply reference Vin.

## 6.6 Function Documentation

### 6.6.1 void CMP\_Init ( **CMP\_Type** \* *base*, **const cmp\_config\_t** \* *config* )

This function initializes the CMP module. The operations included are as follows.

- Enabling the clock for CMP module.
- Configuring the comparator.
- Enabling the CMP module. Note that for some devices, multiple CMP instances share the same clock gate. In this case, to enable the clock for any instance enables all CMPs. See the appropriate MCU reference manual for the clock assignment of the CMP.

Parameters

<i>base</i>	CMP peripheral base address.
<i>config</i>	Pointer to the configuration structure.

### 6.6.2 void CMP\_Deinit ( **CMP\_Type** \* *base* )

This function de-initializes the CMP module. The operations included are as follows.

- Disabling the CMP module.
- Disabling the clock for CMP module.

This function disables the clock for the CMP. Note that for some devices, multiple CMP instances share the same clock gate. In this case, before disabling the clock for the CMP, ensure that all the CMP instances are not used.

Parameters

<i>base</i>	CMP peripheral base address.
-------------	------------------------------

### 6.6.3 static void CMP\_Enable ( **CMP\_Type** \* *base*, **bool enable** ) [inline], [static]

Parameters

<i>base</i>	CMP peripheral base address.
-------------	------------------------------

## Function Documentation

<i>enable</i>	Enables or disables the module.
---------------	---------------------------------

### 6.6.4 void CMP\_GetDefaultConfig ( cmp\_config\_t \* *config* )

This function initializes the user configuration structure to these default values.

```
* config->enableCmp          = true;
* config->hysteresisMode     = kCMP_HysteresisLevel0;
* config->enableHighSpeed   = false;
* config->enableInvertOutput = false;
* config->useUnfilteredOutput= false;
* config->enablePinOut      = false;
* config->enableTriggerMode = false;
*
```

Parameters

<i>config</i>	Pointer to the configuration structure.
---------------	---

### 6.6.5 void CMP\_SetInputChannels ( CMP\_Type \* *base*, uint8\_t *positiveChannel*, uint8\_t *negativeChannel* )

This function sets the input channels for the comparator. Note that two input channels cannot be set the same way in the application. When the user selects the same input from the analog mux to the positive and negative port, the comparator is disabled automatically.

Parameters

<i>base</i>	CMP peripheral base address.
<i>positive-Channel</i>	Positive side input channel number. Available range is 0-7.
<i>negative-Channel</i>	Negative side input channel number. Available range is 0-7.

### 6.6.6 void CMP\_EnableDMA ( CMP\_Type \* *base*, bool *enable* )

This function enables/disables the DMA request for rising/falling events. Either event triggers the generation of the DMA request from CMP if the DMA feature is enabled. Both events are ignored for generating the DMA request from the CMP if the DMA is disabled.

Parameters

<i>base</i>	CMP peripheral base address.
<i>enable</i>	Enables or disables the feature.

**6.6.7 static void CMP\_EnableWindowMode ( *CMP\_Type* \* *base*, *bool enable* )  
[inline], [static]**

Parameters

<i>base</i>	CMP peripheral base address.
<i>enable</i>	Enables or disables the feature.

**6.6.8 static void CMP\_EnablePassThroughMode ( *CMP\_Type* \* *base*, *bool enable* ) [inline], [static]**

Parameters

<i>base</i>	CMP peripheral base address.
<i>enable</i>	Enables or disables the feature.

**6.6.9 void CMP\_SetFilterConfig ( *CMP\_Type* \* *base*, *const cmp\_filter\_config\_t* \* *config* )**

Parameters

<i>base</i>	CMP peripheral base address.
<i>config</i>	Pointer to the configuration structure.

**6.6.10 void CMP\_SetDACConfig ( *CMP\_Type* \* *base*, *const cmp\_dac\_config\_t* \* *config* )**

## Function Documentation

Parameters

<i>base</i>	CMP peripheral base address.
<i>config</i>	Pointer to the configuration structure. "NULL" disables the feature.

### 6.6.11 void CMP\_EnableInterrupts ( **CMP\_Type** \* *base*, **uint32\_t** *mask* )

Parameters

<i>base</i>	CMP peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_cmp_interrupt_enable".

### 6.6.12 void CMP\_DisableInterrupts ( **CMP\_Type** \* *base*, **uint32\_t** *mask* )

Parameters

<i>base</i>	CMP peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_cmp_interrupt_enable".

### 6.6.13 **uint32\_t** CMP\_GetStatusFlags ( **CMP\_Type** \* *base* )

Parameters

<i>base</i>	CMP peripheral base address.
-------------	------------------------------

Returns

Mask value for the asserted flags. See "\_cmp\_status\_flags".

### 6.6.14 void CMP\_ClearStatusFlags ( **CMP\_Type** \* *base*, **uint32\_t** *mask* )

## Parameters

<i>base</i>	CMP peripheral base address.
<i>mask</i>	Mask value for the flags. See "_cmp_status_flags".

## Function Documentation

# Chapter 7

## CMT: Carrier Modulator Transmitter Driver

### 7.1 Overview

The carrier modulator transmitter (CMT) module provides the means to generate the protocol timing and carrier signals for a wide variety of encoding schemes. The CMT incorporates hardware to off-load the critical and/or lengthy timing requirements associated with signal generation from the CPU. The MCUXpresso SDK provides a driver for the CMT module of the MCUXpresso SDK devices.

### 7.2 Clock formulas

The CMT module has internal clock dividers. It was originally designed to be based on an 8 MHz bus clock that can be divided by 1, 2, 4, or 8 according to the specification. To be compatible with a higher bus frequency, the primary prescaler (PPS) was developed to receive a higher frequency and generate a clock enable signal called an intermediate frequency (IF). The IF must be approximately equal to 8 MHz and works as a clock enable to the secondary prescaler. For the PPS, the prescaler is selected according to the bus clock to generate an intermediate clock approximate to 8 MHz and is selected as (bus\_clock\_hz/8000000). The secondary prescaler is the "cmtDivider". The clocks for the CMT module are listed below.

1. CMT clock frequency = bus\_clock\_Hz / (bus\_clock\_Hz / 8000000) / cmtDivider
2. CMT carrier and generator frequency = CMT clock frequency / (highCount1 + lowCount1)  
(In FSK mode, the second frequency = CMT clock frequency / (highCount2 + lowCount2))
3. CMT infrared output signal frequency
  - a. In Time and Baseband mode  
CMT IRO signal mark time = (markCount + 1) / (CMT clock frequency / 8)  
CMT IRO signal space time = spaceCount / (CMT clock frequency / 8)
  - b. In FSK mode  
CMT IRO signal mark time = (markCount + 1) / CMT carrier and generator frequency  
CMT IRO signal space time = spaceCount / CMT carrier and generator frequency

### 7.3 Typical use case

This is an example code to initialize data.

```
cmt_config_t config;
cmt_modulate_config_t modulateConfig;
uint32_t busClock;

// Gets the bus clock for the CMT module.
busClock = CLOCK_GetFreq(kCLOCK_BusClk);

CMT_GetDefaultConfig(&config);

// Interrupts is enabled to change the modulate mark and space count.
config.isInterruptEnabled = true;
```

## Typical use case

```
CMT_Init(CMT, &config, busClock);

// Prepares the modulate configuration for a use case.
modulateConfig.highCount1 = ...;
modulateConfig.lowCount1 = ...;
modulateConfig.markCount = ...;
modulateConfig.spaceCount = ...;

// Sets the time mode.
CMT_SetMode(CMT, kCMT_TimeMode, &modulateConfig);
```

This is an example IRQ handler to change the mark and space count to complete data modulation.

```
// The data length has been transmitted.
uint32_t g_CmtDataBitLen;

void CMT_IRQHandler(void)
{
    if (CMT_GetStatusFlags(CMT))
    {
        if (g_CmtDataBitLen <= CMT_TEST_DATA_BITS)
        {
            // LSB.
            if (data & ((uint32_t)0x01 << g_CmtDataBitLen))
            {
                CMT_SetModulateMarkSpace(CMT, g_CmtModDataOneMarkCount,
                g_CmtModDataOneSpaceCount);
            }
            else
            {
                CMT_SetModulateMarkSpace(CMT, g_CmtModDataZeroMarkCount,
                g_CmtModDataZeroSpaceCount);
            }
        }
    }
}
```

## Data Structures

- struct [cmt\\_modulate\\_config\\_t](#)  
*CMT carrier generator and modulator configuration structure.* [More...](#)
- struct [cmt\\_config\\_t](#)  
*CMT basic configuration structure.* [More...](#)

## Enumerations

- enum [cmt\\_mode\\_t](#) {  
    kCMT\_DirectIROCtl = 0x00U,  
    kCMT\_TimeMode = 0x01U,  
    kCMT\_FSKMode = 0x05U,  
    kCMT\_BasebandMode = 0x09U }  
*The modes of CMT.*
- enum [cmt\\_primary\\_clkdiv\\_t](#) {

```
kCMT_PrimaryClkDiv1 = 0U,
kCMT_PrimaryClkDiv2 = 1U,
kCMT_PrimaryClkDiv3 = 2U,
kCMT_PrimaryClkDiv4 = 3U,
kCMT_PrimaryClkDiv5 = 4U,
kCMT_PrimaryClkDiv6 = 5U,
kCMT_PrimaryClkDiv7 = 6U,
kCMT_PrimaryClkDiv8 = 7U,
kCMT_PrimaryClkDiv9 = 8U,
kCMT_PrimaryClkDiv10 = 9U,
kCMT_PrimaryClkDiv11 = 10U,
kCMT_PrimaryClkDiv12 = 11U,
kCMT_PrimaryClkDiv13 = 12U,
kCMT_PrimaryClkDiv14 = 13U,
kCMT_PrimaryClkDiv15 = 14U,
kCMT_PrimaryClkDiv16 = 15U }
```

*The CMT clock divide primary prescaler.*

- enum `cmt_second_clkdiv_t` {
 

```
kCMT_SecondClkDiv1 = 0U,
kCMT_SecondClkDiv2 = 1U,
kCMT_SecondClkDiv4 = 2U,
kCMT_SecondClkDiv8 = 3U }
```

*The CMT clock divide secondary prescaler.*

- enum `cmt_infrared_output_polarity_t` {
 

```
kCMT_IROActiveLow = 0U,
kCMT_IROActiveHigh = 1U }
```

*The CMT infrared output polarity.*

- enum `cmt_infrared_output_state_t` {
 

```
kCMT_IROCtlLow = 0U,
kCMT_IROCtlHigh = 1U }
```

*The CMT infrared output signal state control.*

- enum `_cmt_interrupt_enable` { `kCMT_EndOfCycleInterruptEnable` = CMT\_MSC\_EOCIE\_MASK }

*CMT interrupt configuration structure, default settings all disabled.*

## Driver version

- #define `FSL_CMT_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 1))  
*CMT driver version 2.0.1.*

## Initialization and deinitialization

- void `CMT_GetDefaultConfig` (`cmt_config_t` \*config)  
*Gets the CMT default configuration structure.*
- void `CMT_Init` (`CMT_Type` \*base, const `cmt_config_t` \*config, `uint32_t` busClock\_Hz)  
*Initializes the CMT module.*
- void `CMT_Deinit` (`CMT_Type` \*base)

## Data Structure Documentation

*Disables the CMT module and gate control.*

## Basic Control Operations

- void [CMT\\_SetMode](#) (CMT\_Type \*base, [cmt\\_mode\\_t](#) mode, [cmt\\_modulate\\_config\\_t](#) \*modulateConfig)  
*Selects the mode for CMT.*
- [cmt\\_mode\\_t CMT\\_GetMode](#) (CMT\_Type \*base)  
*Gets the mode of the CMT module.*
- uint32\_t [CMT\\_GetCMTFrequency](#) (CMT\_Type \*base, uint32\_t busClock\_Hz)  
*Gets the actual CMT clock frequency.*
- static void [CMT\\_SetCarrierGenerateCountOne](#) (CMT\_Type \*base, uint32\_t highCount, uint32\_t lowCount)  
*Sets the primary data set for the CMT carrier generator counter.*
- static void [CMT\\_SetCarrierGenerateCountTwo](#) (CMT\_Type \*base, uint32\_t highCount, uint32\_t lowCount)  
*Sets the secondary data set for the CMT carrier generator counter.*
- void [CMT\\_SetModulateMarkSpace](#) (CMT\_Type \*base, uint32\_t markCount, uint32\_t spaceCount)  
*Sets the modulation mark and space time period for the CMT modulator.*
- static void [CMT\\_EnableExtendedSpace](#) (CMT\_Type \*base, bool enable)  
*Enables or disables the extended space operation.*
- void [CMT\\_SetIroState](#) (CMT\_Type \*base, [cmt\\_infrared\\_output\\_state\\_t](#) state)  
*Sets the IRO (infrared output) signal state.*
- static void [CMT\\_EnableInterrupts](#) (CMT\_Type \*base, uint32\_t mask)  
*Enables the CMT interrupt.*
- static void [CMT\\_DisableInterrupts](#) (CMT\_Type \*base, uint32\_t mask)  
*Disables the CMT interrupt.*
- static uint32\_t [CMT\\_GetStatusFlags](#) (CMT\_Type \*base)  
*Gets the end of the cycle status flag.*

## 7.4 Data Structure Documentation

### 7.4.1 struct cmt\_modulate\_config\_t

#### Data Fields

- uint8\_t [highCount1](#)  
*The high-time for carrier generator first register.*
- uint8\_t [lowCount1](#)  
*The low-time for carrier generator first register.*
- uint8\_t [highCount2](#)  
*The high-time for carrier generator second register for FSK mode.*
- uint8\_t [lowCount2](#)  
*The low-time for carrier generator second register for FSK mode.*
- uint16\_t [markCount](#)  
*The mark time for the modulator gate.*
- uint16\_t [spaceCount](#)  
*The space time for the modulator gate.*

#### 7.4.1.0.0.7 Field Documentation

- 7.4.1.0.0.7.1 `uint8_t cmt_modulate_config_t::highCount1`
- 7.4.1.0.0.7.2 `uint8_t cmt_modulate_config_t::lowCount1`
- 7.4.1.0.0.7.3 `uint8_t cmt_modulate_config_t::highCount2`
- 7.4.1.0.0.7.4 `uint8_t cmt_modulate_config_t::lowCount2`
- 7.4.1.0.0.7.5 `uint16_t cmt_modulate_config_t::markCount`
- 7.4.1.0.0.7.6 `uint16_t cmt_modulate_config_t::spaceCount`

#### 7.4.2 struct cmt\_config\_t

##### Data Fields

- `bool isInterruptEnabled`  
*Timer interrupt 0-disable, 1-enable.*
- `bool isIroEnabled`  
*The IRO output 0-disabled, 1-enabled.*
- `cmt_infrared_output_polarity_t iroPolarity`  
*The IRO polarity.*
- `cmt_second_clkdiv_t divider`  
*The CMT clock divide prescaler.*

#### 7.4.2.0.0.8 Field Documentation

- 7.4.2.0.0.8.1 `bool cmt_config_t::isInterruptEnabled`
- 7.4.2.0.0.8.2 `bool cmt_config_t::isIroEnabled`
- 7.4.2.0.0.8.3 `cmt_infrared_output_polarity_t cmt_config_t::iroPolarity`
- 7.4.2.0.0.8.4 `cmt_second_clkdiv_t cmt_config_t::divider`

#### 7.5 Macro Definition Documentation

7.5.1 `#define FSL_CMT_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

#### 7.6 Enumeration Type Documentation

##### 7.6.1 enum cmt\_mode\_t

Enumerator

- kCMT\_DirectIROCtl*** Carrier modulator is disabled and the IRO signal is directly in software control.
- kCMT\_TimeMode*** Carrier modulator is enabled in time mode.

## Enumeration Type Documentation

***kCMT\_FSKMode*** Carrier modulator is enabled in FSK mode.

***kCMT\_BasebandMode*** Carrier modulator is enabled in baseband mode.

### 7.6.2 enum cmt\_primary\_clkdiv\_t

The primary clock divider is used to divider the bus clock to get the intermediate frequency to approximately equal to 8 MHZ. When the bus clock is 8 MHZ, set primary prescaler to "kCMT\_PrimaryClkDiv1".

Enumerator

***kCMT\_PrimaryClkDiv1*** The intermediate frequency is the bus clock divided by 1.

***kCMT\_PrimaryClkDiv2*** The intermediate frequency is the bus clock divided by 2.

***kCMT\_PrimaryClkDiv3*** The intermediate frequency is the bus clock divided by 3.

***kCMT\_PrimaryClkDiv4*** The intermediate frequency is the bus clock divided by 4.

***kCMT\_PrimaryClkDiv5*** The intermediate frequency is the bus clock divided by 5.

***kCMT\_PrimaryClkDiv6*** The intermediate frequency is the bus clock divided by 6.

***kCMT\_PrimaryClkDiv7*** The intermediate frequency is the bus clock divided by 7.

***kCMT\_PrimaryClkDiv8*** The intermediate frequency is the bus clock divided by 8.

***kCMT\_PrimaryClkDiv9*** The intermediate frequency is the bus clock divided by 9.

***kCMT\_PrimaryClkDiv10*** The intermediate frequency is the bus clock divided by 10.

***kCMT\_PrimaryClkDiv11*** The intermediate frequency is the bus clock divided by 11.

***kCMT\_PrimaryClkDiv12*** The intermediate frequency is the bus clock divided by 12.

***kCMT\_PrimaryClkDiv13*** The intermediate frequency is the bus clock divided by 13.

***kCMT\_PrimaryClkDiv14*** The intermediate frequency is the bus clock divided by 14.

***kCMT\_PrimaryClkDiv15*** The intermediate frequency is the bus clock divided by 15.

***kCMT\_PrimaryClkDiv16*** The intermediate frequency is the bus clock divided by 16.

### 7.6.3 enum cmt\_second\_clkdiv\_t

The second prescaler can be used to divide the 8 MHZ CMT clock by 1, 2, 4, or 8 according to the specification.

Enumerator

***kCMT\_SecondClkDiv1*** The CMT clock is the intermediate frequency frequency divided by 1.

***kCMT\_SecondClkDiv2*** The CMT clock is the intermediate frequency frequency divided by 2.

***kCMT\_SecondClkDiv4*** The CMT clock is the intermediate frequency frequency divided by 4.

***kCMT\_SecondClkDiv8*** The CMT clock is the intermediate frequency frequency divided by 8.

## 7.6.4 enum cmt\_infrared\_output\_polarity\_t

Enumerator

***kCMT\_IROActiveLow*** The CMT infrared output signal polarity is active-low.

***kCMT\_IROActiveHigh*** The CMT infrared output signal polarity is active-high.

## 7.6.5 enum cmt\_infrared\_output\_state\_t

Enumerator

***kCMT\_IROCtlLow*** The CMT Infrared output signal state is controlled to low.

***kCMT\_IROCtlHigh*** The CMT Infrared output signal state is controlled to high.

## 7.6.6 enum \_cmt\_interrupt\_enable

This structure contains the settings for all of the CMT interrupt configurations.

Enumerator

***kCMT\_EndOfCycleInterruptEnable*** CMT end of cycle interrupt.

## 7.7 Function Documentation

### 7.7.1 void CMT\_GetDefaultConfig ( ***cmt\_config\_t \* config*** )

This API gets the default configuration structure for the [CMT\\_Init\(\)](#). Use the initialized structure unchanged in [CMT\\_Init\(\)](#) or modify fields of the structure before calling the [CMT\\_Init\(\)](#).

Parameters

<i>config</i>	The CMT configuration structure pointer.
---------------	--

### 7.7.2 void CMT\_Init ( ***CMT\_Type \* base, const cmt\_config\_t \* config, uint32\_t busClock\_Hz*** )

This function ungates the module clock and sets the CMT internal clock, interrupt, and infrared output signal for the CMT module.

## Function Documentation

Parameters

<i>base</i>	CMT peripheral base address.
<i>config</i>	The CMT basic configuration structure.
<i>busClock_Hz</i>	The CMT module input clock - bus clock frequency.

### 7.7.3 void CMT\_Deinit ( CMT\_Type \* *base* )

This function disables CMT modulator, interrupts, and gates the CMT clock control. CMT\_Init must be called to use the CMT again.

Parameters

<i>base</i>	CMT peripheral base address.
-------------	------------------------------

### 7.7.4 void CMT\_SetMode ( CMT\_Type \* *base*, cmt\_mode\_t *mode*, cmt\_modulate\_config\_t \* *modulateConfig* )

Parameters

<i>base</i>	CMT peripheral base address.
<i>mode</i>	The CMT feature mode enumeration. See "cmt_mode_t".
<i>modulateConfig</i>	The carrier generation and modulator configuration.

### 7.7.5 cmt\_mode\_t CMT\_GetMode ( CMT\_Type \* *base* )

Parameters

<i>base</i>	CMT peripheral base address.
-------------	------------------------------

Returns

The CMT mode. kCMT\_DirectIROCtl Carrier modulator is disabled; the IRO signal is directly in software control. kCMT\_TimeMode Carrier modulator is enabled in time mode. kCMT\_FSKMode Carrier modulator is enabled in FSK mode. kCMT\_BasebandMode Carrier modulator is enabled in baseband mode.

7.7.6 `uint32_t CMT_GetCMTFrequency( CMT_Type * base, uint32_t busClock_Hz )`

## Function Documentation

Parameters

<i>base</i>	CMT peripheral base address.
<i>busClock_Hz</i>	CMT module input clock - bus clock frequency.

Returns

The CMT clock frequency.

### 7.7.7 static void CMT\_SetCarrierGenerateCountOne ( CMT\_Type \* *base*, uint32\_t *highCount*, uint32\_t *lowCount* ) [inline], [static]

This function sets the high-time and low-time of the primary data set for the CMT carrier generator counter to control the period and the duty cycle of the output carrier signal. If the CMT clock period is Tcmt, the period of the carrier generator signal equals (*highCount* + *lowCount*) \* Tcmt. The duty cycle equals to *highCount* / (*highCount* + *lowCount*).

Parameters

<i>base</i>	CMT peripheral base address.
<i>highCount</i>	The number of CMT clocks for carrier generator signal high time, integer in the range of 1 ~ 0xFF.
<i>lowCount</i>	The number of CMT clocks for carrier generator signal low time, integer in the range of 1 ~ 0xFF.

### 7.7.8 static void CMT\_SetCarrierGenerateCountTwo ( CMT\_Type \* *base*, uint32\_t *highCount*, uint32\_t *lowCount* ) [inline], [static]

This function is used for FSK mode setting the high-time and low-time of the secondary data set CMT carrier generator counter to control the period and the duty cycle of the output carrier signal. If the CMT clock period is Tcmt, the period of the carrier generator signal equals (*highCount* + *lowCount*) \* Tcmt. The duty cycle equals *highCount* / (*highCount* + *lowCount*).

Parameters

<i>base</i>	CMT peripheral base address.
-------------	------------------------------

<i>highCount</i>	The number of CMT clocks for carrier generator signal high time, integer in the range of 1 ~ 0xFF.
<i>lowCount</i>	The number of CMT clocks for carrier generator signal low time, integer in the range of 1 ~ 0xFF.

### 7.7.9 void CMT\_SetModulateMarkSpace ( **CMT\_Type** \* *base*, **uint32\_t** *markCount*, **uint32\_t** *spaceCount* )

This function sets the mark time period of the CMT modulator counter to control the mark time of the output modulated signal from the carrier generator output signal. If the CMT clock frequency is Fcmt and the carrier out signal frequency is fcg:

- In Time and Baseband mode: The mark period of the generated signal equals  $(\text{markCount} + 1) / (\text{Fcmt}/8)$ . The space period of the generated signal equals  $\text{spaceCount} / (\text{Fcmt}/8)$ .
- In FSK mode: The mark period of the generated signal equals  $(\text{markCount} + 1)/\text{fcg}$ . The space period of the generated signal equals  $\text{spaceCount} / \text{fcg}$ .

Parameters

<i>base</i>	Base address for current CMT instance.
<i>markCount</i>	The number of clock period for CMT modulator signal mark period, in the range of 0 ~ 0xFFFF.
<i>spaceCount</i>	The number of clock period for CMT modulator signal space period, in the range of the 0 ~ 0xFFFF.

### 7.7.10 static void CMT\_EnableExtendedSpace ( **CMT\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

This function is used to make the space period longer for time, baseband, and FSK modes.

Parameters

<i>base</i>	CMT peripheral base address.
<i>enable</i>	True enable the extended space, false disable the extended space.

## Function Documentation

**7.7.11 void CMT\_SetIroState ( CMT\_Type \* *base*, cmt\_infrared\_output\_state\_t *state* )**

Changes the states of the IRO signal when the kCMT\_DirectIROMode mode is set and the IRO signal is enabled.

Parameters

<i>base</i>	CMT peripheral base address.
<i>state</i>	The control of the IRO signal. See "cmt_infrared_output_state_t"

### 7.7.12 static void CMT\_EnableInterrupts ( CMT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function enables the CMT interrupts according to the provided mask if enabled. The CMT only has the end of the cycle interrupt - an interrupt occurs at the end of the modulator cycle. This interrupt provides a means for the user to reload the new mark/space values into the CMT modulator data registers and verify the modulator mark and space. For example, to enable the end of cycle, do the following.

```
*     CMT_EnableInterrupts(CMT,
                           kCMT_EndOfCycleInterruptEnable);
*
```

Parameters

<i>base</i>	CMT peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of <a href="#">_cmt_interrupt_enable</a> .

### 7.7.13 static void CMT\_DisableInterrupts ( CMT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function disables the CMT interrupts according to the provided maskIf enabled. The CMT only has the end of the cycle interrupt. For example, to disable the end of cycle, do the following.

```
*     CMT_DisableInterrupts(CMT,
                           kCMT_EndOfCycleInterruptEnable);
*
```

Parameters

<i>base</i>	CMT peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of <a href="#">_cmt_interrupt_enable</a> .

### 7.7.14 static uint32\_t CMT\_GetStatusFlags ( CMT\_Type \* *base* ) [inline], [static]

The flag is set:

## Function Documentation

- When the modulator is not currently active and carrier and modulator are set to start the initial CMT transmission.
- At the end of each modulation cycle when the counter is reloaded and the carrier and modulator are enabled.

Parameters

<i>base</i>	CMT peripheral base address.
-------------	------------------------------

Returns

Current status of the end of cycle status flag

- non-zero: End-of-cycle has occurred.
- zero: End-of-cycle has not yet occurred since the flag last cleared.

# Chapter 8

## CRC: Cyclic Redundancy Check Driver

### 8.1 Overview

The MCUXpresso SDK provides a Peripheral driver for the Cyclic Redundancy Check (CRC) module of MCUXpresso SDK devices.

The cyclic redundancy check (CRC) module generates 16/32-bit CRC code for error detection. The CRC module also provides a programmable polynomial, seed, and other parameters required to implement a 16-bit or 32-bit CRC standard.

### 8.2 CRC Driver Initialization and Configuration

[CRC\\_Init\(\)](#) function enables the clock gate for the CRC module in the SIM module and fully (re-)configures the CRC module according to the configuration structure. The seed member of the configuration structure is the initial checksum for which new data can be added to. When starting a new checksum computation, the seed is set to the initial checksum per the CRC protocol specification. For continued checksum operation, the seed is set to the intermediate checksum value as obtained from previous calls to [CRC\\_Get16bitResult\(\)](#) or [CRC\\_Get32bitResult\(\)](#) function. After calling the [CRC\\_Init\(\)](#), one or multiple [CRC\\_WriteData\(\)](#) calls follow to update the checksum with data and [CRC\\_Get16bitResult\(\)](#) or [CRC\\_Get32bitResult\(\)](#) follow to read the result. The crcResult member of the configuration structure determines whether the [CRC\\_Get16bitResult\(\)](#) or [CRC\\_Get32bitResult\(\)](#) return value is a final checksum or an intermediate checksum. The [CRC\\_Init\(\)](#) function can be called as many times as required allowing for runtime changes of the CRC protocol.

[CRC\\_GetDefaultConfig\(\)](#) function can be used to set the module configuration structure with parameters for CRC-16/CCIT-FALSE protocol.

### 8.3 CRC Write Data

The [CRC\\_WriteData\(\)](#) function adds data to the CRC. Internally, it tries to use 32-bit reads and writes for all aligned data in the user buffer and 8-bit reads and writes for all unaligned data in the user buffer. This function can update the CRC with user-supplied data chunks of an arbitrary size, so one can update the CRC byte by byte or with all bytes at once. Prior to calling the CRC configuration function [CRC\\_Init\(\)](#) fully specifies the CRC module configuration for the [CRC\\_WriteData\(\)](#) call.

### 8.4 CRC Get Checksum

The [CRC\\_Get16bitResult\(\)](#) or [CRC\\_Get32bitResult\(\)](#) function reads the CRC module data register. Depending on the prior CRC module usage, the return value is either an intermediate checksum or the final checksum. For example, for 16-bit CRCs the following call sequences can be used.

[CRC\\_Init\(\)](#) / [CRC\\_WriteData\(\)](#) / [CRC\\_Get16bitResult\(\)](#) to get the final checksum.

[CRC\\_Init\(\)](#) / [CRC\\_WriteData\(\)](#) / ... / [CRC\\_WriteData\(\)](#) / [CRC\\_Get16bitResult\(\)](#) to get the final checksum.

## CRC Driver Examples

`CRC_Init()` / `CRC_WriteData()` / `CRC_Get16bitResult()` to get an intermediate checksum.

`CRC_Init()` / `CRC_WriteData()` / ... / `CRC_WriteData()` / `CRC_Get16bitResult()` to get an intermediate checksum.

## 8.5 Comments about API usage in RTOS

If multiple RTOS tasks share the CRC module to compute checksums with different data and/or protocols, the following needs to be implemented by the user.

The triplets

`CRC_Init()` / `CRC_WriteData()` / `CRC_Get16bitResult()` or `CRC_Get32bitResult()`

The triplets are protected by the RTOS mutex to protect the CRC module against concurrent accesses from different tasks. This is an example.

```
CRC_ModuleRTOS_Mutex_Lock;  
CRC_Init();  
CRC_WriteData();  
CRC_Get16bitResult();  
CRC_ModuleRTOS_Mutex_Unlock;
```

## 8.6 Comments about API usage in interrupt handler

All APIs can be used from an interrupt handler although an interrupt latency of equal and lower priority interrupts increases. The user must protect against concurrent accesses from different interrupt handlers and/or tasks.

## 8.7 CRC Driver Examples

### 8.7.1 Simple examples

This is an example with the default CRC-16/CCIT-FALSE protocol.

```
crc_config_t config;  
CRC_Type *base;  
uint8_t data[] = {0x00, 0x01, 0x02, 0x03, 0x04};  
uint16_t checksum;  
  
base = CRC0;  
CRC_GetDefaultConfig(base, &config); /* default gives CRC-16/CCIT-FALSE */  
CRC_Init(base, &config);  
CRC_WriteData(base, data, sizeof(data));  
checksum = CRC_Get16bitResult(base);
```

This is an example with the CRC-32 protocol configuration.

```
crc_config_t config;  
uint32_t checksum;  
  
config.polynomial = 0x04C11DB7u;  
config.seed = 0xFFFFFFFFu;  
config.crcBits = kCrcBits32;  
config.reflectIn = true;
```

```

config.reflectOut = true;
config.complementChecksum = true;
config.crcResult = kCrcFinalChecksum;

CRC_Init(base, &config);
/* example: update by 1 byte at time */
while (dataSize)
{
    uint8_t c = GetCharacter();
    CRC_WriteData(base, &c, 1);
    dataSize--;
}
checksum = CRC_Get32bitResult(base);

```

## 8.7.2 Advanced examples

Assuming there are three tasks/threads, each using the CRC module to compute checksums of a different protocol, with context switches.

First, prepare the three CRC module initialization functions for three different protocols CRC-16 (ARC), CRC-16/CCIT-FALSE, and CRC-32. The table below lists the individual protocol specifications. See also <http://reveng.sourceforge.net/crc-catalogue/>.

	CRC-16/CCIT-FALSE	CRC-16	CRC-32
<b>Width</b>	16 bits	16 bits	32 bits
<b>Polynomial</b>	0x1021	0x8005	0x04C11DB7
<b>Initial seed</b>	0xFFFF	0x0000	0xFFFFFFFF
<b>Complement checksum</b>	No	No	Yes
<b>Reflect In</b>	No	Yes	Yes
<b>Reflect Out</b>	No	Yes	Yes

These are the corresponding initialization functions.

```

void InitCrc16_CCIT(CRC_Type *base, uint32_t seed, bool isLast)
{
    crc_config_t config;

    config.polynomial = 0x1021;
    config.seed = seed;
    config.reflectIn = false;
    config.reflectOut = false;
    config.complementChecksum = false;
    config.crcBits = kCrcBits16;
    config.crcResult = isLast?kCrcFinalChecksum:
        kCrcIntermediateChecksum;

    CRC_Init(base, &config);
}

void InitCrc16(CRC_Type *base, uint32_t seed, bool isLast)
{
    crc_config_t config;

```

## CRC Driver Examples

```
config.polynomial = 0x8005;
config.seed = seed;
config.reflectIn = true;
config.reflectOut = true;
config.complementChecksum = false;
config.crcBits = kCrcBits16;
config.crcResult = isLast?kCrcFinalChecksum:
    kCrcIntermediateChecksum;

    CRC_Init(base, &config);
}

void InitCrc32(CRC_Type *base, uint32_t seed, bool isLast)
{
    crc_config_t config;

    config.polynomial = 0x04C11DB7U;
    config.seed = seed;
    config.reflectIn = true;
    config.reflectOut = true;
    config.complementChecksum = true;
    config.crcBits = kCrcBits32;
    config.crcResult = isLast?kCrcFinalChecksum:
        kCrcIntermediateChecksum;

    CRC_Init(base, &config);
}
```

The following context switches show a possible API usage.

```
uint16_t checksumCrc16;
uint32_t checksumCrc32;
uint16_t checksumCrc16Ccit;

checksumCrc16 = 0x0;
checksumCrc32 = 0xFFFFFFFFU;
checksumCrc16Ccit = 0xFFFFU;

/* Task A bytes[0-3] */
InitCrc16(base, checksumCrc16, false);
CRC_WriteData(base, &data[0], 4);
checksumCrc16 = CRC_Get16bitResult(base);

/* Task B bytes[0-3] */
InitCrc16_CCIT(base, checksumCrc16Ccit, false);
CRC_WriteData(base, &data[0], 4);
checksumCrc16Ccit = CRC_Get16bitResult(base);

/* Task C 4 bytes[0-3] */
InitCrc32(base, checksumCrc32, false);
CRC_WriteData(base, &data[0], 4);
checksumCrc32 = CRC_Get32bitResult(base);

/* Task B add final 5 bytes[4-8] */
InitCrc16_CCIT(base, checksumCrc16Ccit, true);
CRC_WriteData(base, &data[4], 5);
checksumCrc16Ccit = CRC_Get16bitResult(base);

/* Task C 3 bytes[4-6] */
InitCrc32(base, checksumCrc32, false);
CRC_WriteData(base, &data[4], 3);
checksumCrc32 = CRC_Get32bitResult(base);

/* Task A 3 bytes[4-6] */
InitCrc16(base, checksumCrc16, false);
```

```

CRC_WriteData(base, &data[4], 3);
checksumCrc16 = CRC_Get16bitResult(base);

/* Task C add final 2 bytes[7-8] */
InitCrc32(base, checksumCrc32, true);
CRC_WriteData(base, &data[7], 2);
checksumCrc32 = CRC_Get32bitResult(base);

/* Task A add final 2 bytes[7-8] */
InitCrc16(base, checksumCrc16, true);
CRC_WriteData(base, &data[7], 2);
checksumCrc16 = CRC_Get16bitResult(base);

```

## Data Structures

- struct **crc\_config\_t**  
*CRC protocol configuration.* [More...](#)

## Macros

- #define **CRC\_DRIVER\_USE\_CRC16\_CCIT\_FALSE\_AS\_DEFAULT** 1  
*Default configuration structure filled by [CRC\\_GetDefaultConfig\(\)](#).*

## Enumerations

- enum **crc\_bits\_t** {
 kCrcBits16 = 0U,
 kCrcBits32 = 1U }
   
*CRC bit width.*
- enum **crc\_result\_t** {
 kCrcFinalChecksum = 0U,
 kCrcIntermediateChecksum = 1U }
   
*CRC result type.*

## Functions

- void **CRC\_Init** (CRC\_Type \*base, const **crc\_config\_t** \*config)  
*Enables and configures the CRC peripheral module.*
- static void **CRC\_Deinit** (CRC\_Type \*base)  
*Disables the CRC peripheral module.*
- void **CRC\_GetDefaultConfig** (**crc\_config\_t** \*config)  
*Loads default values to the CRC protocol configuration structure.*
- void **CRC\_WriteData** (CRC\_Type \*base, const uint8\_t \*data, size\_t dataSize)  
*Writes data to the CRC module.*
- uint32\_t **CRC\_Get32bitResult** (CRC\_Type \*base)  
*Reads the 32-bit checksum from the CRC module.*
- uint16\_t **CRC\_Get16bitResult** (CRC\_Type \*base)  
*Reads a 16-bit checksum from the CRC module.*

## Driver version

- #define **FSL\_CRC\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 1))  
*CRC driver version.*

## Macro Definition Documentation

### 8.8 Data Structure Documentation

#### 8.8.1 struct crc\_config\_t

This structure holds the configuration for the CRC protocol.

#### Data Fields

- **uint32\_t polynomial**  
*CRC Polynomial, MSBit first.*
- **uint32\_t seed**  
*Starting checksum value.*
- **bool reflectIn**  
*Reflect bits on input.*
- **bool reflectOut**  
*Reflect bits on output.*
- **bool complementChecksum**  
*True if the result shall be complement of the actual checksum.*
- **crc\_bits\_t crcBits**  
*Selects 16- or 32- bit CRC protocol.*
- **crc\_result\_t crcResult**  
*Selects final or intermediate checksum return from [CRC\\_Get16bitResult\(\)](#) or [CRC\\_Get32bitResult\(\)](#)*

#### 8.8.1.0.0.9 Field Documentation

##### 8.8.1.0.0.9.1 uint32\_t crc\_config\_t::polynomial

Example polynomial:  $0x1021 = 1_0000\_0010\_0001 = x^{12}+x^5+1$

##### 8.8.1.0.0.9.2 bool crc\_config\_t::reflectIn

##### 8.8.1.0.0.9.3 bool crc\_config\_t::reflectOut

##### 8.8.1.0.0.9.4 bool crc\_config\_t::complementChecksum

##### 8.8.1.0.0.9.5 crc\_bits\_t crc\_config\_t::crcBits

### 8.9 Macro Definition Documentation

#### 8.9.1 #define FSL\_CRC\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))

Version 2.0.1.

Current version: 2.0.1

Change log:

- Version 2.0.1
  - move DATA and DATALL macro definition from header file to source file

## 8.9.2 #define CRC\_DRIVER\_USE\_CRC16\_CCIT\_FALSE\_AS\_DEFAULT 1

Use CRC16-CCIT-FALSE as default.

## 8.10 Enumeration Type Documentation

### 8.10.1 enum crc\_bits\_t

Enumerator

*kCrcBits16* Generate 16-bit CRC code.

*kCrcBits32* Generate 32-bit CRC code.

### 8.10.2 enum crc\_result\_t

Enumerator

*kCrcFinalChecksum* CRC data register read value is the final checksum. Reflect out and final xor protocol features are applied.

*kCrcIntermediateChecksum* CRC data register read value is intermediate checksum (raw value). Reflect out and final xor protocol feature are not applied. Intermediate checksum can be used as a seed for [CRC\\_Init\(\)](#) to continue adding data to this checksum.

## 8.11 Function Documentation

### 8.11.1 void CRC\_Init ( **CRC\_Type** \* *base*, **const crc\_config\_t** \* *config* )

This function enables the clock gate in the SIM module for the CRC peripheral. It also configures the CRC module and starts a checksum computation by writing the seed.

Parameters

<i>base</i>	CRC peripheral address.
<i>config</i>	CRC module configuration structure.

### 8.11.2 static void CRC\_Deinit ( **CRC\_Type** \* *base* ) [inline], [static]

This function disables the clock gate in the SIM module for the CRC peripheral.

## Function Documentation

Parameters

<i>base</i>	CRC peripheral address.
-------------	-------------------------

### 8.11.3 void CRC\_GetDefaultConfig ( **crc\_config\_t** \* *config* )

Loads default values to the CRC protocol configuration structure. The default values are as follows.

```
* config->polynomial = 0x1021;
* config->seed = 0xFFFF;
* config->reflectIn = false;
* config->reflectOut = false;
* config->complementChecksum = false;
* config->crcBits = kCrcBits16;
* config->crcResult = kCrcFinalChecksum;
*
```

Parameters

<i>config</i>	CRC protocol configuration structure.
---------------	---------------------------------------

### 8.11.4 void CRC\_WriteData ( **CRC\_Type** \* *base*, **const uint8\_t** \* *data*, **size\_t** *dataSize* )

Writes input data buffer bytes to the CRC data register. The configured type of transpose is applied.

Parameters

<i>base</i>	CRC peripheral address.
<i>data</i>	Input data stream, MSByte in data[0].
<i>dataSize</i>	Size in bytes of the input data buffer.

### 8.11.5 **uint32\_t** CRC\_Get32bitResult ( **CRC\_Type** \* *base* )

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

<i>base</i>	CRC peripheral address.
-------------	-------------------------

Returns

An intermediate or the final 32-bit checksum, after configured transpose and complement operations.

#### **8.11.6 `uint16_t CRC_Get16bitResult ( CRC_Type * base )`**

Reads the CRC data register (either an intermediate or the final checksum). The configured type of transpose and complement is applied.

Parameters

<i>base</i>	CRC peripheral address.
-------------	-------------------------

Returns

An intermediate or the final 16-bit checksum, after configured transpose and complement operations.

## Function Documentation

# Chapter 9

## DAC: Digital-to-Analog Converter Driver

### 9.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Digital-to-Analog Converter (DAC) module of MCUXpresso SDK devices.

The DAC driver includes a basic DAC module (converter) and a DAC buffer.

The basic DAC module supports operations unique to the DAC converter in each DAC instance. The APIs in this part are used in the initialization phase, which enables the DAC module in the application. The APIs enable/disable the clock, enable/disable the module, and configure the converter. Call the initial APIs to prepare the DAC module for the application. The DAC buffer operates the DAC hardware buffer. The DAC module supports a hardware buffer to keep a group of DAC values to be converted. This feature supports updating the DAC output value automatically by triggering the buffer read pointer to move in the buffer. Use the APIs to configure the hardware buffer's trigger mode, watermark, work mode, and use size. Additionally, the APIs operate the DMA, interrupts, flags, the pointer (the index of the buffer), item values, and so on.

Note that the most functional features are designed for the DAC hardware buffer.

### 9.2 Typical use case

#### 9.2.1 Working as a basic DAC without the hardware buffer feature

```
// ...  
  
// Configures the DAC.  
DAC_GetDefaultConfig(&dacConfigStruct);  
DAC_Init(DEMO_DAC_INSTANCE, &dacConfigStruct);  
DAC_Enable(DEMO_DAC_INSTANCE, true);  
DAC_SetBufferReadPointer(DEMO_DAC_INSTANCE, 0U);  
  
// ...  
  
DAC_SetBufferValue(DEMO_DAC_INSTANCE, 0U, dacValue);
```

#### 9.2.2 Working with the hardware buffer

```
// ...  
  
EnableIRQ(DEMO_DAC_IRQ_ID);  
  
// ...  
  
// Configures the DAC.  
DAC_GetDefaultConfig(&dacConfigStruct);  
DAC_Init(DEMO_DAC_INSTANCE, &dacConfigStruct);  
DAC_Enable(DEMO_DAC_INSTANCE, true);
```

## Typical use case

```
// Configures the DAC buffer.
DAC_SetDefaultBufferConfig(&dacBufferConfigStruct);
DAC_SetBufferConfig(DEMO_DAC_INSTANCE, &dacBufferConfigStruct);
DAC_SetBufferReadPointer(DEMO_DAC_INSTANCE, 0U); // Make sure the read pointer
      to the start.
for (index = 0U, dacValue = 0; index < DEMO_DAC_USED_BUFFER_SIZE; index++, dacValue += (0xFFFFU /
    DEMO_DAC_USED_BUFFER_SIZE))
{
    DAC_SetBufferValue(DEMO_DAC_INSTANCE, index, dacValue);
}
// Clears flags.
#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferWatermarkInterruptFlag = false;
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferReadPointerTopPositionInterruptFlag = false;
    g_DacBufferReadPointerBottomPositionInterruptFlag = false;

// Enables interrupts.
mask = 0U;
#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    mask |= kDAC_BufferWatermarkInterruptEnable;
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    mask |= kDAC_BufferReadPointerTopInterruptEnable |
        kDAC_BufferReadPointerBottomInterruptEnable;
    DAC_EnableBuffer(DEMO_DAC_INSTANCE, true);
    DAC_EnableBufferInterrupts(DEMO_DAC_INSTANCE, mask);

// ISR for the DAC interrupt.
void DEMO_DAC IRQ_HANDLER_FUNC(void)
{
    uint32_t flags = DAC_GetBufferStatusFlags(DEMO_DAC_INSTANCE);

#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    if (kDAC_BufferWatermarkFlag == (
        kDAC_BufferWatermarkFlag & flags))
    {
        g_DacBufferWatermarkInterruptFlag = true;
    }
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    if (kDAC_BufferReadPointerTopPositionFlag == (
        kDAC_BufferReadPointerTopPositionFlag & flags))
    {
        g_DacBufferReadPointerTopPositionInterruptFlag = true;
    }
    if (kDAC_BufferReadPointerBottomPositionFlag == (
        kDAC_BufferReadPointerBottomPositionFlag & flags))
    {
        g_DacBufferReadPointerBottomPositionInterruptFlag = true;
    }
    DAC_ClearBufferStatusFlags(DEMO_DAC_INSTANCE, flags); /* Clear flags. */
}
```

## Data Structures

- struct [dac\\_config\\_t](#)  
*DAC module configuration.* [More...](#)
- struct [dac\\_buffer\\_config\\_t](#)  
*DAC buffer configuration.* [More...](#)

## Enumerations

- enum `_dac_buffer_status_flags` {
   
  `kDAC_BufferWatermarkFlag` = `DAC_SR_DACBFWMF_MASK`,
   
  `kDAC_BufferReadPointerTopPositionFlag` = `DAC_SR_DACBFRPTF_MASK`,
   
  `kDAC_BufferReadPointerBottomPositionFlag` = `DAC_SR_DACBFRPBF_MASK` }
   
*DAC buffer flags.*
- enum `_dac_buffer_interrupt_enable` {
   
  `kDAC_BufferWatermarkInterruptEnable` = `DAC_C0_DACBWIEN_MASK`,
   
  `kDAC_BufferReadPointerTopInterruptEnable` = `DAC_C0_DACBTIEN_MASK`,
   
  `kDAC_BufferReadPointerBottomInterruptEnable` = `DAC_C0_DACBBIEN_MASK` }
   
*DAC buffer interrupts.*
- enum `dac_reference_voltage_source_t` {
   
  `kDAC_ReferenceVoltageSourceVref1` = `0U`,
   
  `kDAC_ReferenceVoltageSourceVref2` = `1U` }
   
*DAC reference voltage source.*
- enum `dac_buffer_trigger_mode_t` {
   
  `kDAC_BufferTriggerByHardwareMode` = `0U`,
   
  `kDAC_BufferTriggerBySoftwareMode` = `1U` }
   
*DAC buffer trigger mode.*
- enum `dac_buffer_watermark_t` {
   
  `kDAC_BufferWatermark1Word` = `0U`,
   
  `kDAC_BufferWatermark2Word` = `1U`,
   
  `kDAC_BufferWatermark3Word` = `2U`,
   
  `kDAC_BufferWatermark4Word` = `3U` }
   
*DAC buffer watermark.*
- enum `dac_buffer_work_mode_t` {
   
  `kDAC_BufferWorkAsNormalMode` = `0U`,
   
  `kDAC_BufferWorkAsSwingMode`,
   
  `kDAC_BufferWorkAsOneTimeScanMode`,
   
  `kDAC_BufferWorkAsFIFOMode` }
   
*DAC buffer work mode.*

## Driver version

- #define `FSL_DAC_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 1)`)
   
*DAC driver version 2.0.1.*

## Initialization

- void `DAC_Init` (`DAC_Type` \*base, const `dac_config_t` \*config)
   
*Initializes the DAC module.*
- void `DAC_Deinit` (`DAC_Type` \*base)
   
*De-initializes the DAC module.*
- void `DAC_GetDefaultConfig` (`dac_config_t` \*config)
   
*Initializes the DAC user configuration structure.*
- static void `DAC_Enable` (`DAC_Type` \*base, bool enable)
   
*Enables the DAC module.*

## Data Structure Documentation

### Buffer

- static void **DAC\_EnableBuffer** (DAC\_Type \*base, bool enable)  
*Enables the DAC buffer.*
- void **DAC\_SetBufferConfig** (DAC\_Type \*base, const dac\_buffer\_config\_t \*config)  
*Configures the CMP buffer.*
- void **DAC\_GetDefaultBufferConfig** (dac\_buffer\_config\_t \*config)  
*Initializes the DAC buffer configuration structure.*
- static void **DAC\_EnableBufferDMA** (DAC\_Type \*base, bool enable)  
*Enables the DMA for DAC buffer.*
- void **DAC\_SetBufferValue** (DAC\_Type \*base, uint8\_t index, uint16\_t value)  
*Sets the value for items in the buffer.*
- static void **DAC\_DoSoftwareTriggerBuffer** (DAC\_Type \*base)  
*Triggers the buffer using software and updates the read pointer of the DAC buffer.*
- static uint8\_t **DAC\_GetBufferReadPointer** (DAC\_Type \*base)  
*Gets the current read pointer of the DAC buffer.*
- void **DAC\_SetBufferReadPointer** (DAC\_Type \*base, uint8\_t index)  
*Sets the current read pointer of the DAC buffer.*
- void **DAC\_EnableBufferInterrupts** (DAC\_Type \*base, uint32\_t mask)  
*Enables interrupts for the DAC buffer.*
- void **DAC\_DisableBufferInterrupts** (DAC\_Type \*base, uint32\_t mask)  
*Disables interrupts for the DAC buffer.*
- uint32\_t **DAC\_GetBufferStatusFlags** (DAC\_Type \*base)  
*Gets the flags of events for the DAC buffer.*
- void **DAC\_ClearBufferStatusFlags** (DAC\_Type \*base, uint32\_t mask)  
*Clears the flags of events for the DAC buffer.*

### 9.3 Data Structure Documentation

#### 9.3.1 struct dac\_config\_t

##### Data Fields

- **dac\_reference\_voltage\_source\_t referenceVoltageSource**  
*Select the DAC reference voltage source.*
- **bool enableLowPowerMode**  
*Enable the low-power mode.*

##### 9.3.1.0.0.10 Field Documentation

###### 9.3.1.0.0.10.1 **dac\_reference\_voltage\_source\_t dac\_config\_t::referenceVoltageSource**

###### 9.3.1.0.0.10.2 **bool dac\_config\_t::enableLowPowerMode**

#### 9.3.2 struct dac\_buffer\_config\_t

##### Data Fields

- **dac\_buffer\_trigger\_mode\_t triggerMode**  
*Select the buffer's trigger mode.*

- **dac\_buffer\_watermark\_t watermark**  
Select the buffer's watermark.
- **dac\_buffer\_work\_mode\_t workMode**  
Select the buffer's work mode.
- **uint8\_t upperLimit**  
Set the upper limit for the buffer index.

### 9.3.2.0.0.11 Field Documentation

**9.3.2.0.0.11.1 dac\_buffer\_trigger\_mode\_t dac\_buffer\_config\_t::triggerMode**

**9.3.2.0.0.11.2 dac\_buffer\_watermark\_t dac\_buffer\_config\_t::watermark**

**9.3.2.0.0.11.3 dac\_buffer\_work\_mode\_t dac\_buffer\_config\_t::workMode**

**9.3.2.0.0.11.4 uint8\_t dac\_buffer\_config\_t::upperLimit**

Normally, 0-15 is available for a buffer with 16 items.

## 9.4 Macro Definition Documentation

**9.4.1 #define FSL\_DAC\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))**

## 9.5 Enumeration Type Documentation

### 9.5.1 enum \_dac\_buffer\_status\_flags

Enumerator

**kDAC\_BufferWatermarkFlag** DAC Buffer Watermark Flag.

**kDAC\_BufferReadPointerTopPositionFlag** DAC Buffer Read Pointer Top Position Flag.

**kDAC\_BufferReadPointerBottomPositionFlag** DAC Buffer Read Pointer Bottom Position Flag.

### 9.5.2 enum \_dac\_buffer\_interrupt\_enable

Enumerator

**kDAC\_BufferWatermarkInterruptEnable** DAC Buffer Watermark Interrupt Enable.

**kDAC\_BufferReadPointerTopInterruptEnable** DAC Buffer Read Pointer Top Flag Interrupt Enable.

**kDAC\_BufferReadPointerBottomInterruptEnable** DAC Buffer Read Pointer Bottom Flag Interrupt Enable.

## Function Documentation

### 9.5.3 enum dac\_reference\_voltage\_source\_t

Enumerator

*kDAC\_ReferenceVoltageSourceVref1* The DAC selects DACREF\_1 as the reference voltage.

*kDAC\_ReferenceVoltageSourceVref2* The DAC selects DACREF\_2 as the reference voltage.

### 9.5.4 enum dac\_buffer\_trigger\_mode\_t

Enumerator

*kDAC\_BufferTriggerByHardwareMode* The DAC hardware trigger is selected.

*kDAC\_BufferTriggerBySoftwareMode* The DAC software trigger is selected.

### 9.5.5 enum dac\_buffer\_watermark\_t

Enumerator

*kDAC\_BufferWatermark1Word* 1 word away from the upper limit.

*kDAC\_BufferWatermark2Word* 2 words away from the upper limit.

*kDAC\_BufferWatermark3Word* 3 words away from the upper limit.

*kDAC\_BufferWatermark4Word* 4 words away from the upper limit.

### 9.5.6 enum dac\_buffer\_work\_mode\_t

Enumerator

*kDAC\_BufferWorkAsNormalMode* Normal mode.

*kDAC\_BufferWorkAsSwingMode* Swing mode.

*kDAC\_BufferWorkAsOneTimeScanMode* One-Time Scan mode.

*kDAC\_BufferWorkAsFIFOMode* FIFO mode.

## 9.6 Function Documentation

### 9.6.1 void DAC\_Init ( DAC\_Type \* *base*, const dac\_config\_t \* *config* )

This function initializes the DAC module including the following operations.

- Enabling the clock for DAC module.
- Configuring the DAC converter with a user configuration.
- Enabling the DAC module.

Parameters

<i>base</i>	DAC peripheral base address.
<i>config</i>	Pointer to the configuration structure. See "dac_config_t".

### 9.6.2 void DAC\_Deinit ( DAC\_Type \* *base* )

This function de-initializes the DAC module including the following operations.

- Disabling the DAC module.
- Disabling the clock for the DAC module.

Parameters

<i>base</i>	DAC peripheral base address.
-------------	------------------------------

### 9.6.3 void DAC\_GetDefaultConfig ( dac\_config\_t \* *config* )

This function initializes the user configuration structure to a default value. The default values are as follows.

```
* config->referenceVoltageSource = kDAC_ReferenceVoltageSourceVref2;
* config->enableLowPowerMode = false;
*
```

Parameters

<i>config</i>	Pointer to the configuration structure. See "dac_config_t".
---------------	---

### 9.6.4 static void DAC\_Enable ( DAC\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

<i>base</i>	DAC peripheral base address.
-------------	------------------------------

## Function Documentation

<i>enable</i>	Enables or disables the feature.
---------------	----------------------------------

### 9.6.5 static void DAC\_EnableBuffer ( DAC\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

<i>base</i>	DAC peripheral base address.
<i>enable</i>	Enables or disables the feature.

### 9.6.6 void DAC\_SetBufferConfig ( DAC\_Type \* *base*, const dac\_buffer\_config\_t \* *config* )

Parameters

<i>base</i>	DAC peripheral base address.
<i>config</i>	Pointer to the configuration structure. See "dac_buffer_config_t".

### 9.6.7 void DAC\_GetDefaultBufferConfig ( dac\_buffer\_config\_t \* *config* )

This function initializes the DAC buffer configuration structure to default values. The default values are as follows.

```
* config->triggerMode = kDAC_BufferTriggerBySoftwareMode;
* config->watermark   = kDAC_BufferWatermark1Word;
* config->workMode    = kDAC_BufferWorkAsNormalMode;
* config->upperLimit  = DAC_DATL_COUNT - 1U;
*
```

Parameters

<i>config</i>	Pointer to the configuration structure. See "dac_buffer_config_t".
---------------	--

### 9.6.8 static void DAC\_EnableBufferDMA ( DAC\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

<i>base</i>	DAC peripheral base address.
<i>enable</i>	Enables or disables the feature.

### 9.6.9 void DAC\_SetBufferValue ( DAC\_Type \* *base*, uint8\_t *index*, uint16\_t *value* )

Parameters

<i>base</i>	DAC peripheral base address.
<i>index</i>	Setting the index for items in the buffer. The available index should not exceed the size of the DAC buffer.
<i>value</i>	Setting the value for items in the buffer. 12-bits are available.

### 9.6.10 static void DAC\_DoSoftwareTriggerBuffer ( DAC\_Type \* *base* ) [inline], [static]

This function triggers the function using software. The read pointer of the DAC buffer is updated with one step after this function is called. Changing the read pointer depends on the buffer's work mode.

Parameters

<i>base</i>	DAC peripheral base address.
-------------	------------------------------

### 9.6.11 static uint8\_t DAC\_GetBufferReadPointer ( DAC\_Type \* *base* ) [inline], [static]

This function gets the current read pointer of the DAC buffer. The current output value depends on the item indexed by the read pointer. It is updated either by a software trigger or a hardware trigger.

Parameters

<i>base</i>	DAC peripheral base address.
-------------	------------------------------

Returns

The current read pointer of the DAC buffer.

## Function Documentation

### 9.6.12 void DAC\_SetBufferReadPointer ( DAC\_Type \* *base*, uint8\_t *index* )

This function sets the current read pointer of the DAC buffer. The current output value depends on the item indexed by the read pointer. It is updated either by a software trigger or a hardware trigger. After the read pointer changes, the DAC output value also changes.

Parameters

<i>base</i>	DAC peripheral base address.
<i>index</i>	Setting an index value for the pointer.

### 9.6.13 void DAC\_EnableBufferInterrupts ( DAC\_Type \* *base*, uint32\_t *mask* )

Parameters

<i>base</i>	DAC peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_dac_buffer_interrupt_enable".

### 9.6.14 void DAC\_DisableBufferInterrupts ( DAC\_Type \* *base*, uint32\_t *mask* )

Parameters

<i>base</i>	DAC peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_dac_buffer_interrupt_enable".

### 9.6.15 uint32\_t DAC\_GetBufferStatusFlags ( DAC\_Type \* *base* )

Parameters

<i>base</i>	DAC peripheral base address.
-------------	------------------------------

Returns

Mask value for the asserted flags. See "\_dac\_buffer\_status\_flags".

### 9.6.16 void DAC\_ClearBufferStatusFlags ( DAC\_Type \* *base*, uint32\_t *mask* )

## Parameters

<i>base</i>	DAC peripheral base address.
<i>mask</i>	Mask value for flags. See "_dac_buffer_status_flags_t".

## Function Documentation

# Chapter 10

## DMAMUX: Direct Memory Access Multiplexer Driver

### 10.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Direct Memory Access Multiplexer (DMAMUX) of MCUXpresso SDK devices.

### 10.2 Typical use case

#### 10.2.1 DMAMUX Operation

```
DMAMUX_Init(DMAMUX0);
DMAMUX_SetSource(DMAMUX0, channel, source);
DMAMUX_EnableChannel(DMAMUX0, channel);
...
DMAMUX_DisableChannel(DMAMUX, channel);
DMAMUX_Deinit(DMAMUX0);
```

### Driver version

- #define **FSL\_DMAMUX\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 2))  
*DMAMUX driver version 2.0.2.*

### DMAMUX Initialization and de-initialization

- void **DMAMUX\_Init** (DMAMUX\_Type \*base)  
*Initializes the DMAMUX peripheral.*
- void **DMAMUX\_Deinit** (DMAMUX\_Type \*base)  
*Deinitializes the DMAMUX peripheral.*

### DMAMUX Channel Operation

- static void **DMAMUX\_EnableChannel** (DMAMUX\_Type \*base, uint32\_t channel)  
*Enables the DMAMUX channel.*
- static void **DMAMUX\_DisableChannel** (DMAMUX\_Type \*base, uint32\_t channel)  
*Disables the DMAMUX channel.*
- static void **DMAMUX\_SetSource** (DMAMUX\_Type \*base, uint32\_t channel, uint32\_t source)  
*Configures the DMAMUX channel source.*
- static void **DMAMUX\_EnablePeriodTrigger** (DMAMUX\_Type \*base, uint32\_t channel)  
*Enables the DMAMUX period trigger.*
- static void **DMAMUX\_DisablePeriodTrigger** (DMAMUX\_Type \*base, uint32\_t channel)  
*Disables the DMAMUX period trigger.*

### 10.3 Macro Definition Documentation

#### 10.3.1 #define **FSL\_DMAMUX\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 2))

## Function Documentation

### 10.4 Function Documentation

#### 10.4.1 void DMAMUX\_Init ( **DMAMUX\_Type** \* *base* )

This function ungates the DMAMUX clock.

Parameters

<i>base</i>	DMAMUX peripheral base address.
-------------	---------------------------------

#### 10.4.2 void DMAMUX\_Deinit ( DMAMUX\_Type \* *base* )

This function gates the DMAMUX clock.

Parameters

<i>base</i>	DMAMUX peripheral base address.
-------------	---------------------------------

#### 10.4.3 static void DMAMUX\_EnableChannel ( DMAMUX\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

This function enables the DMAMUX channel.

Parameters

<i>base</i>	DMAMUX peripheral base address.
<i>channel</i>	DMAMUX channel number.

#### 10.4.4 static void DMAMUX\_DisableChannel ( DMAMUX\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

This function disables the DMAMUX channel.

Note

The user must disable the DMAMUX channel before configuring it.

Parameters

<i>base</i>	DMAMUX peripheral base address.
-------------	---------------------------------

## Function Documentation

<i>channel</i>	DMAMUX channel number.
----------------	------------------------

### 10.4.5 static void DMAMUX\_SetSource ( DMAMUX\_Type \* *base*, uint32\_t *channel*, uint32\_t *source* ) [inline], [static]

Parameters

<i>base</i>	DMAMUX peripheral base address.
<i>channel</i>	DMAMUX channel number.
<i>source</i>	Channel source, which is used to trigger the DMA transfer.

### 10.4.6 static void DMAMUX\_EnablePeriodTrigger ( DMAMUX\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

This function enables the DMAMUX period trigger feature.

Parameters

<i>base</i>	DMAMUX peripheral base address.
<i>channel</i>	DMAMUX channel number.

### 10.4.7 static void DMAMUX\_DisablePeriodTrigger ( DMAMUX\_Type \* *base*, uint32\_t *channel* ) [inline], [static]

This function disables the DMAMUX period trigger.

Parameters

<i>base</i>	DMAMUX peripheral base address.
<i>channel</i>	DMAMUX channel number.

# Chapter 11

## DSPI: Serial Peripheral Interface Driver

### 11.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Serial Peripheral Interface (SPI) module of MCUXpresso SDK devices.

### Modules

- DSPI DMA Driver
- DSPI Driver
- DSPI FreeRTOS Driver
- DSPI eDMA Driver

## DSPI Driver

### 11.2 DSPI Driver

#### 11.2.1 Overview

This section describes the programming interface of the DSPI Peripheral driver. The DSPI driver configures the DSPI module and provides the functional and transactional interfaces to build the DSPI application.

#### 11.2.2 Typical use case

##### 11.2.2.1 Master Operation

```
dspi_master_handle_t g_m_handle; //global variable
dspi_master_config_t masterConfig;
masterConfig.whichCtar = kDSPI_Ctar0;
masterConfig.ctarConfig.baudRate = baudrate;
masterConfig.ctarConfig.bitsPerFrame = 8;
masterConfig.ctarConfig.cpol = kDSPI_ClockPolarityActiveHigh;
masterConfig.ctarConfig.cpha = kDSPI_ClockPhaseFirstEdge;
masterConfig.ctarConfig.direction = kDSPI_MsbFirst;
masterConfig.ctarConfig.pcsToSckDelayInNanoSec = 1000000000 / baudrate;
masterConfig.ctarConfig.lastSckToPcsDelayInNanoSec = 1000000000 / baudrate;
masterConfig.ctarConfig.betweenTransferDelayInNanoSec = 1000000000 / baudrate;
masterConfig.whichPcs = kDSPI_Pcs0;
masterConfig.pcsActiveHighOrLow = kDSPI_PcsActiveLow;
masterConfig.enableContinuousSCK = false;
masterConfig.enableRx_fifoOverWrite = false;
masterConfig.enableModifiedTimingFormat = false;
masterConfig.samplePoint = kDSPI_SckToSin0Clock;
DSPI_MasterInit(base, &masterConfig, srcClock_Hz);

//srcClock_Hz = CLOCK_GetFreq(xxx);
DSPI_MasterInit(base, &masterConfig, srcClock_Hz);

DSPI_MasterTransferCreateHandle(base, &g_m_handle, NULL, NULL);

masterXfer.txData = masterSendBuffer;
masterXfer.rxData = masterReceiveBuffer;
masterXfer.dataSize = transfer_dataSize;
masterXfer.configFlags = kDSPI_MasterCtar0 | kDSPI_MasterPcs0 ;
DSPI_MasterTransferBlocking(base, &g_m_handle, &masterXfer);
```

##### 11.2.2.2 Slave Operation

```
dspi_slave_handle_t g_s_handle;//global variable
/*Slave config*/
slaveConfig.whichCtar = kDSPI_Ctar0;
slaveConfig.ctarConfig.bitsPerFrame = 8;
slaveConfig.ctarConfig.cpol = kDSPI_ClockPolarityActiveHigh;
slaveConfig.ctarConfig.cpha = kDSPI_ClockPhaseFirstEdge;
```

```

slaveConfig.enableContinuousSCK      = false;
slaveConfig.enableRxFifoOverWrite   = false;
slaveConfig.enableModifiedTimingFormat = false;
slaveConfig.samplePoint             = kDSPI_SckToSin0Clock;
DSPI_SlaveInit(base, &slaveConfig);

slaveXfer.txData      = slaveSendBuffer0;
slaveXfer.rxData      = slaveReceiveBuffer0;
slaveXfer.dataSize    = transfer_dataSize;
slaveXfer.configFlags = kDSPI_SlaveCtar0;

bool isTransferCompleted = false;
DSPI_SlaveTransferCreateHandle(base, &g_s_handle, DSPI_SlaveUserCallback, &
                               isTransferCompleted);

DSPI_SlaveTransferNonBlocking(&g_s_handle, &slaveXfer);

//void DSPI_SlaveUserCallback(SPI_Type *base, dspi_slave_handle_t *handle, status_t status, void
//                           *isTransferCompleted)
//{
//    if (status == kStatus_Success)
//    {
//        __NOP();
//    }
//    else if (status == kStatus_DSPI_Error)
//    {
//        __NOP();
//    }
//    *(bool *)isTransferCompleted = true;
//    PRINTF("This is DSPI slave call back . \r\n");
//}

```

## Data Structures

- struct [dsPIC\\_Command\\_Data\\_Config\\_t](#)  
*DSPI master command date configuration used for the SPIx\_PUSHR.* [More...](#)
- struct [dsPIC\\_Master\\_Ctar\\_Config\\_t](#)  
*DSPI master ctar configuration structure.* [More...](#)
- struct [dsPIC\\_Master\\_Config\\_t](#)  
*DSPI master configuration structure.* [More...](#)
- struct [dsPIC\\_Slave\\_Ctar\\_Config\\_t](#)  
*DSPI slave ctar configuration structure.* [More...](#)
- struct [dsPIC\\_Slave\\_Config\\_t](#)  
*DSPI slave configuration structure.* [More...](#)
- struct [dsPIC\\_Transfer\\_t](#)  
*DSPI master/slave transfer structure.* [More...](#)
- struct [dsPIC\\_Master\\_Handle\\_t](#)  
*DSPI master transfer handle structure used for transactional API.* [More...](#)
- struct [dsPIC\\_Slave\\_Handle\\_t](#)  
*DSPI slave transfer handle structure used for the transactional API.* [More...](#)

## Macros

- #define [DSPI\\_DUMMY\\_DATA](#) (0x00U)

## DSPI Driver

- `#define DSPI_MASTER_CTAR_SHIFT (0U)`  
*DSPI master CTAR shift macro; used internally.*
- `#define DSPI_MASTER_CTAR_MASK (0x0FU)`  
*DSPI master CTAR mask macro; used internally.*
- `#define DSPI_MASTER_PCS_SHIFT (4U)`  
*DSPI master PCS shift macro; used internally.*
- `#define DSPI_MASTER_PCS_MASK (0xF0U)`  
*DSPI master PCS mask macro; used internally.*
- `#define DSPI_SLAVE_CTAR_SHIFT (0U)`  
*DSPI slave CTAR shift macro; used internally.*
- `#define DSPI_SLAVE_CTAR_MASK (0x07U)`  
*DSPI slave CTAR mask macro; used internally.*

## Typedefs

- `typedef void(* dspi_master_transfer_callback_t )(SPI_Type *base, dspi_master_handle_t *handle, status_t status, void *userData)`  
*Completion callback function pointer type.*
- `typedef void(* dspi_slave_transfer_callback_t )(SPI_Type *base, dspi_slave_handle_t *handle, status_t status, void *userData)`  
*Completion callback function pointer type.*

## Enumerations

- `enum _dspi_status {`  
    `kStatus_DSPI_Busy = MAKE_STATUS(kStatusGroup_DSPI, 0),`  
    `kStatus_DSPI_Error = MAKE_STATUS(kStatusGroup_DSPI, 1),`  
    `kStatus_DSPI_Idle = MAKE_STATUS(kStatusGroup_DSPI, 2),`  
    `kStatus_DSPI_OutOfRange = MAKE_STATUS(kStatusGroup_DSPI, 3) }`  
*Status for the DSPI driver.*
- `enum _dspi_flags {`  
    `kDSPI_TxCompleteFlag = SPI_SR_TCF_MASK,`  
    `kDSPI_EndOfQueueFlag = SPI_SR_EOQF_MASK,`  
    `kDSPI_TxFifoUnderflowFlag = SPI_SR_TFUF_MASK,`  
    `kDSPI_TxFifoFillRequestFlag = SPI_SR_TFFF_MASK,`  
    `kDSPI_RxFifoOverflowFlag = SPI_SR_RFOF_MASK,`  
    `kDSPI_RxFifoDrainRequestFlag = SPI_SR_RFDF_MASK,`  
    `kDSPI_TxAndRxStatusFlag = SPI_SR_TXRXS_MASK,`  
    `kDSPI_AllStatusFlag }`  
*DSPI status flags in SPIx\_SR register.*
- `enum _dspi_interrupt_enable {`

- ```

kDSPI_TxCompleteInterruptEnable = SPI_RSER_TCF_RE_MASK,
kDSPI_EndOfQueueInterruptEnable = SPI_RSER_EOQF_RE_MASK,
kDSPI_TxFifoUnderflowInterruptEnable = SPI_RSER_TFUF_RE_MASK,
kDSPI_TxFifoFillRequestInterruptEnable = SPI_RSER_TFFF_RE_MASK,
kDSPI_RxFifoOverflowInterruptEnable = SPI_RSER_RFOF_RE_MASK,
kDSPI_RxFifoDrainRequestInterruptEnable = SPI_RSER_RFDF_RE_MASK,
kDSPI_AllInterruptEnable }

DSPI interrupt source.
• enum _dspi_dma_enable {
  kDSPI_TxDmaEnable = (SPI_RSER_TFFF_RE_MASK | SPI_RSER_TFFF_DIRS_MASK),
  kDSPI_RxDmaEnable = (SPI_RSER_RFDF_RE_MASK | SPI_RSER_RFDF_DIRS_MASK) }

DSPI DMA source.
• enum dspi_master_slave_mode_t {
  kDSPI_Master = 1U,
  kDSPI_Slave = 0U }

DSPI master or slave mode configuration.
• enum dspi_master_sample_point_t {
  kDSPI_SckToSin0Clock = 0U,
  kDSPI_SckToSin1Clock = 1U,
  kDSPI_SckToSin2Clock = 2U }

DSPI Sample Point: Controls when the DSPI master samples SIN in the Modified Transfer Format.
• enum dspi_which_pcs_t {
  kDSPI_Pcs0 = 1U << 0,
  kDSPI_Pcs1 = 1U << 1,
  kDSPI_Pcs2 = 1U << 2,
  kDSPI_Pcs3 = 1U << 3,
  kDSPI_Pcs4 = 1U << 4,
  kDSPI_Pcs5 = 1U << 5 }

DSPI Peripheral Chip Select (Pcs) configuration (which Pcs to configure).
• enum dspi_pcs_polarity_config_t {
  kDSPI_PcsActiveHigh = 0U,
  kDSPI_PcsActiveLow = 1U }

DSPI Peripheral Chip Select (Pcs) Polarity configuration.
• enum _dspi_pcs_polarity {
  kDSPI_Pcs0ActiveLow = 1U << 0,
  kDSPI_Pcs1ActiveLow = 1U << 1,
  kDSPI_Pcs2ActiveLow = 1U << 2,
  kDSPI_Pcs3ActiveLow = 1U << 3,
  kDSPI_Pcs4ActiveLow = 1U << 4,
  kDSPI_Pcs5ActiveLow = 1U << 5,
  kDSPI_PcsAllActiveLow = 0xFFU }

DSPI Peripheral Chip Select (Pcs) Polarity.
• enum dspi_clock_polarity_t {
  kDSPI_ClockPolarityActiveHigh = 0U,
  kDSPI_ClockPolarityActiveLow = 1U }

DSPI clock polarity configuration for a given CTAR.
• enum dspi_clock_phase_t {

```

## DSPI Driver

```
kDSPI_ClockPhaseFirstEdge = 0U,  
kDSPI_ClockPhaseSecondEdge = 1U }
```

*DSPI clock phase configuration for a given CTAR.*

- enum `dspi_shift_direction_t` {

```
kDSPI_MsbFirst = 0U,  
kDSPI_LsbFirst = 1U }
```

*DSPI data shifter direction options for a given CTAR.*

- enum `dspi_delay_type_t` {

```
kDSPI_PcsToSck = 1U,  
kDSPI_LastSckToPcs,  
kDSPI_BetweenTransfer }
```

*DSPI delay type selection.*

- enum `dspi_ctar_selection_t` {

```
kDSPI_Ctar0 = 0U,  
kDSPI_Ctar1 = 1U,  
kDSPI_Ctar2 = 2U,  
kDSPI_Ctar3 = 3U,  
kDSPI_Ctar4 = 4U,  
kDSPI_Ctar5 = 5U,  
kDSPI_Ctar6 = 6U,  
kDSPI_Ctar7 = 7U }
```

*DSPI Clock and Transfer Attributes Register (CTAR) selection.*

- enum `_dspi_transfer_config_flag_for_master` {

```
kDSPI_MasterCtar0 = 0U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterCtar1 = 1U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterCtar2 = 2U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterCtar3 = 3U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterCtar4 = 4U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterCtar5 = 5U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterCtar6 = 6U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterCtar7 = 7U << DSPI_MASTER_CTAR_SHIFT,  
kDSPI_MasterPcs0 = 0U << DSPI_MASTER_PCS_SHIFT,  
kDSPI_MasterPcs1 = 1U << DSPI_MASTER_PCS_SHIFT,  
kDSPI_MasterPcs2 = 2U << DSPI_MASTER_PCS_SHIFT,  
kDSPI_MasterPcs3 = 3U << DSPI_MASTER_PCS_SHIFT,  
kDSPI_MasterPcs4 = 4U << DSPI_MASTER_PCS_SHIFT,  
kDSPI_MasterPcs5 = 5U << DSPI_MASTER_PCS_SHIFT,  
kDSPI_MasterPcsContinuous = 1U << 20,  
kDSPI_MasterActiveAfterTransfer = 1U << 21 }
```

*Use this enumeration for the DSPI master transfer configFlags.*

- enum `_dspi_transfer_config_flag_for_slave` { `kDSPI_SlaveCtar0` = 0U << DSPI\_SLAVE\_CTAR\_SHIFT }

*Use this enumeration for the DSPI slave transfer configFlags.*

- enum `_dspi_transfer_state` {

```
kDSPI_Idle = 0x0U,  
kDSPI_Busy,
```

```
kDSPI_Error }
```

*DSPI transfer state, which is used for DSPI transactional API state machine.*

## Driver version

- #define **FSL\_DSPI\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 4))  
*DSPI driver version 2.1.4.*

## Initialization and deinitialization

- void **DSPI\_MasterInit** (SPI\_Type \*base, const **dspi\_master\_config\_t** \*masterConfig, uint32\_t srcClock\_Hz)  
*Initializes the DSPI master.*
- void **DSPI\_MasterGetDefaultConfig** (**dspi\_master\_config\_t** \*masterConfig)  
*Sets the **dspi\_master\_config\_t** structure to default values.*
- void **DSPI\_SlaveInit** (SPI\_Type \*base, const **dspi\_slave\_config\_t** \*slaveConfig)  
*DSPI slave configuration.*
- void **DSPI\_SlaveGetDefaultConfig** (**dspi\_slave\_config\_t** \*slaveConfig)  
*Sets the **dspi\_slave\_config\_t** structure to a default value.*
- void **DSPI\_Deinit** (SPI\_Type \*base)  
*De-initializes the DSPI peripheral.*
- static void **DSPI\_Enable** (SPI\_Type \*base, bool enable)  
*Enables the DSPI peripheral and sets the MCR MDIS to 0.*

## Status

- static uint32\_t **DSPI\_GetStatusFlags** (SPI\_Type \*base)  
*Gets the DSPI status flag state.*
- static void **DSPI\_ClearStatusFlags** (SPI\_Type \*base, uint32\_t statusFlags)  
*Clears the DSPI status flag.*

## Interrupts

- void **DSPI\_EnableInterrupts** (SPI\_Type \*base, uint32\_t mask)  
*Enables the DSPI interrupts.*
- static void **DSPI\_DisableInterrupts** (SPI\_Type \*base, uint32\_t mask)  
*Disables the DSPI interrupts.*

## DMA Control

- static void **DSPI\_EnableDMA** (SPI\_Type \*base, uint32\_t mask)  
*Enables the DSPI DMA request.*
- static void **DSPI\_DisableDMA** (SPI\_Type \*base, uint32\_t mask)  
*Disables the DSPI DMA request.*

## DSPI Driver

- static uint32\_t **DSPI\_MasterGetTxRegisterAddress** (SPI\_Type \*base)  
*Gets the DSPI master PUSHR data register address for the DMA operation.*
- static uint32\_t **DSPI\_SlaveGetTxRegisterAddress** (SPI\_Type \*base)  
*Gets the DSPI slave PUSHR data register address for the DMA operation.*
- static uint32\_t **DSPI\_GetRxRegisterAddress** (SPI\_Type \*base)  
*Gets the DSPI POPR data register address for the DMA operation.*

## Bus Operations

- static void **DSPI\_SetMasterSlaveMode** (SPI\_Type \*base, **dspi\_master\_slave\_mode\_t** mode)  
*Configures the DSPI for master or slave.*
- static bool **DSPI\_IsMaster** (SPI\_Type \*base)  
*Returns whether the DSPI module is in master mode.*
- static void **DSPI\_StartTransfer** (SPI\_Type \*base)  
*Starts the DSPI transfers and clears HALT bit in MCR.*
- static void **DSPI\_StopTransfer** (SPI\_Type \*base)  
*Stops DSPI transfers and sets the HALT bit in MCR.*
- static void **DSPI\_SetFifoEnable** (SPI\_Type \*base, bool enableTxFifo, bool enableRxFifo)  
*Enables or disables the DSPI FIFOs.*
- static void **DSPI\_FlushFifo** (SPI\_Type \*base, bool flushTxFifo, bool flushRxFifo)  
*Flushes the DSPI FIFOs.*
- static void **DSPI\_SetAllPcsPolarity** (SPI\_Type \*base, uint32\_t mask)  
*Configures the DSPI peripheral chip select polarity simultaneously.*
- uint32\_t **DSPI\_MasterSetBaudRate** (SPI\_Type \*base, **dspi\_ctar\_selection\_t** whichCtar, uint32\_t baudRate\_Bps, uint32\_t srcClock\_Hz)  
*Sets the DSPI baud rate in bits per second.*
- void **DSPI\_MasterSetDelayScaler** (SPI\_Type \*base, **dspi\_ctar\_selection\_t** whichCtar, uint32\_t prescaler, uint32\_t scaler, **dspi\_delay\_type\_t** whichDelay)  
*Manually configures the delay prescaler and scaler for a particular CTAR.*
- uint32\_t **DSPI\_MasterSetDelayTimes** (SPI\_Type \*base, **dspi\_ctar\_selection\_t** whichCtar, **dspi\_delay\_type\_t** whichDelay, uint32\_t srcClock\_Hz, uint32\_t delayTimeInNanoSec)  
*Calculates the delay prescaler and scaler based on the desired delay input in nanoseconds.*
- static void **DSPI\_MasterWriteData** (SPI\_Type \*base, **dspi\_command\_data\_config\_t** \*command, uint16\_t data)  
*Writes data into the data buffer for master mode.*
- void **DSPI\_GetDefaultDataCommandConfig** (**dspi\_command\_data\_config\_t** \*command)  
*Sets the **dspi\_command\_data\_config\_t** structure to default values.*
- void **DSPI\_MasterWriteDataBlocking** (SPI\_Type \*base, **dspi\_command\_data\_config\_t** \*command, uint16\_t data)  
*Writes data into the data buffer master mode and waits till complete to return.*
- static uint32\_t **DSPI\_MasterGetFormattedCommand** (**dspi\_command\_data\_config\_t** \*command)  
*Returns the DSPI command word formatted to the PUSHR data register bit field.*
- void **DSPI\_MasterWriteCommandDataBlocking** (SPI\_Type \*base, uint32\_t data)  
*Writes a 32-bit data word (16-bit command appended with 16-bit data) into the data buffer master mode and waits till complete to return.*
- static void **DSPI\_SlaveWriteData** (SPI\_Type \*base, uint32\_t data)  
*Writes data into the data buffer in slave mode.*
- void **DSPI\_SlaveWriteDataBlocking** (SPI\_Type \*base, uint32\_t data)  
*Writes data into the data buffer in slave mode, waits till data was transmitted, and returns.*

- static uint32\_t **DSPI\_ReadData** (SPI\_Type \*base)  
*Reads data from the data buffer.*

## Transactional

- void **DSPI\_MasterTransferCreateHandle** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle, **dspi\_master\_transfer\_callback\_t** callback, void \*userData)  
*Initializes the DSPI master handle.*
- status\_t **DSPI\_MasterTransferBlocking** (SPI\_Type \*base, **dspi\_transfer\_t** \*transfer)  
*DSPI master transfer data using polling.*
- status\_t **DSPI\_MasterTransferNonBlocking** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle, **dspi\_transfer\_t** \*transfer)  
*DSPI master transfer data using interrupts.*
- status\_t **DSPI\_MasterTransferGetCount** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle, size\_t \*count)  
*Gets the master transfer count.*
- void **DSPI\_MasterTransferAbort** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle)  
*DSPI master aborts a transfer using an interrupt.*
- void **DSPI\_MasterTransferHandleIRQ** (SPI\_Type \*base, dspi\_master\_handle\_t \*handle)  
*DSPI Master IRQ handler function.*
- void **DSPI\_SlaveTransferCreateHandle** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle, **dspi\_slave\_transfer\_callback\_t** callback, void \*userData)  
*Initializes the DSPI slave handle.*
- status\_t **DSPI\_SlaveTransferNonBlocking** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle, **dspi\_transfer\_t** \*transfer)  
*DSPI slave transfers data using an interrupt.*
- status\_t **DSPI\_SlaveTransferGetCount** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle, size\_t \*count)  
*Gets the slave transfer count.*
- void **DSPI\_SlaveTransferAbort** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle)  
*DSPI slave aborts a transfer using an interrupt.*
- void **DSPI\_SlaveTransferHandleIRQ** (SPI\_Type \*base, dspi\_slave\_handle\_t \*handle)  
*DSPI Master IRQ handler function.*

### 11.2.3 Data Structure Documentation

#### 11.2.3.1 struct dspi\_command\_data\_config\_t

##### Data Fields

- bool **isPcsContinuous**  
*Option to enable the continuous assertion of the chip select between transfers.*
- **dspi\_ctar\_selection\_t whichCtar**  
*The desired Clock and Transfer Attributes Register (CTAR) to use for CTAS.*
- **dspi\_which\_pcs\_t whichPcs**  
*The desired PCS signal to use for the data transfer.*
- bool **isEndOfQueue**

## DSPI Driver

*Signals that the current transfer is the last in the queue.*

- **bool clearTransferCount**

*Clears the SPI Transfer Counter (SPI\_TCNT) before transmission starts.*

### 11.2.3.1.0.12 Field Documentation

#### 11.2.3.1.0.12.1 **bool dspi\_command\_data\_config\_t::isPcsContinuous**

#### 11.2.3.1.0.12.2 **dspi\_ctar\_selection\_t dspi\_command\_data\_config\_t::whichCtar**

#### 11.2.3.1.0.12.3 **dspi\_which\_pcs\_t dspi\_command\_data\_config\_t::whichPcs**

#### 11.2.3.1.0.12.4 **bool dspi\_command\_data\_config\_t::isEndOfQueue**

#### 11.2.3.1.0.12.5 **bool dspi\_command\_data\_config\_t::clearTransferCount**

### 11.2.3.2 **struct dspi\_master\_ctar\_config\_t**

#### Data Fields

- **uint32\_t baudRate**  
*Baud Rate for DSPI.*
- **uint32\_t bitsPerFrame**  
*Bits per frame, minimum 4, maximum 16.*
- **dspi\_clock\_polarity\_t cpol**  
*Clock polarity.*
- **dspi\_clock\_phase\_t cpha**  
*Clock phase.*
- **dspi\_shift\_direction\_t direction**  
*MSB or LSB data shift direction.*
- **uint32\_t pcsToSckDelayInNanoSec**  
*PCS to SCK delay time in nanoseconds; setting to 0 sets the minimum delay.*
- **uint32\_t lastSckToPcsDelayInNanoSec**  
*The last SCK to PCS delay time in nanoseconds; setting to 0 sets the minimum delay.*
- **uint32\_t betweenTransferDelayInNanoSec**  
*After the SCK delay time in nanoseconds; setting to 0 sets the minimum delay.*

### 11.2.3.2.0.13 Field Documentation

**11.2.3.2.0.13.1 uint32\_t dspi\_master\_ctar\_config\_t::baudRate**

**11.2.3.2.0.13.2 uint32\_t dspi\_master\_ctar\_config\_t::bitsPerFrame**

**11.2.3.2.0.13.3 dspi\_clock\_polarity\_t dspi\_master\_ctar\_config\_t::cpol**

**11.2.3.2.0.13.4 dspi\_clock\_phase\_t dspi\_master\_ctar\_config\_t::cpha**

**11.2.3.2.0.13.5 dspi\_shift\_direction\_t dspi\_master\_ctar\_config\_t::direction**

**11.2.3.2.0.13.6 uint32\_t dspi\_master\_ctar\_config\_t::pcsToSckDelayInNanoSec**

It also sets the boundary value if out of range.

**11.2.3.2.0.13.7 uint32\_t dspi\_master\_ctar\_config\_t::lastSckToPcsDelayInNanoSec**

It also sets the boundary value if out of range.

**11.2.3.2.0.13.8 uint32\_t dspi\_master\_ctar\_config\_t::betweenTransferDelayInNanoSec**

It also sets the boundary value if out of range.

### 11.2.3.3 struct dspi\_master\_config\_t

#### Data Fields

- [dspi\\_ctar\\_selection\\_t whichCtar](#)  
*The desired CTAR to use.*
- [dspi\\_master\\_ctar\\_config\\_t ctarConfig](#)  
*Set the ctarConfig to the desired CTAR.*
- [dspi\\_which\\_pcs\\_t whichPcs](#)  
*The desired Peripheral Chip Select (pcs).*
- [dspi\\_pcs\\_polarity\\_config\\_t pcsActiveHighOrLow](#)  
*The desired PCS active high or low.*
- [bool enableContinuousSCK](#)  
*CONT\_SCKE, continuous SCK enable.*
- [bool enableRxFifoOverWrite](#)  
*ROOE, receive FIFO overflow overwrite enable.*
- [bool enableModifiedTimingFormat](#)  
*Enables a modified transfer format to be used if true.*
- [dspi\\_master\\_sample\\_point\\_t samplePoint](#)  
*Controls when the module master samples SIN in the Modified Transfer Format.*

## DSPI Driver

### 11.2.3.3.0.14 Field Documentation

11.2.3.3.0.14.1 **dspi\_ctar\_selection\_t** **dspi\_master\_config\_t::whichCtar**

11.2.3.3.0.14.2 **dspi\_master\_ctar\_config\_t** **dspi\_master\_config\_t::ctarConfig**

11.2.3.3.0.14.3 **dspi\_which\_pcs\_t** **dspi\_master\_config\_t::whichPcs**

11.2.3.3.0.14.4 **dspi\_pcs\_polarity\_config\_t** **dspi\_master\_config\_t::pcsActiveHighOrLow**

11.2.3.3.0.14.5 **bool** **dspi\_master\_config\_t::enableContinuousSCK**

Note that the continuous SCK is only supported for CPHA = 1.

11.2.3.3.0.14.6 **bool** **dspi\_master\_config\_t::enableRx\_fifoOverWrite**

If ROOE = 0, the incoming data is ignored and the data from the transfer that generated the overflow is also ignored. If ROOE = 1, the incoming data is shifted to the shift register.

11.2.3.3.0.14.7 **bool** **dspi\_master\_config\_t::enableModifiedTimingFormat**

11.2.3.3.0.14.8 **dspi\_master\_sample\_point\_t** **dspi\_master\_config\_t::samplePoint**

It's valid only when CPHA=0.

### 11.2.3.4 struct **dspi\_slave\_ctar\_config\_t**

#### Data Fields

- **uint32\_t bitsPerFrame**  
*Bits per frame, minimum 4, maximum 16.*
- **dspi\_clock\_polarity\_t cpol**  
*Clock polarity.*
- **dspi\_clock\_phase\_t cpha**  
*Clock phase.*

### 11.2.3.4.0.15 Field Documentation

11.2.3.4.0.15.1 **uint32\_t** **dspi\_slave\_ctar\_config\_t::bitsPerFrame**

11.2.3.4.0.15.2 **dspi\_clock\_polarity\_t** **dspi\_slave\_ctar\_config\_t::cpol**

11.2.3.4.0.15.3 **dspi\_clock\_phase\_t** **dspi\_slave\_ctar\_config\_t::cpha**

Slave only supports MSB and does not support LSB.

### 11.2.3.5 struct `dspi_slave_config_t`

#### Data Fields

- `dspi_ctar_selection_t whichCtar`  
*The desired CTAR to use.*
- `dspi_slave_ctar_config_t ctarConfig`  
*Set the ctarConfig to the desired CTAR.*
- `bool enableContinuousSCK`  
*CONT\_SCKE, continuous SCK enable.*
- `bool enableRxFifoOverWrite`  
*ROOE, receive FIFO overflow overwrite enable.*
- `bool enableModifiedTimingFormat`  
*Enables a modified transfer format to be used if true.*
- `dspi_master_sample_point_t samplePoint`  
*Controls when the module master samples SIN in the Modified Transfer Format.*

#### 11.2.3.5.0.16 Field Documentation

##### 11.2.3.5.0.16.1 `dspi_ctar_selection_t dspi_slave_config_t::whichCtar`

##### 11.2.3.5.0.16.2 `dspi_slave_ctar_config_t dspi_slave_config_t::ctarConfig`

##### 11.2.3.5.0.16.3 `bool dspi_slave_config_t::enableContinuousSCK`

Note that the continuous SCK is only supported for CPHA = 1.

##### 11.2.3.5.0.16.4 `bool dspi_slave_config_t::enableRxFifoOverWrite`

If ROOE = 0, the incoming data is ignored and the data from the transfer that generated the overflow is also ignored. If ROOE = 1, the incoming data is shifted to the shift register.

##### 11.2.3.5.0.16.5 `bool dspi_slave_config_t::enableModifiedTimingFormat`

##### 11.2.3.5.0.16.6 `dspi_master_sample_point_t dspi_slave_config_t::samplePoint`

It's valid only when CPHA=0.

### 11.2.3.6 struct `dspi_transfer_t`

#### Data Fields

- `uint8_t * txData`  
*Send buffer.*
- `uint8_t * rxData`  
*Receive buffer.*
- `volatile size_t dataSize`  
*Transfer bytes.*
- `uint32_t configFlags`  
*Transfer transfer configuration flags; set from \_dspi\_transfer\_config\_flag\_for\_master if the transfer is*

## DSPI Driver

used for master or `_dspi_transfer_config_flag_for_slave` enumeration if the transfer is used for slave.

### 11.2.3.6.0.17 Field Documentation

11.2.3.6.0.17.1 `uint8_t* dspi_transfer_t::txData`

11.2.3.6.0.17.2 `uint8_t* dspi_transfer_t::rxData`

11.2.3.6.0.17.3 `volatile size_t dspi_transfer_t::dataSize`

11.2.3.6.0.17.4 `uint32_t dspi_transfer_t::configFlags`

### 11.2.3.7 `struct _dspi_master_handle`

Forward declaration of the `_dspi_master_handle` typedefs.

## Data Fields

- `uint32_t bitsPerFrame`  
*The desired number of bits per frame.*
- `volatile uint32_t command`  
*The desired data command.*
- `volatile uint32_t lastCommand`  
*The desired last data command.*
- `uint8_t fifoSize`  
*FIFO dataSize.*
- `volatile bool isPcsActiveAfterTransfer`  
*Indicates whether the PCS signal is active after the last frame transfer.*
- `volatile bool isThereExtraByte`  
*Indicates whether there are extra bytes.*
- `uint8_t *volatile txData`  
*Send buffer.*
- `uint8_t *volatile rxData`  
*Receive buffer.*
- `volatile size_t remainingSendByteCount`  
*A number of bytes remaining to send.*
- `volatile size_t remainingReceiveByteCount`  
*A number of bytes remaining to receive.*
- `size_t totalByteCount`  
*A number of transfer bytes.*
- `volatile uint8_t state`  
*DSPI transfer state, see `_dspi_transfer_state`.*
- `dspi_master_transfer_callback_t callback`  
*Completion callback.*
- `void *userData`  
*Callback user data.*

### 11.2.3.7.0.18 Field Documentation

- 11.2.3.7.0.18.1 `uint32_t dspi_master_handle_t::bitsPerFrame`
- 11.2.3.7.0.18.2 `volatile uint32_t dspi_master_handle_t::command`
- 11.2.3.7.0.18.3 `volatile uint32_t dspi_master_handle_t::lastCommand`
- 11.2.3.7.0.18.4 `uint8_t dspi_master_handle_t::fifoSize`
- 11.2.3.7.0.18.5 `volatile bool dspi_master_handle_t::isPcsActiveAfterTransfer`
- 11.2.3.7.0.18.6 `volatile bool dspi_master_handle_t::isThereExtraByte`
- 11.2.3.7.0.18.7 `uint8_t* volatile dspi_master_handle_t::txData`
- 11.2.3.7.0.18.8 `uint8_t* volatile dspi_master_handle_t::rxData`
- 11.2.3.7.0.18.9 `volatile size_t dspi_master_handle_t::remainingSendByteCount`
- 11.2.3.7.0.18.10 `volatile size_t dspi_master_handle_t::remainingReceiveByteCount`
- 11.2.3.7.0.18.11 `volatile uint8_t dspi_master_handle_t::state`
- 11.2.3.7.0.18.12 `dspi_master_transfer_callback_t dspi_master_handle_t::callback`
- 11.2.3.7.0.18.13 `void* dspi_master_handle_t::userData`

### 11.2.3.8 struct \_dspi\_slave\_handle

Forward declaration of the `_dspi_slave_handle` typedefs.

### Data Fields

- `uint32_t bitsPerFrame`  
*The desired number of bits per frame.*
- `volatile bool isThereExtraByte`  
*Indicates whether there are extra bytes.*
- `uint8_t *volatile txData`  
*Send buffer.*
- `uint8_t *volatile rxData`  
*Receive buffer.*
- `volatile size_t remainingSendByteCount`  
*A number of bytes remaining to send.*
- `volatile size_t remainingReceiveByteCount`  
*A number of bytes remaining to receive.*
- `size_t totalByteCount`  
*A number of transfer bytes.*
- `volatile uint8_t state`  
*DSPI transfer state.*

## DSPI Driver

- volatile uint32\_t `errorCount`  
*Error count for slave transfer.*
- `dspi_slave_transfer_callback_t` `callback`  
*Completion callback.*
- void \* `userData`  
*Callback user data.*

### 11.2.3.8.0.19 Field Documentation

11.2.3.8.0.19.1 `uint32_t dspi_slave_handle_t::bitsPerFrame`

11.2.3.8.0.19.2 `volatile bool dspi_slave_handle_t::isThereExtraByte`

11.2.3.8.0.19.3 `uint8_t* volatile dspi_slave_handle_t::txData`

11.2.3.8.0.19.4 `uint8_t* volatile dspi_slave_handle_t::rxData`

11.2.3.8.0.19.5 `volatile size_t dspi_slave_handle_t::remainingSendByteCount`

11.2.3.8.0.19.6 `volatile size_t dspi_slave_handle_t::remainingReceiveByteCount`

11.2.3.8.0.19.7 `volatile uint8_t dspi_slave_handle_t::state`

11.2.3.8.0.19.8 `volatile uint32_t dspi_slave_handle_t::errorCount`

11.2.3.8.0.19.9 `dspi_slave_transfer_callback_t dspi_slave_handle_t::callback`

11.2.3.8.0.19.10 `void* dspi_slave_handle_t::userData`

### 11.2.4 Macro Definition Documentation

11.2.4.1 `#define FSL_DSPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))`

11.2.4.2 `#define DSPI_DUMMY_DATA (0x00U)`

Dummy data used for Tx if there is no txData.

- 11.2.4.3 #define DSPI\_MASTER\_CTAR\_SHIFT (0U)
- 11.2.4.4 #define DSPI\_MASTER\_CTAR\_MASK (0x0FU)
- 11.2.4.5 #define DSPI\_MASTER\_PCS\_SHIFT (4U)
- 11.2.4.6 #define DSPI\_MASTER\_PCS\_MASK (0xF0U)
- 11.2.4.7 #define DSPI\_SLAVE\_CTAR\_SHIFT (0U)
- 11.2.4.8 #define DSPI\_SLAVE\_CTAR\_MASK (0x07U)

## 11.2.5 Typedef Documentation

- 11.2.5.1 `typedef void(* dspi_master_transfer_callback_t)(SPI_Type *base, dspi_master_handle_t *handle, status_t status, void *userData)`

## DSPI Driver

Parameters

|                 |                                                                  |
|-----------------|------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral address.                                         |
| <i>handle</i>   | Pointer to the handle for the DSPI master.                       |
| <i>status</i>   | Success or error code describing whether the transfer completed. |
| <i>userData</i> | Arbitrary pointer-dataSized value passed from the application.   |

### 11.2.5.2 `typedef void(* dspi_slave_transfer_callback_t)(SPI_Type *base, dspi_slave_handle_t *handle, status_t status, void *userData)`

Parameters

|                 |                                                                  |
|-----------------|------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral address.                                         |
| <i>handle</i>   | Pointer to the handle for the DSPI slave.                        |
| <i>status</i>   | Success or error code describing whether the transfer completed. |
| <i>userData</i> | Arbitrary pointer-dataSized value passed from the application.   |

## 11.2.6 Enumeration Type Documentation

### 11.2.6.1 `enum _dspi_status`

Enumerator

*kStatus\_DSPI\_Busy* DSPI transfer is busy.  
*kStatus\_DSPI\_Error* DSPI driver error.  
*kStatus\_DSPI\_Idle* DSPI is idle.  
*kStatus\_DSPI\_OutOfRange* DSPI transfer out of range.

### 11.2.6.2 `enum _dspi_flags`

Enumerator

*kDSPI\_TxCompleteFlag* Transfer Complete Flag.  
*kDSPI\_EndOfQueueFlag* End of Queue Flag.  
*kDSPI\_TxFifoUnderflowFlag* Transmit FIFO Underflow Flag.  
*kDSPI\_TxFifoFillRequestFlag* Transmit FIFO Fill Flag.  
*kDSPI\_RxFifoOverflowFlag* Receive FIFO Overflow Flag.  
*kDSPI\_RxFifoDrainRequestFlag* Receive FIFO Drain Flag.  
*kDSPI\_TxAndRxStatusFlag* The module is in Stopped/Running state.  
*kDSPI\_AllStatusFlag* All statuses above.

### 11.2.6.3 enum \_dspi\_interrupt\_enable

Enumerator

- kDSPI\_TxCompleteInterruptEnable* TCF interrupt enable.
- kDSPI\_EndOfQueueInterruptEnable* EOQF interrupt enable.
- kDSPI\_TxFifoUnderflowInterruptEnable* TFUF interrupt enable.
- kDSPI\_TxFifoFillRequestInterruptEnable* TFFF interrupt enable, DMA disable.
- kDSPI\_RxFifoOverflowInterruptEnable* RFOF interrupt enable.
- kDSPI\_RxFifoDrainRequestInterruptEnable* RFDF interrupt enable, DMA disable.
- kDSPI\_AllInterruptEnable* All above interrupts enable.

### 11.2.6.4 enum \_dspi\_dma\_enable

Enumerator

- kDSPI\_TxDmaEnable* TFFF flag generates DMA requests. No Tx interrupt request.
- kDSPI\_RxDmaEnable* RFDF flag generates DMA requests. No Rx interrupt request.

### 11.2.6.5 enum dspi\_master\_slave\_mode\_t

Enumerator

- kDSPI\_Master* DSPI peripheral operates in master mode.
- kDSPI\_Slave* DSPI peripheral operates in slave mode.

### 11.2.6.6 enum dspi\_master\_sample\_point\_t

This field is valid only when the CPHA bit in the CTAR register is 0.

Enumerator

- kDSPI\_SckToSin0Clock* 0 system clocks between SCK edge and SIN sample.
- kDSPI\_SckToSin1Clock* 1 system clock between SCK edge and SIN sample.
- kDSPI\_SckToSin2Clock* 2 system clocks between SCK edge and SIN sample.

### 11.2.6.7 enum dspi\_which\_pcs\_t

Enumerator

- kDSPI\_Pcs0* Pcs[0].
- kDSPI\_Pcs1* Pcs[1].
- kDSPI\_Pcs2* Pcs[2].

## DSPI Driver

*kDSPI\_Pcs3* Pcs[3].  
*kDSPI\_Pcs4* Pcs[4].  
*kDSPI\_Pcs5* Pcs[5].

### 11.2.6.8 enum dspi\_pcs\_polarity\_config\_t

Enumerator

*kDSPI\_PcsActiveHigh* Pcs Active High (idles low).  
*kDSPI\_PcsActiveLow* Pcs Active Low (idles high).

### 11.2.6.9 enum \_dspi\_pcs\_polarity

Enumerator

*kDSPI\_Pcs0ActiveLow* Pcs0 Active Low (idles high).  
*kDSPI\_Pcs1ActiveLow* Pcs1 Active Low (idles high).  
*kDSPI\_Pcs2ActiveLow* Pcs2 Active Low (idles high).  
*kDSPI\_Pcs3ActiveLow* Pcs3 Active Low (idles high).  
*kDSPI\_Pcs4ActiveLow* Pcs4 Active Low (idles high).  
*kDSPI\_Pcs5ActiveLow* Pcs5 Active Low (idles high).  
*kDSPI\_PcsAllActiveLow* Pcs0 to Pcs5 Active Low (idles high).

### 11.2.6.10 enum dspi\_clock\_polarity\_t

Enumerator

*kDSPI\_ClockPolarityActiveHigh* CPOL=0. Active-high DSPI clock (idles low).  
*kDSPI\_ClockPolarityActiveLow* CPOL=1. Active-low DSPI clock (idles high).

### 11.2.6.11 enum dspi\_clock\_phase\_t

Enumerator

*kDSPI\_ClockPhaseFirstEdge* CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.  
*kDSPI\_ClockPhaseSecondEdge* CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

### 11.2.6.12 enum dspi\_shift\_direction\_t

Enumerator

***kDSPI\_MsbFirst*** Data transfers start with most significant bit.

***kDSPI\_LsbFirst*** Data transfers start with least significant bit. Shifting out of LSB is not supported for slave

### 11.2.6.13 enum dspi\_delay\_type\_t

Enumerator

***kDSPI\_PcsToSck*** Pcs-to-SCK delay.

***kDSPI\_LastSckToPcs*** The last SCK edge to Pcs delay.

***kDSPI\_BetweenTransfer*** Delay between transfers.

### 11.2.6.14 enum dspi\_ctar\_selection\_t

Enumerator

***kDSPI\_Ctar0*** CTAR0 selection option for master or slave mode; note that CTAR0 and CTAR0\_S-LAVE are the same register address.

***kDSPI\_Ctar1*** CTAR1 selection option for master mode only.

***kDSPI\_Ctar2*** CTAR2 selection option for master mode only; note that some devices do not support CTAR2.

***kDSPI\_Ctar3*** CTAR3 selection option for master mode only; note that some devices do not support CTAR3.

***kDSPI\_Ctar4*** CTAR4 selection option for master mode only; note that some devices do not support CTAR4.

***kDSPI\_Ctar5*** CTAR5 selection option for master mode only; note that some devices do not support CTAR5.

***kDSPI\_Ctar6*** CTAR6 selection option for master mode only; note that some devices do not support CTAR6.

***kDSPI\_Ctar7*** CTAR7 selection option for master mode only; note that some devices do not support CTAR7.

### 11.2.6.15 enum \_dspi\_transfer\_config\_flag\_for\_master

Enumerator

***kDSPI\_MasterCtar0*** DSPI master transfer use CTAR0 setting.

***kDSPI\_MasterCtar1*** DSPI master transfer use CTAR1 setting.

***kDSPI\_MasterCtar2*** DSPI master transfer use CTAR2 setting.

## DSPI Driver

- kDSPI\_MasterCtar3* DSPI master transfer use CTAR3 setting.
- kDSPI\_MasterCtar4* DSPI master transfer use CTAR4 setting.
- kDSPI\_MasterCtar5* DSPI master transfer use CTAR5 setting.
- kDSPI\_MasterCtar6* DSPI master transfer use CTAR6 setting.
- kDSPI\_MasterCtar7* DSPI master transfer use CTAR7 setting.
- kDSPI\_MasterPcs0* DSPI master transfer use PCS0 signal.
- kDSPI\_MasterPcs1* DSPI master transfer use PCS1 signal.
- kDSPI\_MasterPcs2* DSPI master transfer use PCS2 signal.
- kDSPI\_MasterPcs3* DSPI master transfer use PCS3 signal.
- kDSPI\_MasterPcs4* DSPI master transfer use PCS4 signal.
- kDSPI\_MasterPcs5* DSPI master transfer use PCS5 signal.
- kDSPI\_MasterPcsContinuous* Indicates whether the PCS signal is continuous.
- kDSPI\_MasterActiveAfterTransfer* Indicates whether the PCS signal is active after the last frame transfer.

### 11.2.6.16 enum \_dsPIC\_transfer\_config\_flag\_for\_slave

Enumerator

- kDSPI\_SlaveCtar0* DSPI slave transfer use CTAR0 setting. DSPI slave can only use PCS0.

### 11.2.6.17 enum \_dsPIC\_transfer\_state

Enumerator

- kDSPI\_Idle* Nothing in the transmitter/receiver.
- kDSPI\_Busy* Transfer queue is not finished.
- kDSPI\_Error* Transfer error.

## 11.2.7 Function Documentation

### 11.2.7.1 void DSPI\_MasterInit ( SPI\_Type \* *base*, const dspi\_master\_config\_t \* *masterConfig*, uint32\_t *srcClock\_Hz* )

This function initializes the DSPI master configuration. This is an example use case.

```
*     dspi_master_config_t masterConfig;
*     masterConfig.whichCtar          = kDSPI_Ctar0;
*     masterConfig.ctarConfig.baudRate = 500000000U;
*     masterConfig.ctarConfig.bitsPerFrame = 8;
*     masterConfig.ctarConfig.cpol    =
*         kDSPI_ClockPolarityActiveHigh;
*     masterConfig.ctarConfig.cpha    =
*         kDSPI_ClockPhaseFirstEdge;
*     masterConfig.ctarConfig.direction =
```

```

    kDSPI_MsbFirst;
* masterConfig.ctarConfig.pcsToSckDelayInNanoSec      = 1000000000U /
    masterConfig.ctarConfig.baudRate ;
* masterConfig.ctarConfig.lastSckToPcsDelayInNanoSec   = 1000000000U
    / masterConfig.ctarConfig.baudRate ;
* masterConfig.ctarConfig.betweenTransferDelayInNanoSec =
    1000000000U / masterConfig.ctarConfig.baudRate ;
* masterConfig.whichPcs                                = kDSPI_Pcs0;
* masterConfig.pcsActiveHighOrLow                      =
    kDSPI_PcsActiveLow;
* masterConfig.enableContinuousSCK                     = false;
* masterConfig.enableRxFifoOverWrite                  = false;
* masterConfig.enableModifiedTimingFormat            = false;
* masterConfig.samplePoint                           =
    kDSPI_SckToSin0Clock;
* DSPI_MasterInit(base, &masterConfig, srcClock_Hz);
*

```

#### Parameters

|                     |                                                                                  |
|---------------------|----------------------------------------------------------------------------------|
| <i>base</i>         | DSPI peripheral address.                                                         |
| <i>masterConfig</i> | Pointer to the structure <a href="#">dsPIC33F dsPIC33EP DSPI API Reference</a> . |
| <i>srcClock_Hz</i>  | Module source input clock in Hertz.                                              |

#### 11.2.7.2 void DSPI\_MasterGetDefaultConfig ( [dsPIC33F dsPIC33EP DSPI API Reference](#) \* *masterConfig* )

The purpose of this API is to get the configuration structure initialized for the [DSPI\\_MasterInit\(\)](#). Users may use the initialized structure unchanged in the [DSPI\\_MasterInit\(\)](#) or modify the structure before calling the [DSPI\\_MasterInit\(\)](#). Example:

```

* dsPIC33F dsPIC33EP DSPI API Reference
* DSPI_MasterGetDefaultConfig(&masterConfig);
*

```

#### Parameters

|                     |                                                                            |
|---------------------|----------------------------------------------------------------------------|
| <i>masterConfig</i> | pointer to <a href="#">dsPIC33F dsPIC33EP DSPI API Reference</a> structure |
|---------------------|----------------------------------------------------------------------------|

#### 11.2.7.3 void DSPI\_SlaveInit ( [SPI\\_Type](#) \* *base*, const [dsPIC33F dsPIC33EP DSPI API Reference](#) \* *slaveConfig* )

This function initializes the DSPI slave configuration. This is an example use case.

```

* dsPIC33F dsPIC33EP DSPI API Reference
* slaveConfig->whichCtar          = kDSPI_Ctar0;
* slaveConfig->ctarConfig.bitsPerFrame = 8;
* slaveConfig->ctarConfig.cpol     =
    kDSPI_ClockPolarityActiveHigh;
* slaveConfig->ctarConfig.cpha     =
    kDSPI_ClockPhaseFirstEdge;
* slaveConfig->enableContinuousSCK = false;

```

## DSPI Driver

```
* slaveConfig->enableRxFifoOverWrite      = false;
* slaveConfig->enableModifiedTimingFormat = false;
* slaveConfig->samplePoint                = kDSPI_SckToSinc0Clock;
* DSPI_SlaveInit(base, &slaveConfig);
*
```

### Parameters

|                    |                                                                 |
|--------------------|-----------------------------------------------------------------|
| <i>base</i>        | DSPI peripheral address.                                        |
| <i>slaveConfig</i> | Pointer to the structure <a href="#">dspi_master_config_t</a> . |

### 11.2.7.4 void DSPI\_SlaveGetDefaultConfig ( [dspi\\_slave\\_config\\_t](#) \* *slaveConfig* )

The purpose of this API is to get the configuration structure initialized for the [DSPI\\_SlaveInit\(\)](#). Users may use the initialized structure unchanged in the [DSPI\\_SlaveInit\(\)](#) or modify the structure before calling the [DSPI\\_SlaveInit\(\)](#). This is an example.

```
* dspi_slave_config_t slaveConfig;
* DSPI_SlaveGetDefaultConfig(&slaveConfig);
*
```

### Parameters

|                    |                                                               |
|--------------------|---------------------------------------------------------------|
| <i>slaveConfig</i> | Pointer to the <a href="#">dspi_slave_config_t</a> structure. |
|--------------------|---------------------------------------------------------------|

### 11.2.7.5 void DSPI\_Deinit ( [SPI\\_Type](#) \* *base* )

Call this API to disable the DSPI clock.

### Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

### 11.2.7.6 static void DSPI\_Enable ( [SPI\\_Type](#) \* *base*, [bool](#) *enable* ) [inline], [static]

### Parameters

|               |                                                      |
|---------------|------------------------------------------------------|
| <i>base</i>   | DSPI peripheral address.                             |
| <i>enable</i> | Pass true to enable module, false to disable module. |

### 11.2.7.7 static uint32\_t DSPI\_GetStatusFlags ( SPI\_Type \* *base* ) [inline], [static]

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

Returns

DSPI status (in SR register).

### 11.2.7.8 static void DSPI\_ClearStatusFlags ( SPI\_Type \* *base*, uint32\_t *statusFlags* ) [inline], [static]

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status bit to clear. The list of status bits is defined in the dspi\_status\_and\_interrupt\_request\_t. The function uses these bit positions in its algorithm to clear the desired flag state. This is an example.

```
* DSPI_ClearStatusFlags(base, kDSPI_TxCompleteFlag |
                      kDSPI_EndOfQueueFlag);
*
```

Parameters

|                    |                                                |
|--------------------|------------------------------------------------|
| <i>base</i>        | DSPI peripheral address.                       |
| <i>statusFlags</i> | The status flag used from the type dspi_flags. |

< The status flags are cleared by writing 1 (w1c).

### 11.2.7.9 void DSPI\_EnableInterrupts ( SPI\_Type \* *base*, uint32\_t *mask* )

This function configures the various interrupt masks of the DSPI. The parameters are a base and an interrupt mask. Note, for Tx Fill and Rx FIFO drain requests, enable the interrupt request and disable the DMA request.

```
* DSPI_EnableInterrupts(base,
                      kDSPI_TxCompleteInterruptEnable |
                      kDSPI_EndOfQueueInterruptEnable );
*
```

## DSPI Driver

Parameters

|             |                                                          |
|-------------|----------------------------------------------------------|
| <i>base</i> | DSPI peripheral address.                                 |
| <i>mask</i> | The interrupt mask; use the enum _dspi_interrupt_enable. |

### 11.2.7.10 static void DSPI\_DisableInterrupts ( SPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

```
* DSPI_DisableInterrupts(base,  
    kDSPI_TxCompleteInterruptEnable |  
    kDSPI_EndOfQueueInterruptEnable );  
*
```

Parameters

|             |                                                          |
|-------------|----------------------------------------------------------|
| <i>base</i> | DSPI peripheral address.                                 |
| <i>mask</i> | The interrupt mask; use the enum _dspi_interrupt_enable. |

### 11.2.7.11 static void DSPI\_EnableDMA ( SPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function configures the Rx and Tx DMA mask of the DSPI. The parameters are a base and a DMA mask.

```
* DSPI_EnableDMA(base, kDSPI_TxDmaEnable |  
    kDSPI_RxDmaEnable);  
*
```

Parameters

|             |                                                   |
|-------------|---------------------------------------------------|
| <i>base</i> | DSPI peripheral address.                          |
| <i>mask</i> | The interrupt mask; use the enum dspi_dma_enable. |

### 11.2.7.12 static void DSPI\_DisableDMA ( SPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function configures the Rx and Tx DMA mask of the DSPI. The parameters are a base and a DMA mask.

```
* SPI_DisableDMA(base, kDSPI_TxDmaEnable | kDSPI_RxDmaEnable);  
*
```

Parameters

|             |                                                                 |
|-------------|-----------------------------------------------------------------|
| <i>base</i> | DSPI peripheral address.                                        |
| <i>mask</i> | The interrupt mask; use the enum <code>dspi_dma_enable</code> . |

#### **11.2.7.13 static uint32\_t DSPI\_MasterGetTxRegisterAddress ( SPI\_Type \* *base* ) [inline], [static]**

This function gets the DSPI master PUSHR data register address because this value is needed for the DMA operation.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

Returns

The DSPI master PUSHR data register address.

#### **11.2.7.14 static uint32\_t DSPI\_SlaveGetTxRegisterAddress ( SPI\_Type \* *base* ) [inline], [static]**

This function gets the DSPI slave PUSHR data register address as this value is needed for the DMA operation.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

Returns

The DSPI slave PUSHR data register address.

#### **11.2.7.15 static uint32\_t DSPI\_GetRxRegisterAddress ( SPI\_Type \* *base* ) [inline], [static]**

This function gets the DSPI POPR data register address as this value is needed for the DMA operation.

## DSPI Driver

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

Returns

The DSPI POPR data register address.

### 11.2.7.16 static void DSPI\_SetMasterSlaveMode ( SPI\_Type \* *base*, dspi\_master\_slave\_mode\_t *mode* ) [inline], [static]

Parameters

|             |                                                                  |
|-------------|------------------------------------------------------------------|
| <i>base</i> | DSPI peripheral address.                                         |
| <i>mode</i> | Mode setting (master or slave) of type dspi_master_slave_mode_t. |

### 11.2.7.17 static bool DSPI\_IsMaster ( SPI\_Type \* *base* ) [inline], [static]

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

Returns

Returns true if the module is in master mode or false if the module is in slave mode.

### 11.2.7.18 static void DSPI\_StartTransfer ( SPI\_Type \* *base* ) [inline], [static]

This function sets the module to start data transfer in either master or slave mode.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

### 11.2.7.19 static void DSPI\_StopTransfer ( SPI\_Type \* *base* ) [inline], [static]

This function stops data transfers in either master or slave modes.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

#### 11.2.7.20 static void DSPI\_SetFifoEnable ( SPI\_Type \* *base*, bool *enableTxFifo*, bool *enableRxFifo* ) [inline], [static]

This function allows the caller to disable/enable the Tx and Rx FIFOs independently. Note that to disable, pass in a logic 0 (false) for the particular FIFO configuration. To enable, pass in a logic 1 (true).

Parameters

|                     |                                                                     |
|---------------------|---------------------------------------------------------------------|
| <i>base</i>         | DSPI peripheral address.                                            |
| <i>enableTxFifo</i> | Disables (false) the TX FIFO; Otherwise, enables (true) the TX FIFO |
| <i>enableRxFifo</i> | Disables (false) the RX FIFO; Otherwise, enables (true) the RX FIFO |

#### 11.2.7.21 static void DSPI\_FlushFifo ( SPI\_Type \* *base*, bool *flushTxFifo*, bool *flushRxFifo* ) [inline], [static]

Parameters

|                    |                                                                           |
|--------------------|---------------------------------------------------------------------------|
| <i>base</i>        | DSPI peripheral address.                                                  |
| <i>flushTxFifo</i> | Flushes (true) the Tx FIFO; Otherwise, does not flush (false) the Tx FIFO |
| <i>flushRxFifo</i> | Flushes (true) the Rx FIFO; Otherwise, does not flush (false) the Rx FIFO |

#### 11.2.7.22 static void DSPI\_SetAllPcsPolarity ( SPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

For example, PCS0 and PCS1 are set to active low and other PCS is set to active high. Note that the number of PCSs is specific to the device.

```
* DSPI_SetAllPcsPolarity(base, kDSPI_Pcs0ActiveLow |
    kDSPI_Pcs1ActiveLow);
```

Parameters

## DSPI Driver

|             |                                                                       |
|-------------|-----------------------------------------------------------------------|
| <i>base</i> | DSPI peripheral address.                                              |
| <i>mask</i> | The PCS polarity mask; use the enum <code>_dspi_pcs_polarity</code> . |

### 11.2.7.23 `uint32_t DSPI_MasterSetBaudRate ( SPI_Type * base, dspi_ctar_selection_t whichCtar, uint32_t baudRate_Bps, uint32_t srcClock_Hz )`

This function takes in the desired baudRate\_Bps (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate, and returns the calculated baud rate in bits-per-second. It requires that the caller also provide the frequency of the module source clock (in Hertz).

Parameters

|                     |                                                                                                          |
|---------------------|----------------------------------------------------------------------------------------------------------|
| <i>base</i>         | DSPI peripheral address.                                                                                 |
| <i>whichCtar</i>    | The desired Clock and Transfer Attributes Register (CTAR) of the type <code>dspi_ctar_selection_t</code> |
| <i>baudRate_Bps</i> | The desired baud rate in bits per second                                                                 |
| <i>srcClock_Hz</i>  | Module source input clock in Hertz                                                                       |

Returns

The actual calculated baud rate

### 11.2.7.24 `void DSPI_MasterSetDelayScaler ( SPI_Type * base, dspi_ctar_selection_t whichCtar, uint32_t prescaler, uint32_t scaler, dspi_delay_type_t whichDelay )`

This function configures the PCS to SCK delay pre-scalar (PcsSCK) and scalar (CSSCK), after SCK delay pre-scalar (PASC) and scalar (ASC), and the delay after transfer pre-scalar (PDT) and scalar (DT).

These delay names are available in the type `dspi_delay_type_t`.

The user passes the delay to the configuration along with the prescaler and scalar value. This allows the user to directly set the prescaler/scalar values if pre-calculated or to manually increment either value.

Parameters

|                  |                                                                                                        |
|------------------|--------------------------------------------------------------------------------------------------------|
| <i>base</i>      | DSPI peripheral address.                                                                               |
| <i>whichCtar</i> | The desired Clock and Transfer Attributes Register (CTAR) of type <code>dspi_ctar_selection_t</code> . |

|                   |                                                                                |
|-------------------|--------------------------------------------------------------------------------|
| <i>prescaler</i>  | The prescaler delay value (can be an integer 0, 1, 2, or 3).                   |
| <i>scaler</i>     | The scaler delay value (can be any integer between 0 to 15).                   |
| <i>whichDelay</i> | The desired delay to configure; must be of type <code>dspi_delay_type_t</code> |

### 11.2.7.25 `uint32_t DSPI_MasterSetDelayTimes ( SPI_Type * base, dspi_ctar_selection_t whichCtar, dspi_delay_type_t whichDelay, uint32_t srcClock_Hz, uint32_t delayTimeInNanoSec )`

This function calculates the values for the following. PCS to SCK delay pre-scalar (PCSSCK) and scalar (CSSCK), or After SCK delay pre-scalar (PASC) and scalar (ASC), or Delay after transfer pre-scalar (PDT) and scalar (DT).

These delay names are available in the type `dspi_delay_type_t`.

The user passes which delay to configure along with the desired delay value in nanoseconds. The function calculates the values needed for the prescaler and scaler. Note that returning the calculated delay as an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. The higher-level peripheral driver alerts the user of an out of range delay input.

Parameters

|                           |                                                                                                        |
|---------------------------|--------------------------------------------------------------------------------------------------------|
| <i>base</i>               | DSPI peripheral address.                                                                               |
| <i>whichCtar</i>          | The desired Clock and Transfer Attributes Register (CTAR) of type <code>dspi_ctar_selection_t</code> . |
| <i>whichDelay</i>         | The desired delay to configure, must be of type <code>dspi_delay_type_t</code>                         |
| <i>srcClock_Hz</i>        | Module source input clock in Hertz                                                                     |
| <i>delayTimeInNanoSec</i> | The desired delay value in nanoseconds.                                                                |

Returns

The actual calculated delay value.

### 11.2.7.26 `static void DSPI_MasterWriteData ( SPI_Type * base, dspi_command_data_config_t * command, uint16_t data ) [inline], [static]`

In master mode, the 16-bit data is appended to the 16-bit command info. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the

## DSPI Driver

desired Clock and Transfer Attributes register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). This is an example.

```
* dspi_command_data_config_t commandConfig;
* commandConfig.isPcsContinuous = true;
* commandConfig.whichCtar = kDSPICtar0;
* commandConfig.whichPcs = kDSPIPcs0;
* commandConfig.clearTransferCount = false;
* commandConfig.isEndOfQueue = false;
* DSPI_MasterWriteData(base, &commandConfig, dataWord);
```

Parameters

|                |                                   |
|----------------|-----------------------------------|
| <i>base</i>    | DSPI peripheral address.          |
| <i>command</i> | Pointer to the command structure. |
| <i>data</i>    | The data word to be sent.         |

### 11.2.7.27 void DSPI\_GetDefaultDataCommandConfig ( **dspi\_command\_data\_config\_t** \* *command* )

The purpose of this API is to get the configuration structure initialized for use in the DSPI\_MasterWrite\_xx(). Users may use the initialized structure unchanged in the DSPI\_MasterWrite\_xx() or modify the structure before calling the DSPI\_MasterWrite\_xx(). This is an example.

```
* dspi_command_data_config_t command;
* DSPI_GetDefaultDataCommandConfig(&command);
*
```

Parameters

|                |                                                             |
|----------------|-------------------------------------------------------------|
| <i>command</i> | Pointer to the <b>dspi_command_data_config_t</b> structure. |
|----------------|-------------------------------------------------------------|

### 11.2.7.28 void DSPI\_MasterWriteDataBlocking ( **SPI\_Type** \* *base*, **dspi\_command\_data\_config\_t** \* *command*, **uint16\_t** *data* )

In master mode, the 16-bit data is appended to the 16-bit command info. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the desired Clock and Transfer Attributes register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). This is an example.

```
* dspi_command_config_t commandConfig;
* commandConfig.isPcsContinuous = true;
* commandConfig.whichCtar = kDSPICtar0;
```

```

* commandConfig.whichPcs = kDSPIPcs1;
* commandConfig.clearTransferCount = false;
* commandConfig.isEndOfQueue = false;
* DSPI_MasterWriteDataBlocking(base, &commandConfig, dataWord);
*

```

Note that this function does not return until after the transmit is complete. Also note that the DSPI must be enabled and running to transmit data (MCR[MDIS] & [HALT] = 0). Because the SPI is a synchronous protocol, the received data is available when the transmit completes.

Parameters

|                |                                   |
|----------------|-----------------------------------|
| <i>base</i>    | DSPI peripheral address.          |
| <i>command</i> | Pointer to the command structure. |
| <i>data</i>    | The data word to be sent.         |

#### 11.2.7.29 static uint32\_t DSPI\_MasterGetFormattedCommand ( *dspi\_command\_data\_config\_t* \* *command* ) [inline], [static]

This function allows the caller to pass in the data command structure and returns the command word formatted according to the DSPI PUSHR register bit field placement. The user can then "OR" the returned command word with the desired data to send and use the function DSPI\_HAL\_WriteCommandDataMastermode or DSPI\_HAL\_WriteCommandDataMastermodeBlocking to write the entire 32-bit command data word to the PUSHR. This helps improve performance in cases where the command structure is constant. For example, the user calls this function before starting a transfer to generate the command word. When they are ready to transmit the data, they OR this formatted command word with the desired data to transmit. This process increases transmit performance when compared to calling send functions, such as DSPI\_HAL\_WriteDataMastermode, which format the command word each time a data word is to be sent.

Parameters

|                |                                   |
|----------------|-----------------------------------|
| <i>command</i> | Pointer to the command structure. |
|----------------|-----------------------------------|

Returns

The command word formatted to the PUSHR data register bit field.

#### 11.2.7.30 void DSPI\_MasterWriteCommandDataBlocking ( *SPI\_Type* \* *base*, *uint32\_t* *data* )

In this function, the user must append the 16-bit data to the 16-bit command information and then provide the total 32-bit word as the data to send. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the desired Clock and Transfer Attributes

## DSPI Driver

register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). The user is responsible for appending this command with the data to send. This is an example:

```
* dataWord = <16-bit command> | <16-bit data>;
* DSPI_MasterWriteCommandDataBlocking(base, dataWord);
*
```

Note that this function does not return until after the transmit is complete. Also note that the DSPI must be enabled and running to transmit data (MCR[MDIS] & [HALT] = 0). Because the SPI is a synchronous protocol, the received data is available when the transmit completes.

For a blocking polling transfer, see methods below. Option 1: uint32\_t command\_to\_send = DSPI\_MasterGetFormattedCommand(&command); uint32\_t data0 = command\_to\_send | data\_need\_to\_send\_0; uint32\_t data1 = command\_to\_send | data\_need\_to\_send\_1; uint32\_t data2 = command\_to\_send | data\_need\_to\_send\_2;

DSPI\_MasterWriteCommandDataBlocking(base,data0); DSPI\_MasterWriteCommandDataBlocking(base,data1);  
DSPI\_MasterWriteCommandDataBlocking(base,data2);

Option 2: DSPI\_MasterWriteDataBlocking(base,&command,data\_need\_to\_send\_0); DSPI\_MasterWriteDataBlocking(base,&command,data\_need\_to\_send\_1); DSPI\_MasterWriteDataBlocking(base,&command,data\_need\_to\_send\_2);

Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | DSPI peripheral address.                              |
| <i>data</i> | The data word (command and data combined) to be sent. |

### 11.2.7.31 static void DSPI\_SlaveWriteData ( SPI\_Type \* *base*, uint32\_t *data* ) [inline], [static]

In slave mode, up to 16-bit words may be written.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
| <i>data</i> | The data to send.        |

### 11.2.7.32 void DSPI\_SlaveWriteDataBlocking ( SPI\_Type \* *base*, uint32\_t *data* )

In slave mode, up to 16-bit words may be written. The function first clears the transmit complete flag, writes data into data register, and finally waits until the data is transmitted.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
| <i>data</i> | The data to send.        |

#### 11.2.7.33 static uint32\_t DSPI\_ReadData ( SPI\_Type \* *base* ) [inline], [static]

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | DSPI peripheral address. |
|-------------|--------------------------|

Returns

The data from the read data buffer.

#### 11.2.7.34 void DSPI\_MasterTransferCreateHandle ( SPI\_Type \* *base*, dspi\_master\_handle\_t \* *handle*, dspi\_master\_transfer\_callback\_t *callback*, void \* *userData* )

This function initializes the DSPI handle, which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Parameters

|                 |                                              |
|-----------------|----------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                |
| <i>handle</i>   | DSPI handle pointer to dspi_master_handle_t. |
| <i>callback</i> | DSPI callback.                               |
| <i>userData</i> | Callback function parameter.                 |

#### 11.2.7.35 status\_t DSPI\_MasterTransferBlocking ( SPI\_Type \* *base*, dspi\_transfer\_t \* *transfer* )

This function transfers data using polling. This is a blocking function, which does not return until all transfers have been completed.

## DSPI Driver

Parameters

|                 |                                                           |
|-----------------|-----------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                             |
| <i>transfer</i> | Pointer to the <a href="#">dspi_transfer_t</a> structure. |

Returns

status of status\_t.

### 11.2.7.36 status\_t DSPI\_MasterTransferNonBlocking ( SPI\_Type \* *base*, dspi\_master\_handle\_t \* *handle*, dspi\_transfer\_t \* *transfer* )

This function transfers data using interrupts. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

|                 |                                                                                                |
|-----------------|------------------------------------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                                                  |
| <i>handle</i>   | Pointer to the <a href="#">dspi_master_handle_t</a> structure which stores the transfer state. |
| <i>transfer</i> | Pointer to the <a href="#">dspi_transfer_t</a> structure.                                      |

Returns

status of status\_t.

### 11.2.7.37 status\_t DSPI\_MasterTransferGetCount ( SPI\_Type \* *base*, dspi\_master\_handle\_t \* *handle*, size\_t \* *count* )

This function gets the master transfer count.

Parameters

|               |                                                                                                |
|---------------|------------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                                  |
| <i>handle</i> | Pointer to the <a href="#">dspi_master_handle_t</a> structure which stores the transfer state. |
| <i>count</i>  | The number of bytes transferred by using the non-blocking transaction.                         |

Returns

status of status\_t.

**11.2.7.38 void DSPI\_MasterTransferAbort ( SPI\_Type \* *base*, dspi\_master\_handle\_t \* *handle* )**

This function aborts a transfer using an interrupt.

## DSPI Driver

Parameters

|               |                                                                                             |
|---------------|---------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                               |
| <i>handle</i> | Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state. |

### 11.2.7.39 `void DSPI_MasterTransferHandleIRQ ( SPI_Type * base, dspi_master_handle_t * handle )`

This function processes the DSPI transmit and receive IRQ.

Parameters

|               |                                                                                             |
|---------------|---------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                               |
| <i>handle</i> | Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state. |

### 11.2.7.40 `void DSPI_SlaveTransferCreateHandle ( SPI_Type * base, dspi_slave_handle_t * handle, dspi_slave_transfer_callback_t callback, void * userData )`

This function initializes the DSPI handle, which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Parameters

|                 |                                                               |
|-----------------|---------------------------------------------------------------|
| <i>handle</i>   | DSPI handle pointer to the <code>dspi_slave_handle_t</code> . |
| <i>base</i>     | DSPI peripheral base address.                                 |
| <i>callback</i> | DSPI callback.                                                |
| <i>userData</i> | Callback function parameter.                                  |

### 11.2.7.41 `status_t DSPI_SlaveTransferNonBlocking ( SPI_Type * base, dspi_slave_handle_t * handle, dspi_transfer_t * transfer )`

This function transfers data using an interrupt. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

|                 |                                                                                            |
|-----------------|--------------------------------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                                              |
| <i>handle</i>   | Pointer to the <code>dspi_slave_handle_t</code> structure which stores the transfer state. |
| <i>transfer</i> | Pointer to the <code>dspi_transfer_t</code> structure.                                     |

Returns

status of `status_t`.

#### 11.2.7.42 `status_t DSPI_SlaveTransferGetCount ( SPI_Type * base, dspi_slave_handle_t * handle, size_t * count )`

This function gets the slave transfer count.

Parameters

|               |                                                                                             |
|---------------|---------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                               |
| <i>handle</i> | Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state. |
| <i>count</i>  | The number of bytes transferred by using the non-blocking transaction.                      |

Returns

status of `status_t`.

#### 11.2.7.43 `void DSPI_SlaveTransferAbort ( SPI_Type * base, dspi_slave_handle_t * handle )`

This function aborts a transfer using an interrupt.

Parameters

|               |                                                                                            |
|---------------|--------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                              |
| <i>handle</i> | Pointer to the <code>dspi_slave_handle_t</code> structure which stores the transfer state. |

#### 11.2.7.44 `void DSPI_SlaveTransferHandleIRQ ( SPI_Type * base, dspi_slave_handle_t * handle )`

This function processes the DSPI transmit and receive IRQ.

## DSPI Driver

Parameters

|               |                                                                                            |
|---------------|--------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                              |
| <i>handle</i> | Pointer to the <code>dspi_slave_handle_t</code> structure which stores the transfer state. |

## 11.3 DSPI DMA Driver

### 11.3.1 Overview

This section describes the programming interface of the DSPI Peripheral driver. The DSPI driver configures DSPI module and provides the functional and transactional interfaces to build the DSPI application.

## Data Structures

- struct [dspi\\_master\\_dma\\_handle\\_t](#)  
*DSPI master DMA transfer handle structure used for transactional API. [More...](#)*
- struct [dspi\\_slave\\_dma\\_handle\\_t](#)  
*DSPI slave DMA transfer handle structure used for transactional API. [More...](#)*

## Typedefs

- typedef void(\* [dspi\\_master\\_dma\\_transfer\\_callback\\_t](#) )(SPI\_Type \*base, dspi\_master\_dma\_handle\_t \*handle, status\_t status, void \*userData)  
*Completion callback function pointer type.*
- typedef void(\* [dspi\\_slave\\_dma\\_transfer\\_callback\\_t](#) )(SPI\_Type \*base, dspi\_slave\_dma\_handle\_t \*handle, status\_t status, void \*userData)  
*Completion callback function pointer type.*

## Functions

- void [DSPI\\_MasterTransferCreateHandleDMA](#) (SPI\_Type \*base, dspi\_master\_dma\_handle\_t \*handle, [dspi\\_master\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData, dma\_handle\_t \*dmaRxRegToRxDataHandle, dma\_handle\_t \*dmaTxDataToIntermediaryHandle, dma\_handle\_t \*dmaIntermediaryToTxRegHandle)  
*Initializes the DSPI master DMA handle.*
- status\_t [DSPI\\_MasterTransferDMA](#) (SPI\_Type \*base, dspi\_master\_dma\_handle\_t \*handle, [dspi\\_transfer\\_t](#) \*transfer)  
*DSPI master transfers data using DMA.*
- void [DSPI\\_MasterTransferAbortDMA](#) (SPI\_Type \*base, dspi\_master\_dma\_handle\_t \*handle)  
*DSPI master aborts a transfer which is using DMA.*
- status\_t [DSPI\\_MasterTransferGetCountDMA](#) (SPI\_Type \*base, dspi\_master\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets the master DMA transfer remaining bytes.*
- void [DSPI\\_SlaveTransferCreateHandleDMA](#) (SPI\_Type \*base, dspi\_slave\_dma\_handle\_t \*handle, [dspi\\_slave\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData, dma\_handle\_t \*dmaRxRegToRxDataHandle, dma\_handle\_t \*dmaTxDataToTxRegHandle)  
*Initializes the DSPI slave DMA handle.*
- status\_t [DSPI\\_SlaveTransferDMA](#) (SPI\_Type \*base, dspi\_slave\_dma\_handle\_t \*handle, [dspi\\_transfer\\_t](#) \*transfer)  
*DSPI slave transfers data using DMA.*

## DSPI DMA Driver

- void **DSPI\_SlaveTransferAbortDMA** (SPI\_Type \*base, dspi\_slave\_dma\_handle\_t \*handle)  
*DSPI slave aborts a transfer which is using DMA.*
- status\_t **DSPI\_SlaveTransferGetCountDMA** (SPI\_Type \*base, dspi\_slave\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets the slave DMA transfer remaining bytes.*

### 11.3.2 Data Structure Documentation

#### 11.3.2.1 struct \_dspi\_master\_dma\_handle

Forward declaration of the DSPI DMA master handle typedefs.

#### Data Fields

- uint32\_t **bitsPerFrame**  
*The desired number of bits per frame.*
- volatile uint32\_t **command**  
*The desired data command.*
- volatile uint32\_t **lastCommand**  
*The desired last data command.*
- uint8\_t **fifoSize**  
*FIFO dataSize.*
- volatile bool **isPcsActiveAfterTransfer**  
*Indicates whether the PCS signal keeps active after the last frame transfer.*
- volatile bool **isThereExtraByte**  
*Indicates whether there is an extra byte.*
- uint8\_t \*volatile **txData**  
*Send buffer.*
- uint8\_t \*volatile **rxData**  
*Receive buffer.*
- volatile size\_t **remainingSendByteCount**  
*A number of bytes remaining to send.*
- volatile size\_t **remainingReceiveByteCount**  
*A number of bytes remaining to receive.*
- size\_t **totalByteCount**  
*A number of transfer bytes.*
- uint32\_t **rxBuffIfNull**  
*Used if there is not rxData for DMA purpose.*
- uint32\_t **txBuffIfNull**  
*Used if there is not txData for DMA purpose.*
- volatile uint8\_t **state**  
*DSPI transfer state, see \_dspi\_transfer\_state.*
- **dspi\_master\_dma\_transfer\_callback\_t callback**  
*Completion callback.*
- void \* **userData**  
*Callback user data.*
- dma\_handle\_t \* **dmaRxRegToRxDataHandle**  
*dma\_handle\_t handle point used for RxReg to RxData buff*

- `dma_handle_t *dmaTxDataToIntermediaryHandle`  
`dma_handle_t handle point used for TxData to Intermediary`
- `dma_handle_t *dmaIntermediaryToTxRegHandle`  
`dma_handle_t handle point used for Intermediary to TxReg`

### 11.3.2.1.0.20 Field Documentation

11.3.2.1.0.20.1 `uint32_t dspi_master_dma_handle_t::bitsPerFrame`

11.3.2.1.0.20.2 `volatile uint32_t dspi_master_dma_handle_t::command`

11.3.2.1.0.20.3 `volatile uint32_t dspi_master_dma_handle_t::lastCommand`

11.3.2.1.0.20.4 `uint8_t dspi_master_dma_handle_t::fifoSize`

11.3.2.1.0.20.5 `volatile bool dspi_master_dma_handle_t::isPcsActiveAfterTransfer`

11.3.2.1.0.20.6 `volatile bool dspi_master_dma_handle_t::isThereExtraByte`

11.3.2.1.0.20.7 `uint8_t* volatile dspi_master_dma_handle_t::txData`

11.3.2.1.0.20.8 `uint8_t* volatile dspi_master_dma_handle_t::rxData`

11.3.2.1.0.20.9 `volatile size_t dspi_master_dma_handle_t::remainingSendByteCount`

11.3.2.1.0.20.10 `volatile size_t dspi_master_dma_handle_t::remainingReceiveByteCount`

11.3.2.1.0.20.11 `uint32_t dspi_master_dma_handle_t::rxBuffIfNull`

11.3.2.1.0.20.12 `uint32_t dspi_master_dma_handle_t::txBuffIfNull`

11.3.2.1.0.20.13 `volatile uint8_t dspi_master_dma_handle_t::state`

11.3.2.1.0.20.14 `dspi_master_dma_transfer_callback_t dspi_master_dma_handle_t::callback`

11.3.2.1.0.20.15 `void* dspi_master_dma_handle_t::userData`

### 11.3.2.2 `struct _dspi_slave_dma_handle`

Forward declaration of the DSPI DMA slave handle typedefs.

### Data Fields

- `uint32_t bitsPerFrame`  
`Desired number of bits per frame.`
- `volatile bool isThereExtraByte`  
`Indicates whether there is an extra byte.`
- `uint8_t *volatile txData`  
`A send buffer.`
- `uint8_t *volatile rxData`

## DSPI DMA Driver

- **A receive buffer.**
  - volatile size\_t **remainingSendByteCount**  
*A number of bytes remaining to send.*
  - volatile size\_t **remainingReceiveByteCount**  
*A number of bytes remaining to receive.*
- size\_t **totalByteCount**  
*A number of transfer bytes.*
- uint32\_t **rxBuffIfNull**  
*Used if there is not rxData for DMA purpose.*
- uint32\_t **txBuffIfNull**  
*Used if there is not txData for DMA purpose.*
- uint32\_t **txLastData**  
*Used if there is an extra byte when 16 bits per frame for DMA purpose.*
- volatile uint8\_t **state**  
*DSPI transfer state.*
- uint32\_t **errorCount**  
*Error count for the slave transfer.*
- **dspi\_slave\_dma\_transfer\_callback\_t callback**  
*Completion callback.*
- void \* **userData**  
*Callback user data.*
- dma\_handle\_t \* **dmaRxRegToRxDataHandle**  
*dma\_handle\_t handle point used for RxReg to RxData buff*
- dma\_handle\_t \* **dmaTxDataToTxRegHandle**  
*dma\_handle\_t handle point used for TxData to TxReg*

### 11.3.2.2.0.21 Field Documentation

- 11.3.2.2.0.21.1 `uint32_t dspi_slave_dma_handle_t::bitsPerFrame`
- 11.3.2.2.0.21.2 `volatile bool dspi_slave_dma_handle_t::isThereExtraByte`
- 11.3.2.2.0.21.3 `uint8_t* volatile dspi_slave_dma_handle_t::txData`
- 11.3.2.2.0.21.4 `uint8_t* volatile dspi_slave_dma_handle_t::rxData`
- 11.3.2.2.0.21.5 `volatile size_t dspi_slave_dma_handle_t::remainingSendByteCount`
- 11.3.2.2.0.21.6 `volatile size_t dspi_slave_dma_handle_t::remainingReceiveByteCount`
- 11.3.2.2.0.21.7 `uint32_t dspi_slave_dma_handle_t::rxBuffIfNull`
- 11.3.2.2.0.21.8 `uint32_t dspi_slave_dma_handle_t::txBuffIfNull`
- 11.3.2.2.0.21.9 `uint32_t dspi_slave_dma_handle_t::txLastData`
- 11.3.2.2.0.21.10 `volatile uint8_t dspi_slave_dma_handle_t::state`
- 11.3.2.2.0.21.11 `uint32_t dspi_slave_dma_handle_t::errorCount`
- 11.3.2.2.0.21.12 `dspi_slave_dma_transfer_callback_t dspi_slave_dma_handle_t::callback`
- 11.3.2.2.0.21.13 `void* dspi_slave_dma_handle_t::userData`

### 11.3.3 Typedef Documentation

- 11.3.3.1 `typedef void(* dspi_master_dma_transfer_callback_t)(SPI_Type *base, dspi_master_dma_handle_t *handle, status_t status, void *userData)`

## DSPI DMA Driver

Parameters

|                 |                                                                  |
|-----------------|------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                    |
| <i>handle</i>   | Pointer to the handle for the DSPI master.                       |
| <i>status</i>   | Success or error code describing whether the transfer completed. |
| <i>userData</i> | Arbitrary pointer-dataSized value passed from the application.   |

### 11.3.3.2 **typedef void(\* dspi\_slave\_dma\_transfer\_callback\_t)(SPI\_Type \*base, dspi\_slave\_dma\_handle\_t \*handle, status\_t status, void \*userData)**

Parameters

|                 |                                                                  |
|-----------------|------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                    |
| <i>handle</i>   | Pointer to the handle for the DSPI slave.                        |
| <i>status</i>   | Success or error code describing whether the transfer completed. |
| <i>userData</i> | Arbitrary pointer-dataSized value passed from the application.   |

## 11.3.4 Function Documentation

### 11.3.4.1 **void DSPI\_MasterTransferCreateHandleDMA ( SPI\_Type \* *base*, dspi\_master\_dma\_handle\_t \* *handle*, dspi\_master\_dma\_transfer\_callback\_t *callback*, void \* *userData*, dma\_handle\_t \* *dmaRxRegToRxDataHandle*, dma\_handle\_t \* *dmaTxDataToIntermediaryHandle*, dma\_handle\_t \* *dmaIntermediaryToTxRegHandle* )**

This function initializes the DSPI DMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI DMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx is the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for *dmaRxRegToRxDataHandle* and Tx DMAMUX source for *dmaIntermediaryToTxRegHandle*. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for *dmaRxRegToRxDataHandle*.

Parameters

|                                        |                                                                                   |
|----------------------------------------|-----------------------------------------------------------------------------------|
| <i>base</i>                            | DSPI peripheral base address.                                                     |
| <i>handle</i>                          | DSPI handle pointer to <code>dspi_master_dma_handle_t</code> .                    |
| <i>callback</i>                        | DSPI callback.                                                                    |
| <i>userData</i>                        | A callback function parameter.                                                    |
| <i>dmaRxRegTo-RxDataHandle</i>         | <code>dmaRxRegToRxDataHandle</code> pointer to <code>dma_handle_t</code> .        |
| <i>dmaTxDataTo-Intermediary-Handle</i> | <code>dmaTxDataToIntermediaryHandle</code> pointer to <code>dma_handle_t</code> . |
| <i>dma-Intermediary-ToTxReg-Handle</i> | <code>dmaIntermediaryToTxRegHandle</code> pointer to <code>dma_handle_t</code> .  |

#### 11.3.4.2 `status_t DSPI_MasterTransferDMA ( SPI_Type * base, dspi_master_dma_handle_t * handle, dspi_transfer_t * transfer )`

This function transfers data using DMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note that the master DMA transfer does not support the `transfer_size` of 1 when the `bitsPerFrame` is greater than 8.

Parameters

|                 |                                                                                                   |
|-----------------|---------------------------------------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                                                     |
| <i>handle</i>   | A pointer to the <code>dspi_master_dma_handle_t</code> structure which stores the transfer state. |
| <i>transfer</i> | A pointer to the <code>dspi_transfer_t</code> structure.                                          |

Returns

status of `status_t`.

#### 11.3.4.3 `void DSPI_MasterTransferAbortDMA ( SPI_Type * base, dspi_master_dma_handle_t * handle )`

This function aborts a transfer which is using DMA.

## DSPI DMA Driver

Parameters

|               |                                                                                      |
|---------------|--------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                        |
| <i>handle</i> | A pointer to the dspi_master_dma_handle_t structure which stores the transfer state. |

### 11.3.4.4 status\_t DSPI\_MasterTransferGetCountDMA ( SPI\_Type \* *base*, dspi\_master\_dma\_handle\_t \* *handle*, size\_t \* *count* )

This function gets the master DMA transfer remaining bytes.

Parameters

|               |                                                                                      |
|---------------|--------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                        |
| <i>handle</i> | A pointer to the dspi_master_dma_handle_t structure which stores the transfer state. |
| <i>count</i>  | A number of bytes transferred by the non-blocking transaction.                       |

Returns

status of status\_t.

### 11.3.4.5 void DSPI\_SlaveTransferCreateHandleDMA ( SPI\_Type \* *base*, dspi\_slave\_dma\_handle\_t \* *handle*, dspi\_slave\_dma\_transfer\_callback\_t *callback*, void \* *userData*, dma\_handle\_t \* *dmaRxRegToRxDataHandle*, dma\_handle\_t \* *dmaTxDataToTxRegHandle* )

This function initializes the DSPI DMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI DMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx is the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for dmaRxRegToRxDataHandle and Tx DMAMUX source for dmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for dmaRxRegToRxDataHandle.

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | DSPI peripheral base address. |
|-------------|-------------------------------|

|                                |                                                 |
|--------------------------------|-------------------------------------------------|
| <i>handle</i>                  | DSPI handle pointer to dspi_slave_dma_handle_t. |
| <i>callback</i>                | DSPI callback.                                  |
| <i>userData</i>                | A callback function parameter.                  |
| <i>dmaRxRegTo-RxDataHandle</i> | dmaRxRegToRxDataHandle pointer to dma_handle_t. |
| <i>dmaTxDataTo-TxRegHandle</i> | dmaTxDataToTxRegHandle pointer to dma_handle_t. |

#### 11.3.4.6 status\_t DSPI\_SlaveTransferDMA ( SPI\_Type \* *base*, dspi\_slave\_dma\_handle\_t \* *handle*, dspi\_transfer\_t \* *transfer* )

This function transfers data using DMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note that the slave DMA transfer does not support the transfer\_size of 1 when the bitsPerFrame is greater than eight.

Parameters

|                 |                                                                                     |
|-----------------|-------------------------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                                       |
| <i>handle</i>   | A pointer to the dspi_slave_dma_handle_t structure which stores the transfer state. |
| <i>transfer</i> | A pointer to the <a href="#">dspi_transfer_t</a> structure.                         |

Returns

status of status\_t.

#### 11.3.4.7 void DSPI\_SlaveTransferAbortDMA ( SPI\_Type \* *base*, dspi\_slave\_dma\_handle\_t \* *handle* )

This function aborts a transfer which is using DMA.

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | DSPI peripheral base address. |
|-------------|-------------------------------|

## DSPI DMA Driver

|               |                                                                                     |
|---------------|-------------------------------------------------------------------------------------|
| <i>handle</i> | A pointer to the dspi_slave_dma_handle_t structure which stores the transfer state. |
|---------------|-------------------------------------------------------------------------------------|

### 11.3.4.8 status\_t DSPI\_SlaveTransferGetCountDMA ( SPI\_Type \* *base*, dspi\_slave\_dma\_handle\_t \* *handle*, size\_t \* *count* )

This function gets the slave DMA transfer remaining bytes.

Parameters

|               |                                                                                     |
|---------------|-------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                       |
| <i>handle</i> | A pointer to the dspi_slave_dma_handle_t structure which stores the transfer state. |
| <i>count</i>  | A number of bytes transferred by the non-blocking transaction.                      |

Returns

status of status\_t.

## 11.4 DSPI eDMA Driver

### 11.4.1 Overview

This section describes the programming interface of the DSPI Peripheral driver. The DSPI driver configures DSPI module and provides the functional and transactional interfaces to build the DSPI application.

## Data Structures

- struct [dspi\\_master\\_edma\\_handle\\_t](#)  
*DSPI master eDMA transfer handle structure used for the transactional API. [More...](#)*
- struct [dspi\\_slave\\_edma\\_handle\\_t](#)  
*DSPI slave eDMA transfer handle structure used for the transactional API. [More...](#)*

## Typedefs

- typedef void(\* [dspi\\_master\\_edma\\_transfer\\_callback\\_t](#)) (SPI\_Type \*base, dspi\_master\_edma\_handle\_t \*handle, status\_t status, void \*userData)  
*Completion callback function pointer type.*
- typedef void(\* [dspi\\_slave\\_edma\\_transfer\\_callback\\_t](#)) (SPI\_Type \*base, dspi\_slave\_edma\_handle\_t \*handle, status\_t status, void \*userData)  
*Completion callback function pointer type.*

## Functions

- void [DSPI\\_MasterTransferCreateHandleEDMA](#) (SPI\_Type \*base, dspi\_master\_edma\_handle\_t \*handle, [dspi\\_master\\_edma\\_transfer\\_callback\\_t](#) callback, void \*userData, [edma\\_handle\\_t](#) \*edmaRxRegToRxDataHandle, [edma\\_handle\\_t](#) \*edmaTxDataToIntermediaryHandle, [edma\\_handle\\_t](#) \*edmaIntermediaryToTxRegHandle)  
*Initializes the DSPI master eDMA handle.*
- status\_t [DSPI\\_MasterTransferEDMA](#) (SPI\_Type \*base, dspi\_master\_edma\_handle\_t \*handle, [dspi\\_transfer\\_t](#) \*transfer)  
*DSPI master transfer data using eDMA.*
- void [DSPI\\_MasterTransferAbortEDMA](#) (SPI\_Type \*base, dspi\_master\_edma\_handle\_t \*handle)  
*DSPI master aborts a transfer which is using eDMA.*
- status\_t [DSPI\\_MasterTransferGetCountEDMA](#) (SPI\_Type \*base, dspi\_master\_edma\_handle\_t \*handle, size\_t \*count)  
*Gets the master eDMA transfer count.*
- void [DSPI\\_SlaveTransferCreateHandleEDMA](#) (SPI\_Type \*base, dspi\_slave\_edma\_handle\_t \*handle, [dspi\\_slave\\_edma\\_transfer\\_callback\\_t](#) callback, void \*userData, [edma\\_handle\\_t](#) \*edmaRxRegToRxDataHandle, [edma\\_handle\\_t](#) \*edmaTxDataToTxRegHandle)  
*Initializes the DSPI slave eDMA handle.*
- status\_t [DSPI\\_SlaveTransferEDMA](#) (SPI\_Type \*base, dspi\_slave\_edma\_handle\_t \*handle, [dspi\\_transfer\\_t](#) \*transfer)  
*DSPI slave transfer data using eDMA.*

## DSPI eDMA Driver

- void **DSPI\_SlaveTransferAbortEDMA** (SPI\_Type \*base, dspi\_slave\_edma\_handle\_t \*handle)  
*DSPI slave aborts a transfer which is using eDMA.*
- status\_t **DSPI\_SlaveTransferGetCountEDMA** (SPI\_Type \*base, dspi\_slave\_edma\_handle\_t \*handle, size\_t \*count)  
*Gets the slave eDMA transfer count.*

### 11.4.2 Data Structure Documentation

#### 11.4.2.1 struct \_dspi\_master\_edma\_handle

Forward declaration of the DSPI eDMA master handle typedefs.

#### Data Fields

- uint32\_t **bitsPerFrame**  
*The desired number of bits per frame.*
- volatile uint32\_t **command**  
*The desired data command.*
- volatile uint32\_t **lastCommand**  
*The desired last data command.*
- uint8\_t **fifoSize**  
*FIFO dataSize.*
- volatile bool **isPcsActiveAfterTransfer**  
*Indicates whether the PCS signal keeps active after the last frame transfer.*
- uint8\_t **nbytes**  
*eDMA minor byte transfer count initially configured.*
- volatile uint8\_t **state**  
*DSPI transfer state , \_dspi\_transfer\_state.*
- uint8\_t \*volatile **txData**  
*Send buffer.*
- uint8\_t \*volatile **rxData**  
*Receive buffer.*
- volatile size\_t **remainingSendByteCount**  
*A number of bytes remaining to send.*
- volatile size\_t **remainingReceiveByteCount**  
*A number of bytes remaining to receive.*
- size\_t **totalByteCount**  
*A number of transfer bytes.*
- uint32\_t **rxBuffIfNull**  
*Used if there is not rxData for DMA purpose.*
- uint32\_t **txBuffIfNull**  
*Used if there is not txData for DMA purpose.*
- **dspi\_master\_edma\_transfer\_callback\_t callback**  
*Completion callback.*
- void \* **userData**  
*Callback user data.*
- **edma\_handle\_t \* edmaRxRegToRxDataHandle**  
*edma\_handle\_t handle point used for RxReg to RxData buff*

- `edma_handle_t * edmaTxDataToIntermediaryHandle`  
`edma_handle_t` handle point used for TxData to Intermediary
- `edma_handle_t * edmaIntermediaryToTxRegHandle`  
`edma_handle_t` handle point used for Intermediary to TxReg
- `edma_tcd_t dspiSoftwareTCD [2]`  
`SoftwareTCD`, internal used.

#### 11.4.2.1.0.22 Field Documentation

11.4.2.1.0.22.1 `uint32_t dspi_master_edma_handle_t::bitsPerFrame`

11.4.2.1.0.22.2 `volatile uint32_t dspi_master_edma_handle_t::command`

11.4.2.1.0.22.3 `volatile uint32_t dspi_master_edma_handle_t::lastCommand`

11.4.2.1.0.22.4 `uint8_t dspi_master_edma_handle_t::fifoSize`

11.4.2.1.0.22.5 `volatile bool dspi_master_edma_handle_t::isPcsActiveAfterTransfer`

11.4.2.1.0.22.6 `uint8_t dspi_master_edma_handle_t::nbytes`

11.4.2.1.0.22.7 `volatile uint8_t dspi_master_edma_handle_t::state`

11.4.2.1.0.22.8 `uint8_t* volatile dspi_master_edma_handle_t::txData`

11.4.2.1.0.22.9 `uint8_t* volatile dspi_master_edma_handle_t::rxData`

11.4.2.1.0.22.10 `volatile size_t dspi_master_edma_handle_t::remainingSendByteCount`

11.4.2.1.0.22.11 `volatile size_t dspi_master_edma_handle_t::remainingReceiveByteCount`

11.4.2.1.0.22.12 `uint32_t dspi_master_edma_handle_t::rxBuffIfNull`

11.4.2.1.0.22.13 `uint32_t dspi_master_edma_handle_t::txBuffIfNull`

11.4.2.1.0.22.14 `dspi_master_edma_transfer_callback_t dspi_master_edma_handle_t::callback`

11.4.2.1.0.22.15 `void* dspi_master_edma_handle_t::userData`

#### 11.4.2.2 `struct _dspi_slave_edma_handle`

Forward declaration of the DSPI eDMA slave handle typedefs.

#### Data Fields

- `uint32_t bitsPerFrame`  
`The desired number of bits per frame.`
- `uint8_t *volatile txData`  
`Send buffer.`
- `uint8_t *volatile rxData`

## DSPI eDMA Driver

*Receive buffer.*

- volatile size\_t **remainingSendByteCount**  
*A number of bytes remaining to send.*
- volatile size\_t **remainingReceiveByteCount**  
*A number of bytes remaining to receive.*
- size\_t **totalByteCount**  
*A number of transfer bytes.*
- uint32\_t **rxBuffIfNull**  
*Used if there is not rxData for DMA purpose.*
- uint32\_t **txBuffIfNull**  
*Used if there is not txData for DMA purpose.*
- uint32\_t **txLastData**  
*Used if there is an extra byte when 16bits per frame for DMA purpose.*
- uint8\_t **nbytes**  
*eDMA minor byte transfer count initially configured.*
- volatile uint8\_t **state**  
*DSPI transfer state.*
- **dspi\_slave\_edma\_transfer\_callback\_t callback**  
*Completion callback.*
- void \* **userData**  
*Callback user data.*
- **edma\_handle\_t \* edmaRxRegToRxDataHandle**  
*edma\_handle\_t handle point used for RxReg to RxData buff*
- **edma\_handle\_t \* edmaTxDataToTxRegHandle**  
*edma\_handle\_t handle point used for TxData to TxReg*

#### 11.4.2.2.0.23 Field Documentation

- 11.4.2.2.0.23.1 `uint32_t dspi_slave_edma_handle_t::bitsPerFrame`
- 11.4.2.2.0.23.2 `uint8_t* volatile dspi_slave_edma_handle_t::txData`
- 11.4.2.2.0.23.3 `uint8_t* volatile dspi_slave_edma_handle_t::rxData`
- 11.4.2.2.0.23.4 `volatile size_t dspi_slave_edma_handle_t::remainingSendByteCount`
- 11.4.2.2.0.23.5 `volatile size_t dspi_slave_edma_handle_t::remainingReceiveByteCount`
- 11.4.2.2.0.23.6 `uint32_t dspi_slave_edma_handle_t::rxBuffIfNull`
- 11.4.2.2.0.23.7 `uint32_t dspi_slave_edma_handle_t::txBuffIfNull`
- 11.4.2.2.0.23.8 `uint32_t dspi_slave_edma_handle_t::txLastData`
- 11.4.2.2.0.23.9 `uint8_t dspi_slave_edma_handle_t::nbytes`
- 11.4.2.2.0.23.10 `volatile uint8_t dspi_slave_edma_handle_t::state`
- 11.4.2.2.0.23.11 `dspi_slave_edma_transfer_callback_t dspi_slave_edma_handle_t::callback`
- 11.4.2.2.0.23.12 `void* dspi_slave_edma_handle_t::userData`

#### 11.4.3 Typedef Documentation

- 11.4.3.1 `typedef void(* dspi_master_edma_transfer_callback_t)(SPI_Type *base, dspi_master_edma_handle_t *handle, status_t status, void *userData)`

## DSPI eDMA Driver

Parameters

|                 |                                                                   |
|-----------------|-------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                     |
| <i>handle</i>   | A pointer to the handle for the DSPI master.                      |
| <i>status</i>   | Success or error code describing whether the transfer completed.  |
| <i>userData</i> | An arbitrary pointer-dataSized value passed from the application. |

### 11.4.3.2 **typedef void(\* dspi\_slave\_edma\_transfer\_callback\_t)(SPI\_Type \*base, dspi\_slave\_edma\_handle\_t \*handle, status\_t status, void \*userData)**

Parameters

|                 |                                                                   |
|-----------------|-------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                     |
| <i>handle</i>   | A pointer to the handle for the DSPI slave.                       |
| <i>status</i>   | Success or error code describing whether the transfer completed.  |
| <i>userData</i> | An arbitrary pointer-dataSized value passed from the application. |

## 11.4.4 Function Documentation

### 11.4.4.1 **void DSPI\_MasterTransferCreateHandleEDMA ( SPI\_Type \* base, dspi\_master\_edma\_handle\_t \* handle, dspi\_master\_edma\_transfer\_callback\_t callback, void \* userData, edma\_handle\_t \* edmaRxRegToRxDataHandle, edma\_handle\_t \* edmaTxDataToIntermediaryHandle, edma\_handle\_t \* edmaIntermediaryToTxRegHandle )**

This function initializes the DSPI eDMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI eDMA has separated (RX and TX as two sources) or shared (RX and TX are the same source) DMA request source. (1) For the separated DMA request source, enable and set the RX DMAMUX source for edmaRxRegToRxDataHandle and TX DMAMUX source for edmaIntermediaryToTxRegHandle. (2) For the shared DMA request source, enable and set the RX/RX DMAMUX source for the edmaRxRegToRxDataHandle.

Parameters

|                                          |                                                                                     |
|------------------------------------------|-------------------------------------------------------------------------------------|
| <i>base</i>                              | DSPI peripheral base address.                                                       |
| <i>handle</i>                            | DSPI handle pointer to <code>dspi_master_edma_handle_t</code> .                     |
| <i>callback</i>                          | DSPI callback.                                                                      |
| <i>userData</i>                          | A callback function parameter.                                                      |
| <i>edmaRxRegTo-RxDataHandle</i>          | <code>edmaRxRegToRxDataHandle</code> pointer to <code>edma_handle_t</code> .        |
| <i>edmaTxData-To-Intermediary-Handle</i> | <code>edmaTxDataToIntermediaryHandle</code> pointer to <code>edma_handle_t</code> . |
| <i>edma-Intermediary-ToTxReg-Handle</i>  | <code>edmaIntermediaryToTxRegHandle</code> pointer to <code>edma_handle_t</code> .  |

#### 11.4.4.2 `status_t DSPI_MasterTransferEDMA ( SPI_Type * base, dspi_master_edma_handle_t * handle, dspi_transfer_t * transfer )`

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

|                 |                                                                                                    |
|-----------------|----------------------------------------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                                                      |
| <i>handle</i>   | A pointer to the <code>dspi_master_edma_handle_t</code> structure which stores the transfer state. |
| <i>transfer</i> | A pointer to the <code>dspi_transfer_t</code> structure.                                           |

Returns

status of `status_t`.

#### 11.4.4.3 `void DSPI_MasterTransferAbortEDMA ( SPI_Type * base, dspi_master_edma_handle_t * handle )`

This function aborts a transfer which is using eDMA.

## DSPI eDMA Driver

Parameters

|               |                                                                                       |
|---------------|---------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                         |
| <i>handle</i> | A pointer to the dspi_master_edma_handle_t structure which stores the transfer state. |

### 11.4.4.4 status\_t DSPI\_MasterTransferGetCountEDMA ( SPI\_Type \* *base*, dspi\_master\_edma\_handle\_t \* *handle*, size\_t \* *count* )

This function gets the master eDMA transfer count.

Parameters

|               |                                                                                       |
|---------------|---------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                         |
| <i>handle</i> | A pointer to the dspi_master_edma_handle_t structure which stores the transfer state. |
| <i>count</i>  | A number of bytes transferred by the non-blocking transaction.                        |

Returns

status of status\_t.

### 11.4.4.5 void DSPI\_SlaveTransferCreateHandleEDMA ( SPI\_Type \* *base*, dspi\_slave\_edma\_handle\_t \* *handle*, dspi\_slave\_edma\_transfer\_callback\_t *callback*, void \* *userData*, edma\_handle\_t \* *edmaRxRegToRxDataHandle*, edma\_handle\_t \* *edmaTxDataToTxRegHandle* )

This function initializes the DSPI eDMA handle which can be used for other DSPI transactional APIs. Usually, for a specified DSPI instance, call this API once to get the initialized handle.

Note that DSPI eDMA has separated (RN and TX in 2 sources) or shared (RX and TX are the same source) DMA request source. (1)For the separated DMA request source, enable and set the RX DMAMUX source for edmaRxRegToRxDataHandle and TX DMAMUX source for edmaTxDataToTxRegHandle. (2)For the shared DMA request source, enable and set the RX/RX DMAMUX source for the edmaRxRegToRxDataHandle.

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | DSPI peripheral base address. |
|-------------|-------------------------------|

|                                  |                                                                 |
|----------------------------------|-----------------------------------------------------------------|
| <i>handle</i>                    | DSPI handle pointer to <code>dspi_slave_edma_handle_t</code> .  |
| <i>callback</i>                  | DSPI callback.                                                  |
| <i>userData</i>                  | A callback function parameter.                                  |
| <i>edmaRxRegTo-RxDataHandle</i>  | edmaRxRegToRxDataHandle pointer to <code>edma_handle_t</code> . |
| <i>edmaTxData-ToTxReg-Handle</i> | edmaTxDataToTxRegHandle pointer to <code>edma_handle_t</code> . |

#### 11.4.4.6 `status_t DSPI_SlaveTransferEDMA ( SPI_Type * base, dspi_slave_edma_handle_t * handle, dspi_transfer_t * transfer )`

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called. Note that the slave eDMA transfer doesn't support transfer\_size is 1 when the bitsPerFrame is greater than eight.

Parameters

|                 |                                                                                                   |
|-----------------|---------------------------------------------------------------------------------------------------|
| <i>base</i>     | DSPI peripheral base address.                                                                     |
| <i>handle</i>   | A pointer to the <code>dspi_slave_edma_handle_t</code> structure which stores the transfer state. |
| <i>transfer</i> | A pointer to the <code>dspi_transfer_t</code> structure.                                          |

Returns

status of `status_t`.

#### 11.4.4.7 `void DSPI_SlaveTransferAbortEDMA ( SPI_Type * base, dspi_slave_edma_handle_t * handle )`

This function aborts a transfer which is using eDMA.

Parameters

|               |                                                                                                   |
|---------------|---------------------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                                     |
| <i>handle</i> | A pointer to the <code>dspi_slave_edma_handle_t</code> structure which stores the transfer state. |

#### 11.4.4.8 `status_t DSPI_SlaveTransferGetCountEDMA ( SPI_Type * base, dspi_slave_edma_handle_t * handle, size_t * count )`

This function gets the slave eDMA transfer count.

## DSPI eDMA Driver

Parameters

|               |                                                                                      |
|---------------|--------------------------------------------------------------------------------------|
| <i>base</i>   | DSPI peripheral base address.                                                        |
| <i>handle</i> | A pointer to the dspi_slave_edma_handle_t structure which stores the transfer state. |
| <i>count</i>  | A number of bytes transferred so far by the non-blocking transaction.                |

Returns

status of status\_t.

## 11.5 DSPI FreeRTOS Driver

### 11.5.1 Overview

#### DSPI RTOS Operation

- status\_t **DSPI\_RTOS\_Init** (dspi\_rtos\_handle\_t \*handle, SPI\_Type \*base, const dspi\_master\_config\_t \*masterConfig, uint32\_t srcClock\_Hz)  
*Initializes the DSPI.*
- status\_t **DSPI\_RTOS\_Deinit** (dspi\_rtos\_handle\_t \*handle)  
*Deinitializes the DSPI.*
- status\_t **DSPI\_RTOS\_Transfer** (dspi\_rtos\_handle\_t \*handle, dspi\_transfer\_t \*transfer)  
*Performs the SPI transfer.*

### 11.5.2 Function Documentation

#### 11.5.2.1 status\_t DSPI\_RTOS\_Init ( *dspi\_rtos\_handle\_t \* handle, SPI\_Type \* base, const dspi\_master\_config\_t \* masterConfig, uint32\_t srcClock\_Hz* )

This function initializes the DSPI module and the related RTOS context.

Parameters

|                     |                                                                           |
|---------------------|---------------------------------------------------------------------------|
| <i>handle</i>       | The RTOS DSPI handle, the pointer to an allocated space for RTOS context. |
| <i>base</i>         | The pointer base address of the DSPI instance to initialize.              |
| <i>masterConfig</i> | A configuration structure to set-up the DSPI in master mode.              |
| <i>srcClock_Hz</i>  | A frequency of the input clock of the DSPI module.                        |

Returns

status of the operation.

#### 11.5.2.2 status\_t DSPI\_RTOS\_Deinit ( *dspi\_rtos\_handle\_t \* handle* )

This function deinitializes the DSPI module and the related RTOS context.

Parameters

## DSPI FreeRTOS Driver

|               |                       |
|---------------|-----------------------|
| <i>handle</i> | The RTOS DSPI handle. |
|---------------|-----------------------|

### 11.5.2.3 **status\_t DSPI\_RTOS\_Transfer ( *dspi\_rtos\_handle\_t \* handle*, *dspi\_transfer\_t \* transfer* )**

This function performs the SPI transfer according to the data given in the transfer structure.

#### Parameters

|                 |                                                 |
|-----------------|-------------------------------------------------|
| <i>handle</i>   | The RTOS DSPI handle.                           |
| <i>transfer</i> | A structure specifying the transfer parameters. |

#### Returns

status of the operation.

# Chapter 12

## eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

### 12.1 Overview

The MCUXpresso SDK provides a peripheral driver for the enhanced Direct Memory Access (eDMA) of MCUXpresso SDK devices.

### 12.2 Typical use case

#### 12.2.1 eDMA Operation

```
edma_transfer_config_t transferConfig;
edma_config_t userConfig;
uint32_t transferDone = false;

EDMA_GetDefaultConfig(&userConfig);
EDMA_Init(DMA0, &userConfig);
EDMA_CreateHandle(&g_EDMA_Handle, DMA0, channel);
EDMA_SetCallback(&g_EDMA_Handle, EDMA_Callback, &transferDone);
EDMA_PrepTransfer(&transferConfig, srcAddr, srcWidth, destAddr, destWidth,
                  bytesEachRequest, transferBytes, kEDMA_MemoryToMemory);
EDMA_SubmitTransfer(&g_EDMA_Handle, &transferConfig, true);
EDMA_StartTransfer(&g_EDMA_Handle);
/* Waits for the eDMA transfer to finish */
while (transferDone != true);
```

## Data Structures

- struct `edma_config_t`  
*eDMA global configuration structure.* [More...](#)
- struct `edma_transfer_config_t`  
*eDMA transfer configuration* [More...](#)
- struct `edma_channel_Preemption_config_t`  
*eDMA channel priority configuration* [More...](#)
- struct `edma_minor_offset_config_t`  
*eDMA minor offset configuration* [More...](#)
- struct `edma_tcd_t`  
*eDMA TCD.* [More...](#)
- struct `edma_handle_t`  
*eDMA transfer handle structure* [More...](#)

## Macros

- #define `DMA_DCHPRI_INDEX`(channel) (((channel) & ~0x03U) | (3 - ((channel)&0x03U)))  
*Compute the offset unit from DCHPRI3.*
- #define `DMA_DCHPRIIn`(base, channel) ((volatile uint8\_t \*)(&(base->DCHPRI3)))[`DMA_DCHPRI_INDEX`(channel)]  
*Get the pointer of DCHPRIIn.*

## Typical use case

### Typedefs

- `typedef void(* edma_callback )(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tcds)`  
*Define callback function for eDMA.*

### Enumerations

- `enum edma_transfer_size_t {  
 kEDMA_TransferSize1Bytes = 0x0U,  
 kEDMA_TransferSize2Bytes = 0x1U,  
 kEDMA_TransferSize4Bytes = 0x2U,  
 kEDMA_TransferSize16Bytes = 0x4U,  
 kEDMA_TransferSize32Bytes = 0x5U }`  
*eDMA transfer configuration*
- `enum edma_modulo_t {`

```

kEDMA_ModuloDisable = 0x0U,
kEDMA_Modulo2bytes,
kEDMA_Modulo4bytes,
kEDMA_Modulo8bytes,
kEDMA_Modulo16bytes,
kEDMA_Modulo32bytes,
kEDMA_Modulo64bytes,
kEDMA_Modulo128bytes,
kEDMA_Modulo256bytes,
kEDMA_Modulo512bytes,
kEDMA_Modulo1Kbytes,
kEDMA_Modulo2Kbytes,
kEDMA_Modulo4Kbytes,
kEDMA_Modulo8Kbytes,
kEDMA_Modulo16Kbytes,
kEDMA_Modulo32Kbytes,
kEDMA_Modulo64Kbytes,
kEDMA_Modulo128Kbytes,
kEDMA_Modulo256Kbytes,
kEDMA_Modulo512Kbytes,
kEDMA_Modulo1Mbytes,
kEDMA_Modulo2Mbytes,
kEDMA_Modulo4Mbytes,
kEDMA_Modulo8Mbytes,
kEDMA_Modulo16Mbytes,
kEDMA_Modulo32Mbytes,
kEDMA_Modulo64Mbytes,
kEDMA_Modulo128Mbytes,
kEDMA_Modulo256Mbytes,
kEDMA_Modulo512Mbytes,
kEDMA_Modulo1Gbytes,
kEDMA_Modulo2Gbytes }

eDMA modulo configuration
• enum edma_bandwidth_t {
  kEDMA_BandwidthStallNone = 0x0U,
  kEDMA_BandwidthStall4Cycle = 0x2U,
  kEDMA_BandwidthStall8Cycle = 0x3U }

Bandwidth control.
• enum edma_channel_link_type_t {
  kEDMA_LinkNone = 0x0U,
  kEDMA_MinorLink,
  kEDMA_MajorLink }

Channel link type.
• enum _edma_channel_status_flags {

```

## Typical use case

- ```
kEDMA_DoneFlag = 0x1U,  
kEDMA_ErrorFlag = 0x2U,  
kEDMA_InterruptFlag = 0x4U }  
    eDMA channel status flags.  
• enum _edma_error_status_flags {  
    kEDMA_DestinationBusErrorFlag = DMA_ES_DBE_MASK,  
    kEDMA_SourceBusErrorFlag = DMA_ES_SBE_MASK,  
    kEDMA_ScatterGatherErrorFlag = DMA_ES_SGE_MASK,  
    kEDMA_NbytesErrorFlag = DMA_ES_NCE_MASK,  
    kEDMA_DestinationOffsetErrorFlag = DMA_ES_DOE_MASK,  
    kEDMA_DestinationAddressErrorFlag = DMA_ES_DAE_MASK,  
    kEDMA_SourceOffsetErrorFlag = DMA_ES_SOE_MASK,  
    kEDMA_SourceAddressErrorFlag = DMA_ES_SAE_MASK,  
    kEDMA_ErrorChannelFlag = DMA_ES_ERRCHN_MASK,  
    kEDMA_ChannelPriorityErrorFlag = DMA_ES_CPE_MASK,  
    kEDMA_TransferCanceledFlag = DMA_ES_ECX_MASK,  
    kEDMA_ValidFlag = DMA_ES_VLD_MASK }  
    eDMA channel error status flags.  
• enum edma_interrupt_enable_t {  
    kEDMA_ErrorInterruptEnable = 0x1U,  
    kEDMA_MajorInterruptEnable = DMA_CSR_INTMAJOR_MASK,  
    kEDMA_HalfInterruptEnable = DMA_CSR_INTHALF_MASK }  
    eDMA interrupt source  
• enum edma_transfer_type_t {  
    kEDMA_MemoryToMemory = 0x0U,  
    kEDMA_PeripheralToMemory,  
    kEDMA_MemoryToPeripheral }  
    eDMA transfer type  
• enum _edma_transfer_status {  
    kStatus_EDMA_QueueFull = MAKE_STATUS(kStatusGroup_EDMA, 0),  
    kStatus_EDMA_Busy = MAKE_STATUS(kStatusGroup_EDMA, 1) }  
    eDMA transfer status
```

## Driver version

- #define **FSL\_EDMA\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 1))  
*eDMA driver version*

## eDMA initialization and de-initialization

- void **EDMA\_Init** (DMA\_Type \*base, const **edma\_config\_t** \*config)  
*Initializes the eDMA peripheral.*
- void **EDMA\_Deinit** (DMA\_Type \*base)  
*Deinitializes the eDMA peripheral.*
- void **EDMA\_GetDefaultConfig** (**edma\_config\_t** \*config)  
*Gets the eDMA default configuration structure.*

## eDMA Channel Operation

- void [EDMA\\_ResetChannel](#) (DMA\_Type \*base, uint32\_t channel)  
*Sets all TCD registers to default values.*
- void [EDMA\\_SetTransferConfig](#) (DMA\_Type \*base, uint32\_t channel, const [edma\\_transfer\\_config\\_t](#) \*config, [edma\\_tcd\\_t](#) \*nextTcd)  
*Configures the eDMA transfer attribute.*
- void [EDMA\\_SetMinorOffsetConfig](#) (DMA\_Type \*base, uint32\_t channel, const [edma\\_minor\\_offset\\_config\\_t](#) \*config)  
*Configures the eDMA minor offset feature.*
- static void [EDMA\\_SetChannelPreemptionConfig](#) (DMA\_Type \*base, uint32\_t channel, const [edma\\_channel\\_Preemption\\_config\\_t](#) \*config)  
*Configures the eDMA channel preemption feature.*
- void [EDMA\\_SetChannelLink](#) (DMA\_Type \*base, uint32\_t channel, [edma\\_channel\\_link\\_type\\_t](#) type, uint32\_t linkedChannel)  
*Sets the channel link for the eDMA transfer.*
- void [EDMA\\_SetBandWidth](#) (DMA\_Type \*base, uint32\_t channel, [edma\\_bandwidth\\_t](#) bandWidth)  
*Sets the bandwidth for the eDMA transfer.*
- void [EDMA\\_SetModulo](#) (DMA\_Type \*base, uint32\_t channel, [edma\\_modulo\\_t](#) srcModulo, [edma\\_modulo\\_t](#) destModulo)  
*Sets the source modulo and the destination modulo for the eDMA transfer.*
- static void [EDMA\\_EnableAutoStopRequest](#) (DMA\_Type \*base, uint32\_t channel, bool enable)  
*Enables an auto stop request for the eDMA transfer.*
- void [EDMA\\_EnableChannelInterrupts](#) (DMA\_Type \*base, uint32\_t channel, uint32\_t mask)  
*Enables the interrupt source for the eDMA transfer.*
- void [EDMA\\_DisableChannelInterrupts](#) (DMA\_Type \*base, uint32\_t channel, uint32\_t mask)  
*Disables the interrupt source for the eDMA transfer.*

## eDMA TCD Operation

- void [EDMA\\_TcdReset](#) ([edma\\_tcd\\_t](#) \*tcd)  
*Sets all fields to default values for the TCD structure.*
- void [EDMA\\_TcdSetTransferConfig](#) ([edma\\_tcd\\_t](#) \*tcd, const [edma\\_transfer\\_config\\_t](#) \*config, [edma\\_tcd\\_t](#) \*nextTcd)  
*Configures the eDMA TCD transfer attribute.*
- void [EDMA\\_TcdSetMinorOffsetConfig](#) ([edma\\_tcd\\_t](#) \*tcd, const [edma\\_minor\\_offset\\_config\\_t](#) \*config)  
*Configures the eDMA TCD minor offset feature.*
- void [EDMA\\_TcdSetChannelLink](#) ([edma\\_tcd\\_t](#) \*tcd, [edma\\_channel\\_link\\_type\\_t](#) type, uint32\_t linkedChannel)  
*Sets the channel link for the eDMA TCD.*
- static void [EDMA\\_TcdSetBandWidth](#) ([edma\\_tcd\\_t](#) \*tcd, [edma\\_bandwidth\\_t](#) bandWidth)  
*Sets the bandwidth for the eDMA TCD.*
- void [EDMA\\_TcdSetModulo](#) ([edma\\_tcd\\_t](#) \*tcd, [edma\\_modulo\\_t](#) srcModulo, [edma\\_modulo\\_t](#) destModulo)  
*Sets the source modulo and the destination modulo for the eDMA TCD.*
- static void [EDMA\\_TcdEnableAutoStopRequest](#) ([edma\\_tcd\\_t](#) \*tcd, bool enable)  
*Enables the auto stop request for the eDMA TCD.*
- void [EDMA\\_TcdEnableInterrupts](#) ([edma\\_tcd\\_t](#) \*tcd, uint32\_t mask)  
*Enables the interrupt source for the eDMA TCD.*

## Typical use case

- void [EDMA\\_TcdDisableInterrupts](#) (`edma_tcd_t` \*`tcd`, `uint32_t` `mask`)  
*Disables the interrupt source for the eDMA TCD.*

## eDMA Channel Transfer Operation

- static void [EDMA\\_EnableChannelRequest](#) (`DMA_Type` \*`base`, `uint32_t` `channel`)  
*Enables the eDMA hardware channel request.*
- static void [EDMA\\_DisableChannelRequest](#) (`DMA_Type` \*`base`, `uint32_t` `channel`)  
*Disables the eDMA hardware channel request.*
- static void [EDMA\\_TriggerChannelStart](#) (`DMA_Type` \*`base`, `uint32_t` `channel`)  
*Starts the eDMA transfer by using the software trigger.*

## eDMA Channel Status Operation

- `uint32_t` [EDMA\\_GetRemainingMajorLoopCount](#) (`DMA_Type` \*`base`, `uint32_t` `channel`)  
*Gets the remaining major loop count from the eDMA current channel TCD.*
- static `uint32_t` [EDMA\\_GetErrorStatusFlags](#) (`DMA_Type` \*`base`)  
*Gets the eDMA channel error status flags.*
- `uint32_t` [EDMA\\_GetChannelStatusFlags](#) (`DMA_Type` \*`base`, `uint32_t` `channel`)  
*Gets the eDMA channel status flags.*
- void [EDMA\\_ClearChannelStatusFlags](#) (`DMA_Type` \*`base`, `uint32_t` `channel`, `uint32_t` `mask`)  
*Clears the eDMA channel status flags.*

## eDMA Transactional Operation

- void [EDMA\\_CreateHandle](#) (`edma_handle_t` \*`handle`, `DMA_Type` \*`base`, `uint32_t` `channel`)  
*Creates the eDMA handle.*
- void [EDMA\\_InstallTCDMemory](#) (`edma_handle_t` \*`handle`, `edma_tcd_t` \*`tcdPool`, `uint32_t` `tcdSize`)  
*Installs the TCDs memory pool into the eDMA handle.*
- void [EDMA\\_SetCallback](#) (`edma_handle_t` \*`handle`, `edma_callback` `callback`, `void` \*`userData`)  
*Installs a callback function for the eDMA transfer.*
- void [EDMA\\_PrepTransfer](#) (`edma_transfer_config_t` \*`config`, `void` \*`srcAddr`, `uint32_t` `srcWidth`,  
`void` \*`destAddr`, `uint32_t` `destWidth`, `uint32_t` `bytesEachRequest`, `uint32_t` `transferBytes`, `edma_transfer_type_t` `type`)  
*Prepares the eDMA transfer structure.*
- status\_t [EDMA\\_SubmitTransfer](#) (`edma_handle_t` \*`handle`, const `edma_transfer_config_t` \*`config`)  
*Submits the eDMA transfer request.*
- void [EDMA\\_StartTransfer](#) (`edma_handle_t` \*`handle`)  
*eDMA starts transfer.*
- void [EDMA\\_StopTransfer](#) (`edma_handle_t` \*`handle`)  
*eDMA stops transfer.*
- void [EDMA\\_AbortTransfer](#) (`edma_handle_t` \*`handle`)  
*eDMA aborts transfer.*
- void [EDMA\\_HandleIRQ](#) (`edma_handle_t` \*`handle`)  
*eDMA IRQ handler for the current major loop transfer completion.*

## 12.3 Data Structure Documentation

### 12.3.1 struct edma\_config\_t

## Data Fields

- bool `enableContinuousLinkMode`  
*Enable (true) continuous link mode.*
  - bool `enableHaltOnError`  
*Enable (true) transfer halt on error.*
  - bool `enableRoundRobinArbitration`  
*Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection.*
  - bool `enableDebugMode`  
*Enable(true) eDMA debug mode.*

**12.3.1.0.0.24 Field Documentation**

**12.3.1.0.0.24.1 bool edma\_config\_t::enableContinuousLinkMode**

Upon minor loop completion, the channel activates again if that channel has a minor loop channel link enabled and the link channel is itself.

#### **12.3.1.0.0.24.2 bool edma\_config\_t::enableHaltOnError**

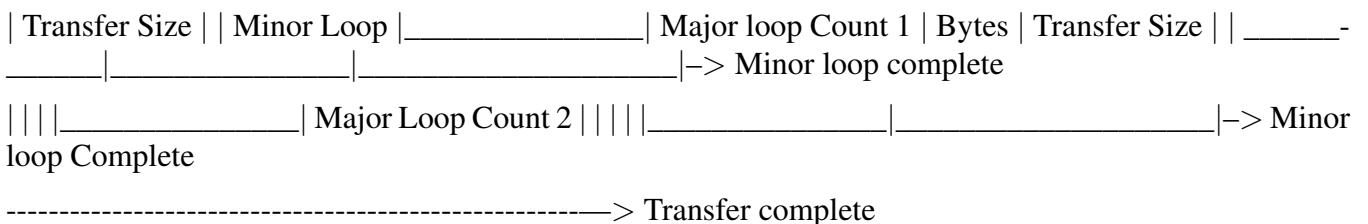
Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

### **12.3.1.0.0.24.3 bool edma\_config\_t::enableDebugMode**

When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

### 12.3.2 struct edma\_transfer\_config t

This structure configures the source/destination transfer attribute. This figure shows the eDMA's transfer model:



## Data Structure Documentation

### Data Fields

- `uint32_t srcAddr`  
*Source data address.*
- `uint32_t destAddr`  
*Destination data address.*
- `edma_transfer_size_t srcTransferSize`  
*Source data transfer size.*
- `edma_transfer_size_t destTransferSize`  
*Destination data transfer size.*
- `int16_t srcOffset`  
*Sign-extended offset applied to the current source address to form the next-state value as each source read is completed.*
- `int16_t destOffset`  
*Sign-extended offset applied to the current destination address to form the next-state value as each destination write is completed.*
- `uint32_t minorLoopBytes`  
*Bytes to transfer in a minor loop.*
- `uint32_t majorLoopCounts`  
*Major loop iteration count.*

#### 12.3.2.0.0.25 Field Documentation

12.3.2.0.0.25.1 `uint32_t edma_transfer_config_t::srcAddr`

12.3.2.0.0.25.2 `uint32_t edma_transfer_config_t::destAddr`

12.3.2.0.0.25.3 `edma_transfer_size_t edma_transfer_config_t::srcTransferSize`

12.3.2.0.0.25.4 `edma_transfer_size_t edma_transfer_config_t::destTransferSize`

12.3.2.0.0.25.5 `int16_t edma_transfer_config_t::srcOffset`

12.3.2.0.0.25.6 `int16_t edma_transfer_config_t::destOffset`

12.3.2.0.0.25.7 `uint32_t edma_transfer_config_t::majorLoopCounts`

### 12.3.3 struct `edma_channel_Preemption_config_t`

### Data Fields

- `bool enableChannelPreemption`  
*If true: a channel can be suspended by other channel with higher priority.*
- `bool enablePreemptAbility`  
*If true: a channel can suspend other channel with low priority.*
- `uint8_t channelPriority`  
*Channel priority.*

### 12.3.4 struct edma\_minor\_offset\_config\_t

#### Data Fields

- bool `enableSrcMinorOffset`  
*Enable(true) or Disable(false) source minor loop offset.*
- bool `enableDestMinorOffset`  
*Enable(true) or Disable(false) destination minor loop offset.*
- uint32\_t `minorOffset`  
*Offset for a minor loop mapping.*

#### 12.3.4.0.0.26 Field Documentation

##### 12.3.4.0.0.26.1 bool edma\_minor\_offset\_config\_t::enableSrcMinorOffset

##### 12.3.4.0.0.26.2 bool edma\_minor\_offset\_config\_t::enableDestMinorOffset

##### 12.3.4.0.0.26.3 uint32\_t edma\_minor\_offset\_config\_t::minorOffset

### 12.3.5 struct edma\_tcd\_t

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

#### Data Fields

- \_\_IO uint32\_t `SADDR`  
*SADDR register, used to save source address.*
- \_\_IO uint16\_t `SOFF`  
*SOFF register, save offset bytes every transfer.*
- \_\_IO uint16\_t `ATTR`  
*ATTR register, source/destination transfer size and modulo.*
- \_\_IO uint32\_t `NBYTES`  
*Nbytes register, minor loop length in bytes.*
- \_\_IO uint32\_t `SLAST`  
*SLAST register.*
- \_\_IO uint32\_t `DADDR`  
*DADDR register, used for destination address.*
- \_\_IO uint16\_t `DOFF`  
*DOFF register, used for destination offset.*
- \_\_IO uint16\_t `CITER`  
*CITER register, current minor loop numbers, for unfinished minor loop.*
- \_\_IO uint32\_t `DLAST_SGA`  
*DLASTSGA register, next stcd address used in scatter-gather mode.*
- \_\_IO uint16\_t `CSR`  
*CSR register, for TCD control status.*
- \_\_IO uint16\_t `BITER`  
*BITER register, begin minor loop count.*

## Data Structure Documentation

### 12.3.5.0.0.27 Field Documentation

12.3.5.0.0.27.1 `__IO uint16_t edma_tcd_t::CITER`

12.3.5.0.0.27.2 `__IO uint16_t edma_tcd_t::BITER`

### 12.3.6 struct `edma_handle_t`

#### Data Fields

- `edma_callback callback`  
*Callback function for major count exhausted.*
- `void * userData`  
*Callback function parameter.*
- `DMA_Type * base`  
*eDMA peripheral base address.*
- `edma_tcd_t * tcdPool`  
*Pointer to memory stored TCDs.*
- `uint8_t channel`  
*eDMA channel number.*
- `volatile int8_t header`  
*The first TCD index.*
- `volatile int8_t tail`  
*The last TCD index.*
- `volatile int8_t tcdUsed`  
*The number of used TCD slots.*
- `volatile int8_t tcdSize`  
*The total number of TCD slots in the queue.*
- `uint8_t flags`  
*The status of the current channel.*

### 12.3.6.0.0.28 Field Documentation

12.3.6.0.0.28.1 `edma_callback edma_handle_t::callback`

12.3.6.0.0.28.2 `void* edma_handle_t::userData`

12.3.6.0.0.28.3 `DMA_Type* edma_handle_t::base`

12.3.6.0.0.28.4 `edma_tcd_t* edma_handle_t::tcdPool`

12.3.6.0.0.28.5 `uint8_t edma_handle_t::channel`

12.3.6.0.0.28.6 `volatile int8_t edma_handle_t::header`

Should point to the next TCD to be loaded into the eDMA engine.

12.3.6.0.0.28.7 `volatile int8_t edma_handle_t::tail`

Should point to the next TCD to be stored into the memory pool.

**12.3.6.0.0.28.8 volatile int8\_t edma\_handle\_t::tcdUsed**

Should reflect the number of TCDs can be used/loaded in the memory.

**12.3.6.0.0.28.9 volatile int8\_t edma\_handle\_t::tcdSize****12.3.6.0.0.28.10 uint8\_t edma\_handle\_t::flags****12.4 Macro Definition Documentation****12.4.1 #define FSL\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 1))**

Version 2.1.1.

**12.5 Typedef Documentation****12.5.1 typedef void(\* edma\_callback)(struct \_edma\_handle \*handle, void \*userData, bool transferDone, uint32\_t tclds)****12.6 Enumeration Type Documentation****12.6.1 enum edma\_transfer\_size\_t**

Enumerator

*kEDMA\_TransferSize1Bytes* Source/Destination data transfer size is 1 byte every time.

*kEDMA\_TransferSize2Bytes* Source/Destination data transfer size is 2 bytes every time.

*kEDMA\_TransferSize4Bytes* Source/Destination data transfer size is 4 bytes every time.

*kEDMA\_TransferSize16Bytes* Source/Destination data transfer size is 16 bytes every time.

*kEDMA\_TransferSize32Bytes* Source/Destination data transfer size is 32 bytes every time.

**12.6.2 enum edma\_modulo\_t**

Enumerator

*kEDMA\_ModuloDisable* Disable modulo.

*kEDMA\_Modulo2bytes* Circular buffer size is 2 bytes.

*kEDMA\_Modulo4bytes* Circular buffer size is 4 bytes.

*kEDMA\_Modulo8bytes* Circular buffer size is 8 bytes.

*kEDMA\_Modulo16bytes* Circular buffer size is 16 bytes.

*kEDMA\_Modulo32bytes* Circular buffer size is 32 bytes.

*kEDMA\_Modulo64bytes* Circular buffer size is 64 bytes.

*kEDMA\_Modulo128bytes* Circular buffer size is 128 bytes.

*kEDMA\_Modulo256bytes* Circular buffer size is 256 bytes.

*kEDMA\_Modulo512bytes* Circular buffer size is 512 bytes.

*kEDMA\_Modulo1Kbytes* Circular buffer size is 1 K bytes.

## Enumeration Type Documentation

<i>kEDMA_Modulo2Kbytes</i>	Circular buffer size is 2 K bytes.
<i>kEDMA_Modulo4Kbytes</i>	Circular buffer size is 4 K bytes.
<i>kEDMA_Modulo8Kbytes</i>	Circular buffer size is 8 K bytes.
<i>kEDMA_Modulo16Kbytes</i>	Circular buffer size is 16 K bytes.
<i>kEDMA_Modulo32Kbytes</i>	Circular buffer size is 32 K bytes.
<i>kEDMA_Modulo64Kbytes</i>	Circular buffer size is 64 K bytes.
<i>kEDMA_Modulo128Kbytes</i>	Circular buffer size is 128 K bytes.
<i>kEDMA_Modulo256Kbytes</i>	Circular buffer size is 256 K bytes.
<i>kEDMA_Modulo512Kbytes</i>	Circular buffer size is 512 K bytes.
<i>kEDMA_Modulo1Mbytes</i>	Circular buffer size is 1 M bytes.
<i>kEDMA_Modulo2Mbytes</i>	Circular buffer size is 2 M bytes.
<i>kEDMA_Modulo4Mbytes</i>	Circular buffer size is 4 M bytes.
<i>kEDMA_Modulo8Mbytes</i>	Circular buffer size is 8 M bytes.
<i>kEDMA_Modulo16Mbytes</i>	Circular buffer size is 16 M bytes.
<i>kEDMA_Modulo32Mbytes</i>	Circular buffer size is 32 M bytes.
<i>kEDMA_Modulo64Mbytes</i>	Circular buffer size is 64 M bytes.
<i>kEDMA_Modulo128Mbytes</i>	Circular buffer size is 128 M bytes.
<i>kEDMA_Modulo256Mbytes</i>	Circular buffer size is 256 M bytes.
<i>kEDMA_Modulo512Mbytes</i>	Circular buffer size is 512 M bytes.
<i>kEDMA_Modulo1Gbytes</i>	Circular buffer size is 1 G bytes.
<i>kEDMA_Modulo2Gbytes</i>	Circular buffer size is 2 G bytes.

### 12.6.3 enum edma\_bandwidth\_t

Enumerator

<i>kEDMA_BandwidthStallNone</i>	No eDMA engine stalls.
<i>kEDMA_BandwidthStall4Cycle</i>	eDMA engine stalls for 4 cycles after each read/write.
<i>kEDMA_BandwidthStall8Cycle</i>	eDMA engine stalls for 8 cycles after each read/write.

### 12.6.4 enum edma\_channel\_link\_type\_t

Enumerator

<i>kEDMA_LinkNone</i>	No channel link.
<i>kEDMA_MinorLink</i>	Channel link after each minor loop.
<i>kEDMA_MajorLink</i>	Channel link while major loop count exhausted.

### 12.6.5 enum \_edma\_channel\_status\_flags

Enumerator

<i>kEDMA_DoneFlag</i>	DONE flag, set while transfer finished, CITER value exhausted.
-----------------------	--

***kEDMA\_ErrorFlag*** eDMA error flag, an error occurred in a transfer

***kEDMA\_InterruptFlag*** eDMA interrupt flag, set while an interrupt occurred of this channel

## 12.6.6 enum \_edma\_error\_status\_flags

Enumerator

***kEDMA\_DestinationBusErrorFlag*** Bus error on destination address.

***kEDMA\_SourceBusErrorFlag*** Bus error on the source address.

***kEDMA\_ScatterGatherErrorFlag*** Error on the Scatter/Gather address, not 32byte aligned.

***kEDMA\_NbytesErrorFlag*** NBYTES/CITER configuration error.

***kEDMA\_DestinationOffsetErrorFlag*** Destination offset not aligned with destination size.

***kEDMA\_DestinationAddressErrorFlag*** Destination address not aligned with destination size.

***kEDMA\_SourceOffsetErrorFlag*** Source offset not aligned with source size.

***kEDMA\_SourceAddressErrorFlag*** Source address not aligned with source size.

***kEDMA\_ErrorChannelFlag*** Error channel number of the cancelled channel number.

***kEDMA\_ChannelPriorityErrorFlag*** Channel priority is not unique.

***kEDMA\_TransferCanceledFlag*** Transfer cancelled.

***kEDMA\_ValidFlag*** No error occurred, this bit is 0. Otherwise, it is 1.

## 12.6.7 enum edma\_interrupt\_enable\_t

Enumerator

***kEDMA\_ErrorInterruptEnable*** Enable interrupt while channel error occurs.

***kEDMA\_MajorInterruptEnable*** Enable interrupt while major count exhausted.

***kEDMA\_HalfInterruptEnable*** Enable interrupt while major count to half value.

## 12.6.8 enum edma\_transfer\_type\_t

Enumerator

***kEDMA\_MemoryToMemory*** Transfer from memory to memory.

***kEDMA\_PeripheralToMemory*** Transfer from peripheral to memory.

***kEDMA\_MemoryToPeripheral*** Transfer from memory to peripheral.

## 12.6.9 enum \_edma\_transfer\_status

Enumerator

***kStatus\_EDMA\_QueueFull*** TCD queue is full.

***kStatus\_EDMA\_Busy*** Channel is busy and can't handle the transfer request.

## Function Documentation

### 12.7 Function Documentation

#### 12.7.1 void EDMA\_Init ( DMA\_Type \* *base*, const edma\_config\_t \* *config* )

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>config</i>	A pointer to the configuration structure, see "edma_config_t".

Note

This function enables the minor loop map feature.

### 12.7.2 void EDMA\_Deinit ( DMA\_Type \* *base* )

This function gates the eDMA clock.

Parameters

<i>base</i>	eDMA peripheral base address.
-------------	-------------------------------

### 12.7.3 void EDMA\_GetDefaultConfig ( edma\_config\_t \* *config* )

This function sets the configuration structure to default values. The default configuration is set to the following values.

```
* config.enableContinuousLinkMode = false;
* config.enableHaltOnError = true;
* config.enableRoundRobinArbitration = false;
* config.enableDebugMode = false;
*
```

Parameters

<i>config</i>	A pointer to the eDMA configuration structure.
---------------	--

### 12.7.4 void EDMA\_ResetChannel ( DMA\_Type \* *base*, uint32\_t *channel* )

This function sets TCD registers for this channel to default values.

## Function Documentation

### Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

### Note

This function must not be called while the channel transfer is ongoing or it causes unpredictable results.

This function enables the auto stop request feature.

### **12.7.5 void EDMA\_SetTransferConfig ( DMA\_Type \* *base*, uint32\_t *channel*, const edma\_transfer\_config\_t \* *config*, edma\_tcd\_t \* *nextTcd* )**

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address.

Example:

```
* edma_transfer_t config;
* edma_tcd_t tcd;
* config.srcAddr = ...;
* config.destAddr = ...;
*
* ...
* EDMA_SetTransferConfig(DMA0, channel, &config, &tcd);
*
```

### Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>config</i>	Pointer to eDMA transfer configuration structure.
<i>nextTcd</i>	Point to TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

### Note

If nextTcd is not NULL, it means scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the eDMA\_ResetChannel.

### **12.7.6 void EDMA\_SetMinorOffsetConfig ( DMA\_Type \* *base*, uint32\_t *channel*, const edma\_minor\_offset\_config\_t \* *config* )**

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>config</i>	A pointer to the minor offset configuration structure.

**12.7.7 static void EDMA\_SetChannelPreemptionConfig ( DMA\_Type \* *base*, uint32\_t *channel*, const edma\_channel\_Preemption\_config\_t \* *config* ) [inline], [static]**

This function configures the channel preemption attribute and the priority of the channel.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number
<i>config</i>	A pointer to the channel preemption configuration structure.

**12.7.8 void EDMA\_SetChannelLink ( DMA\_Type \* *base*, uint32\_t *channel*, edma\_channel\_link\_type\_t *type*, uint32\_t *linkedChannel* )**

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>type</i>	A channel link type, which can be one of the following: <ul style="list-style-type: none"><li>• kEDMA_LinkNone</li><li>• kEDMA_MinorLink</li><li>• kEDMA_MajorLink</li></ul>

## Function Documentation

<i>linkedChannel</i>	The linked channel number.
----------------------	----------------------------

Note

Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

### **12.7.9 void EDMA\_SetBandWidth ( DMA\_Type \* *base*, uint32\_t *channel*, edma\_bandwidth\_t *bandWidth* )**

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>bandWidth</i>	A bandwidth setting, which can be one of the following: <ul style="list-style-type: none"><li>• kEDMABandwidthStallNone</li><li>• kEDMABandwidthStall4Cycle</li><li>• kEDMABandwidthStall8Cycle</li></ul>

### **12.7.10 void EDMA\_SetModulo ( DMA\_Type \* *base*, uint32\_t *channel*, edma\_modulo\_t *srcModulo*, edma\_modulo\_t *destModulo* )**

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

<i>srcModulo</i>	A source modulo value.
<i>destModulo</i>	A destination modulo value.

### 12.7.11 static void EDMA\_EnableAutoStopRequest ( DMA\_Type \* *base*, uint32\_t *channel*, bool *enable* ) [inline], [static]

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>enable</i>	The command to enable (true) or disable (false).

### 12.7.12 void EDMA\_EnableChannelInterrupts ( DMA\_Type \* *base*, uint32\_t *channel*, uint32\_t *mask* )

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>mask</i>	The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

### 12.7.13 void EDMA\_DisableChannelInterrupts ( DMA\_Type \* *base*, uint32\_t *channel*, uint32\_t *mask* )

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

## Function Documentation

<i>mask</i>	The mask of the interrupt source to be set. Use the defined edma_interrupt_enable_t type.
-------------	---

### 12.7.14 void EDMA\_TcdReset ( *edma\_tcd\_t \* tcd* )

This function sets all fields for this TCD structure to default value.

Parameters

<i>tcd</i>	Pointer to the TCD structure.
------------	-------------------------------

Note

This function enables the auto stop request feature.

### 12.7.15 void EDMA\_TcdSetTransferConfig ( *edma\_tcd\_t \* tcd, const edma\_transfer\_config\_t \* config, edma\_tcd\_t \* nextTcd* )

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The STCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```
*     edma_transfer_t config = {  
*     ...  
*     }  
*     edma_tcd_t tcd __aligned(32);  
*     edma_tcd_t nextTcd __aligned(32);  
*     EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);  
*
```

Parameters

<i>tcd</i>	Pointer to the TCD structure.
<i>config</i>	Pointer to eDMA transfer configuration structure.
<i>nextTcd</i>	Pointer to the next TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

Note

TCD address should be 32 bytes aligned or it causes an eDMA error.

If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA\_TcdReset.

**12.7.16 void EDMA\_TcdSetMinorOffsetConfig ( *edma\_tcd\_t* \* *tcד*, *const edma\_minor\_offset\_config\_t* \* *config* )**

A minor offset is a signed-extended value added to the source address or a destination address after each minor loop.

## Function Documentation

Parameters

<i>tcd</i>	A point to the TCD structure.
<i>config</i>	A pointer to the minor offset configuration structure.

### **12.7.17 void EDMA\_TcdSetChannelLink ( *edma\_tcd\_t \* tcd*, *edma\_channel\_link\_type\_t type*, *uint32\_t linkedChannel* )**

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note

Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

<i>tcd</i>	Point to the TCD structure.
<i>type</i>	Channel link type, it can be one of: <ul style="list-style-type: none"><li>• kEDMA_LinkNone</li><li>• kEDMA_MinorLink</li><li>• kEDMA_MajorLink</li></ul>
<i>linkedChannel</i>	The linked channel number.

### **12.7.18 static void EDMA\_TcdSetBandWidth ( *edma\_tcd\_t \* tcd*, *edma\_bandwidth\_t bandWidth* ) [inline], [static]**

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

<i>tcd</i>	A pointer to the TCD structure.
<i>bandWidth</i>	A bandwidth setting, which can be one of the following: <ul style="list-style-type: none"><li>• kEDMABandwidthStallNone</li><li>• kEDMABandwidthStall4Cycle</li><li>• kEDMABandwidthStall8Cycle</li></ul>

### **12.7.19 void EDMA\_TcdSetModulo ( edma\_tcd\_t \* *tcd*, edma\_modulo\_t *srcModulo*, edma\_modulo\_t *destModulo* )**

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

<i>tcd</i>	A pointer to the TCD structure.
<i>srcModulo</i>	A source modulo value.
<i>destModulo</i>	A destination modulo value.

### **12.7.20 static void EDMA\_TcdEnableAutoStopRequest ( edma\_tcd\_t \* *tcd*, bool *enable* ) [inline], [static]**

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

<i>tcd</i>	A pointer to the TCD structure.
<i>enable</i>	The command to enable (true) or disable (false).

### **12.7.21 void EDMA\_TcdEnableInterrupts ( edma\_tcd\_t \* *tcd*, uint32\_t *mask* )**

Parameters

## Function Documentation

<i>tcd</i>	Point to the TCD structure.
<i>mask</i>	The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

### 12.7.22 void EDMA\_TcdDisableInterrupts ( *edma\_tcd\_t \* tcd, uint32\_t mask* )

Parameters

<i>tcd</i>	Point to the TCD structure.
<i>mask</i>	The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

### 12.7.23 static void EDMA\_EnableChannelRequest ( *DMA\_Type \* base, uint32\_t channel* ) [inline], [static]

This function enables the hardware channel request.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

### 12.7.24 static void EDMA\_DisableChannelRequest ( *DMA\_Type \* base, uint32\_t channel* ) [inline], [static]

This function disables the hardware channel request.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

### 12.7.25 static void EDMA\_TriggerChannelStart ( *DMA\_Type \* base, uint32\_t channel* ) [inline], [static]

This function starts a minor loop transfer.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

### 12.7.26 **uint32\_t EDMA\_GetRemainingMajorLoopCount ( DMA\_Type \* *base*, uint32\_t *channel* )**

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the the number of major loop count that has not finished.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

Returns

Major loop count which has not been transferred yet for the current TCD.

Note

1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.
1. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA\_TCDn\_NBYTES\_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma\_engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: RemainingBytes = RemainingMajorLoopCount \* NBYTES(initially configured)

### 12.7.27 **static uint32\_t EDMA\_GetErrorStatusFlags ( DMA\_Type \* *base* ) [inline], [static]**

## Function Documentation

Parameters

<i>base</i>	eDMA peripheral base address.
-------------	-------------------------------

Returns

The mask of error status flags. Users need to use the \_edma\_error\_status\_flags type to decode the return variables.

### 12.7.28 **uint32\_t EDMA\_GetChannelStatusFlags ( DMA\_Type \* *base*, uint32\_t *channel* )**

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

Returns

The mask of channel status flags. Users need to use the \_edma\_channel\_status\_flags type to decode the return variables.

### 12.7.29 **void EDMA\_ClearChannelStatusFlags ( DMA\_Type \* *base*, uint32\_t *channel*, uint32\_t *mask* )**

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>mask</i>	The mask of channel status to be cleared. Users need to use the defined _edma_channel_status_flags type.

### 12.7.30 **void EDMA\_CreateHandle ( edma\_handle\_t \* *handle*, DMA\_Type \* *base*, uint32\_t *channel* )**

This function is called if using the transactional API for eDMA. This function initializes the internal state of the eDMA handle.

Parameters

<i>handle</i>	eDMA handle pointer. The eDMA handle stores callback function and parameters.
<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.

### **12.7.31 void EDMA\_InstallTCDMemory( edma\_handle\_t \* *handle*, edma\_tcd\_t \* *tcdPool*, uint32\_t *tcdSize* )**

This function is called after the EDMA\_CreateHandle to use scatter/gather feature.

Parameters

<i>handle</i>	eDMA handle pointer.
<i>tcdPool</i>	A memory pool to store TCDs. It must be 32 bytes aligned.
<i>tcdSize</i>	The number of TCD slots.

### **12.7.32 void EDMA\_SetCallback( edma\_handle\_t \* *handle*, edma\_callback *callback*, void \* *userData* )**

This callback is called in the eDMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

Parameters

<i>handle</i>	eDMA handle pointer.
<i>callback</i>	eDMA callback function pointer.
<i>userData</i>	A parameter for the callback function.

### **12.7.33 void EDMA\_PreparesTransfer( edma\_transfer\_config\_t \* *config*, void \* *srcAddr*, uint32\_t *srcWidth*, void \* *destAddr*, uint32\_t *destWidth*, uint32\_t *bytesEachRequest*, uint32\_t *transferBytes*, edma\_transfer\_type\_t *type* )**

This function prepares the transfer configuration structure according to the user input.

## Function Documentation

Parameters

<i>config</i>	The user configuration structure of type edma_transfer_t.
<i>srcAddr</i>	eDMA transfer source address.
<i>srcWidth</i>	eDMA transfer source address width(bytes).
<i>destAddr</i>	eDMA transfer destination address.
<i>destWidth</i>	eDMA transfer destination address width(bytes).
<i>bytesEachRequest</i>	eDMA transfer bytes per channel request.
<i>transferBytes</i>	eDMA transfer bytes to be transferred.
<i>type</i>	eDMA transfer type.

Note

The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

### 12.7.34 **status\_t EDMA\_SubmitTransfer ( edma\_handle\_t \* *handle*, const edma\_transfer\_config\_t \* *config* )**

This function submits the eDMA transfer request according to the transfer configuration structure. If submitting the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

<i>handle</i>	eDMA handle pointer.
<i>config</i>	Pointer to eDMA transfer configuration structure.

Return values

<i>kStatus_EDMA_Success</i>	It means submit transfer request succeed.
<i>kStatus_EDMA_Queue-Full</i>	It means TCD queue is full. Submit transfer request is not allowed.

<i>kStatus_EDMA_Busy</i>	It means the given channel is busy, need to submit request later.
--------------------------	---

### 12.7.35 void EDMA\_StartTransfer ( *edma\_handle\_t \* handle* )

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

<i>handle</i>	eDMA handle pointer.
---------------	----------------------

### 12.7.36 void EDMA\_StopTransfer ( *edma\_handle\_t \* handle* )

This function disables the channel request to pause the transfer. Users can call [EDMA\\_StartTransfer\(\)](#) again to resume the transfer.

Parameters

<i>handle</i>	eDMA handle pointer.
---------------	----------------------

### 12.7.37 void EDMA\_AbortTransfer ( *edma\_handle\_t \* handle* )

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

<i>handle</i>	DMA handle pointer.
---------------	---------------------

### 12.7.38 void EDMA\_HandleIRQ ( *edma\_handle\_t \* handle* )

This function clears the channel major interrupt flag and calls the callback function if it is not NULL.

Note: For the case using TCD queue, when the major iteration count is exhausted, additional operations are performed. These include the final address adjustments and reloading of the BITER field into the CITER. Assertion of an optional interrupt request also occurs at this time, as does a possible fetch of a new TCD from memory using the scatter/gather address pointer included in the descriptor (if scatter/gather is enabled).

## Function Documentation

For instance, when the time interrupt of TCD[0] happens, the TCD[1] has already been loaded into the eDMA engine. As sga and sga\_index are calculated based on the DLAST\_SGA bitfield lies in the TCD\_CSR register, the sga\_index in this case should be 2 (DLAST\_SGA of TCD[1] stores the address of TCD[2]). Thus, the "tcdUsed" updated should be (tcdUsed - 2U) which indicates the number of TCDs can be loaded in the memory pool (because TCD[0] and TCD[1] have been loaded into the eDMA engine at this point already.).

For the last two continuous ISRs in a scatter/gather process, they both load the last TCD (The last ISR does not load a new TCD) from the memory pool to the eDMA engine when major loop completes. Therefore, ensure that the header and tcdUsed updated are identical for them. tcdUsed are both 0 in this case as no TCD to be loaded.

See the "eDMA basic data flow" in the eDMA Functional description part of the Reference Manual for further details.

### Parameters

<i>handle</i>	eDMA handle pointer.
---------------	----------------------

# Chapter 13

## EWM: External Watchdog Monitor Driver

### 13.1 Overview

The MCUXpresso SDK provides a peripheral driver for the module of MCUXpresso SDK devices.

### 13.2 Typical use case

```
ewm_config_t config;
EWM_GetDefaultConfig(&config);
config.enableInterrupt = true;
config.compareLowValue = 0U;
config.compareHighValue = 0xAAU;
NVIC_EnableIRQ(WDOG_EWM_IRQn);
EWM_Init(base, &config);
```

## Data Structures

- struct `ewm_config_t`  
*Describes EWM clock source.* [More...](#)

## Enumerations

- enum `_ewm_interrupt_enable_t` { `kEWM_InterruptEnable` = EWM\_CTRL\_INTEN\_MASK }  
*EWM interrupt configuration structure with default settings all disabled.*
- enum `_ewm_status_flags_t` { `kEWM_RunningFlag` = EWM\_CTRL\_EWMEN\_MASK }  
*EWM status flags.*

## Driver version

- #define `FSL_EWM_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 1))  
*EWM driver version 2.0.1.*

## EWM initialization and de-initialization

- void `EWM_Init` (EWM\_Type \*base, const `ewm_config_t` \*config)  
*Initializes the EWM peripheral.*
- void `EWM_Deinit` (EWM\_Type \*base)  
*Deinitializes the EWM peripheral.*
- void `EWM_GetDefaultConfig` (`ewm_config_t` \*config)  
*Initializes the EWM configuration structure.*

## EWM functional Operation

- static void `EWM_EnableInterrupts` (EWM\_Type \*base, uint32\_t mask)  
*Enables the EWM interrupt.*
- static void `EWM_DisableInterrupts` (EWM\_Type \*base, uint32\_t mask)

## Enumeration Type Documentation

- static uint32\_t [EWM\\_GetStatusFlags](#) (EWM\_Type \*base)  
*Disables the EWM interrupt.*
- void [EWM\\_Refresh](#) (EWM\_Type \*base)  
*Gets all status flags.*
- Services the EWM.

## 13.3 Data Structure Documentation

### 13.3.1 struct ewm\_config\_t

Data structure for EWM configuration.

This structure is used to configure the EWM.

#### Data Fields

- bool [enableEwm](#)  
*Enable EWM module.*
- bool [enableEwmInput](#)  
*Enable EWM\_in input.*
- bool [setInputAssertLogic](#)  
*EWM\_in signal assertion state.*
- bool [enableInterrupt](#)  
*Enable EWM interrupt.*
- uint8\_t [compareLowValue](#)  
*Compare low-register value.*
- uint8\_t [compareHighValue](#)  
*Compare high-register value.*

## 13.4 Macro Definition Documentation

### 13.4.1 #define FSL\_EWM\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))

## 13.5 Enumeration Type Documentation

### 13.5.1 enum \_ewm\_interrupt\_enable\_t

This structure contains the settings for all of EWM interrupt configurations.

Enumerator

**kEWM\_InterruptEnable** Enable the EWM to generate an interrupt.

### 13.5.2 enum \_ewm\_status\_flags\_t

This structure contains the constants for the EWM status flags for use in the EWM functions.

Enumerator

***kEWM\_RunningFlag*** Running flag, set when EWM is enabled.

## 13.6 Function Documentation

### 13.6.1 void EWM\_Init ( EWM\_Type \* *base*, const ewm\_config\_t \* *config* )

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration. Note that, except for the interrupt enable control bit, other control bits and registers are write once after a CPU reset. Modifying them more than once generates a bus transfer error.

This is an example.

```
*     ewm_config_t config;
*     EWM_GetDefaultConfig(&config);
*     config.compareHighValue = 0xAAU;
*     EWM_Init(ewm_base, &config);
*
```

Parameters

<i>base</i>	EWM peripheral base address
<i>config</i>	The configuration of the EWM

### 13.6.2 void EWM\_Deinit ( EWM\_Type \* *base* )

This function is used to shut down the EWM.

Parameters

<i>base</i>	EWM peripheral base address
-------------	-----------------------------

### 13.6.3 void EWM\_GetDefaultConfig ( ewm\_config\_t \* *config* )

This function initializes the EWM configuration structure to default values. The default values are as follows.

```
*     ewmConfig->enableEwm = true;
*     ewmConfig->enableEwmInput = false;
*     ewmConfig->setInputAssertLogic = false;
*     ewmConfig->enableInterrupt = false;
*     ewmConfig->ewm_lpo_clock_source_t = kEWM_LpoClockSource0;
*     ewmConfig->prescaler = 0;
*     ewmConfig->compareLowValue = 0;
*     ewmConfig->compareHighValue = 0xFEU;
*
```

## Function Documentation

Parameters

<i>config</i>	Pointer to the EWM configuration structure.
---------------	---

See Also

[ewm\\_config\\_t](#)

### 13.6.4 static void EWM\_EnableInterrupts ( **EWM\_Type** \* *base*, **uint32\_t** *mask* ) [**inline**], [**static**]

This function enables the EWM interrupt.

Parameters

<i>base</i>	EWM peripheral base address
<i>mask</i>	The interrupts to enable The parameter can be combination of the following source if defined <ul style="list-style-type: none"><li>• kEWM InterruptEnable</li></ul>

### 13.6.5 static void EWM\_DisableInterrupts ( **EWM\_Type** \* *base*, **uint32\_t** *mask* ) [**inline**], [**static**]

This function enables the EWM interrupt.

Parameters

<i>base</i>	EWM peripheral base address
<i>mask</i>	The interrupts to disable The parameter can be combination of the following source if defined <ul style="list-style-type: none"><li>• kEWM InterruptEnable</li></ul>

### 13.6.6 static **uint32\_t** EWM\_GetStatusFlags ( **EWM\_Type** \* *base* ) [**inline**], [**static**]

This function gets all status flags.

This is an example for getting the running flag.

```
*     uint32_t status;
*     status = EWM_GetStatusFlags(ewm_base) & kEWM_RunningFlag;
*
```

## Parameters

<i>base</i>	EWM peripheral base address
-------------	-----------------------------

## Returns

State of the status flag: asserted (true) or not-asserted (false).

## See Also

[\\_ewm\\_status\\_flags\\_t](#)

- True: a related status flag has been set.
- False: a related status flag is not set.

### 13.6.7 void EWM\_Refresh ( EWM\_Type \* *base* )

This function resets the EWM counter to zero.

## Parameters

<i>base</i>	EWM peripheral base address
-------------	-----------------------------

## Function Documentation

# Chapter 14

## C90TFS Flash Driver

### 14.1 Overview

The flash provides the C90TFS Flash driver of MCUXpresso SDK devices with the C90TFS Flash module inside. The flash driver provides general APIs to handle specific operations on C90TFS/FTFx Flash module. The user can use those APIs directly in the application. In addition, it provides internal functions called by the driver. Although these functions are not meant to be called from the user's application directly, the APIs can still be used.

### Data Structures

- struct [flash\\_execute\\_in\\_ram\\_function\\_config\\_t](#)  
*Flash execute-in-RAM function information. [More...](#)*
- struct [flash\\_swap\\_state\\_config\\_t](#)  
*Flash Swap information. [More...](#)*
- struct [flash\\_swap\\_ifr\\_field\\_config\\_t](#)  
*Flash Swap IFR fields. [More...](#)*
- union [flash\\_swap\\_ifr\\_field\\_data\\_t](#)  
*Flash Swap IFR field data. [More...](#)*
- union [pflash\\_protection\\_status\\_low\\_t](#)  
*PFlash protection status - low 32bit. [More...](#)*
- struct [pflash\\_protection\\_status\\_t](#)  
*PFlash protection status - full. [More...](#)*
- struct [flash\\_prefetch\\_speculation\\_status\\_t](#)  
*Flash prefetch speculation status. [More...](#)*
- struct [flash\\_protection\\_config\\_t](#)  
*Active flash protection information for the current operation. [More...](#)*
- struct [flash\\_access\\_config\\_t](#)  
*Active flash Execute-Only access information for the current operation. [More...](#)*
- struct [flash\\_operation\\_config\\_t](#)  
*Active flash information for the current operation. [More...](#)*
- struct [flash\\_config\\_t](#)  
*Flash driver state information. [More...](#)*

### Typedefs

- [typedef void\(\\* flash\\_callback\\_t \)\(void\)](#)  
*A callback type used for the Pflash block.*

### Enumerations

- enum [flash\\_margin\\_value\\_t](#) {  
  kFLASH\_MarginValueNormal,  
  kFLASH\_MarginValueUser,  
  kFLASH\_MarginValueFactory,

## Overview

- `kFLASH_MarginValueInvalid }`  
*Enumeration for supported flash margin levels.*
- enum `flash_security_state_t` {  
  `kFLASH_SecurityStateNotSecure,`  
  `kFLASH_SecurityStateBackdoorEnabled,`  
  `kFLASH_SecurityStateBackdoorDisabled }`  
*Enumeration for the three possible flash security states.*
- enum `flash_protection_state_t` {  
  `kFLASH_ProtectionStateUnprotected,`  
  `kFLASH_ProtectionStateProtected,`  
  `kFLASH_ProtectionStateMixed }`  
*Enumeration for the three possible flash protection levels.*
- enum `flash_execute_only_access_state_t` {  
  `kFLASH_AccessStateUnLimited,`  
  `kFLASH_AccessStateExecuteOnly,`  
  `kFLASH_AccessStateMixed }`  
*Enumeration for the three possible flash execute access levels.*
- enum `flash_property_tag_t` {  
  `kFLASH_PropertyPflashSectorSize = 0x00U,`  
  `kFLASH_PropertyPflashTotalSize = 0x01U,`  
  `kFLASH_PropertyPflashBlockSize = 0x02U,`  
  `kFLASH_PropertyPflashBlockCount = 0x03U,`  
  `kFLASH_PropertyPflashBlockBaseAddr = 0x04U,`  
  `kFLASH_PropertyPflashFacSupport = 0x05U,`  
  `kFLASH_PropertyPflashAccessSegmentSize = 0x06U,`  
  `kFLASH_PropertyPflashAccessSegmentCount = 0x07U,`  
  `kFLASH_PropertyFlexRamBlockBaseAddr = 0x08U,`  
  `kFLASH_PropertyFlexRamTotalSize = 0x09U,`  
  `kFLASH_PropertyDflashSectorSize = 0x10U,`  
  `kFLASH_PropertyDflashTotalSize = 0x11U,`  
  `kFLASH_PropertyDflashBlockSize = 0x12U,`  
  `kFLASH_PropertyDflashBlockCount = 0x13U,`  
  `kFLASH_PropertyDflashBlockBaseAddr = 0x14U,`  
  `kFLASH_PropertyEepromTotalSize = 0x15U,`  
  `kFLASH_PropertyFlashMemoryIndex = 0x20U,`  
  `kFLASH_PropertyFlashCacheControllerIndex = 0x21U }`  
*Enumeration for various flash properties.*
- enum `_flash_execute_in_ram_function_constants` {  
  `kFLASH_ExecuteInRamFunctionMaxSizeInWords = 16U,`  
  `kFLASH_ExecuteInRamFunctionTotalNum = 2U }`  
*Constants for execute-in-RAM flash function.*
- enum `flash_read_resource_option_t` {  
  `kFLASH_ResourceOptionFlashIfr,`  
  `kFLASH_ResourceOptionVersionId = 0x01U }`  
*Enumeration for the two possible options of flash read resource command.*
- enum `_flash_read_resource_range` {

```

kFLASH_ResourceRangePflashIfrSizeInBytes = 1024U,
kFLASH_ResourceRangeVersionIdSizeInBytes = 8U,
kFLASH_ResourceRangeVersionIdStart = 0x08U,
kFLASH_ResourceRangeVersionIdEnd = 0x0FU,
kFLASH_ResourceRangePflashSwapIfrStart = 0x40000U,
kFLASH_ResourceRangePflashSwapIfrEnd,
kFLASH_ResourceRangeDflashIfrStart = 0x800000U,
kFLASH_ResourceRangeDflashIfrEnd = 0x8003FFU }

```

*Enumeration for the range of special-purpose flash resource.*

- enum `_k3_flash_read_once_index` {
   
kFLASH\_RecordIndexSwapAddr = 0xA1U,
   
kFLASH\_RecordIndexSwapEnable = 0xA2U,
   
kFLASH\_RecordIndexSwapDisable = 0xA3U }

*Enumeration for the index of read/program once record.*

- enum `flash_flexram_function_option_t` {
   
kFLASH\_FlexramFunctionOptionAvailableAsRam = 0xFFU,
   
kFLASH\_FlexramFunctionOptionAvailableForEeprom = 0x00U }

*Enumeration for the two possible options of set FlexRAM function command.*

- enum `_flash_acceleration_ram_property`
  
*Enumeration for acceleration RAM property.*
- enum `flash_swap_function_option_t` {
   
kFLASH\_SwapFunctionOptionEnable = 0x00U,
   
kFLASH\_SwapFunctionOptionDisable = 0x01U }

*Enumeration for the possible options of Swap function.*

- enum `flash_swap_control_option_t` {
   
kFLASH\_SwapControlOptionInitializeSystem = 0x01U,
   
kFLASH\_SwapControlOptionSetInUpdateState = 0x02U,
   
kFLASH\_SwapControlOptionSetInCompleteState = 0x04U,
   
kFLASH\_SwapControlOptionReportStatus = 0x08U,
   
kFLASH\_SwapControlOptionDisableSystem = 0x10U }

*Enumeration for the possible options of Swap control commands.*

- enum `flash_swap_state_t` {
   
kFLASH\_SwapStateUninitialized = 0x00U,
   
kFLASH\_SwapStateReady = 0x01U,
   
kFLASH\_SwapStateUpdate = 0x02U,
   
kFLASH\_SwapStateUpdateErased = 0x03U,
   
kFLASH\_SwapStateComplete = 0x04U,
   
kFLASH\_SwapStateDisabled = 0x05U }

*Enumeration for the possible flash Swap status.*

- enum `flash_swap_block_status_t` {
   
kFLASH\_SwapBlockStatusLowerHalfProgramBlocksAtZero,
   
kFLASH\_SwapBlockStatusUpperHalfProgramBlocksAtZero }

*Enumeration for the possible flash Swap block status*

- enum `flash_partition_flexram_load_option_t` {
   
kFLASH\_PartitionFlexramLoadOptionLoadedWithValidEepromData,
   
kFLASH\_PartitionFlexramLoadOptionNotLoaded = 0x01U }

*Enumeration for the FlexRAM load during reset option.*

## Overview

- enum `flash_memory_index_t` {  
    `kFLASH_MemoryIndexPrimaryFlash` = 0x00U,  
    `kFLASH_MemoryIndexSecondaryFlash` = 0x01U }  
    *Enumeration for the flash memory index.*
- enum `flash_cache_controller_index_t` {  
    `kFLASH_CacheControllerIndexForCore0` = 0x00U,  
    `kFLASH_CacheControllerIndexForCore1` = 0x01U }  
    *Enumeration for the flash cache controller index.*
- enum `flash_prefetch_speculation_option_t`  
    *Enumeration for the two possible options of flash prefetch speculation.*
- enum `flash_cache_clear_process_t` {  
    `kFLASH_CacheClearProcessPre` = 0x00U,  
    `kFLASH_CacheClearProcessPost` = 0x01U }  
    *Flash cache clear process code.*

## Flash version

- enum `_flash_driver_version_constants` {  
    `kFLASH_DriverVersionName` = 'F',  
    `kFLASH_DriverVersionMajor` = 2,  
    `kFLASH_DriverVersionMinor` = 3,  
    `kFLASH_DriverVersionBugfix` = 1 }  
    *Flash driver version for ROM.*
- #define `MAKE_VERSION`(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))  
    *Constructs the version number for drivers.*
- #define `FSL_FLASH_DRIVER_VERSION` (`MAKE_VERSION`(2, 3, 1))  
    *Flash driver version for SDK.*

## Flash configuration

- #define `FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT` 1  
    *Indicates whether to support FlexNVM in the Flash driver.*
- #define `FLASH_SSD_IS_FLEXNVM_ENABLED` (`FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT` && `FSL_FEATURE_FLASH_HAS_FLEX_NVM`)  
    *Indicates whether the FlexNVM is enabled in the Flash driver.*
- #define `FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT` 1  
    *Indicates whether to support Secondary flash in the Flash driver.*
- #define `FLASH_SSD_IS_SECONDARY_FLASH_ENABLED` (0)  
    *Indicates whether the secondary flash is supported in the Flash driver.*
- #define `FLASH_DRIVER_IS_FLASH_RESIDENT` 1  
    *Flash driver location.*
- #define `FLASH_DRIVER_IS_EXPORTED` 0  
    *Flash Driver Export option.*

## Flash status

- enum `_flash_status` {
   
`kStatus_FLASH_Success` = MAKE\_STATUS(kStatusGroupGeneric, 0),
   
`kStatus_FLASH_InvalidArgument` = MAKE\_STATUS(kStatusGroupGeneric, 4),
   
`kStatus_FLASH_SizeError` = MAKE\_STATUS(kStatusGroupFlashDriver, 0),
   
`kStatus_FLASH_AlignmentError`,
   
`kStatus_FLASH_AddressError` = MAKE\_STATUS(kStatusGroupFlashDriver, 2),
   
`kStatus_FLASH_AccessError`,
   
`kStatus_FLASH_ProtectionViolation`,
   
`kStatus_FLASH_CommandFailure`,
   
`kStatus_FLASH_UnknownProperty` = MAKE\_STATUS(kStatusGroupFlashDriver, 6),
   
`kStatus_FLASH_EraseKeyError` = MAKE\_STATUS(kStatusGroupFlashDriver, 7),
   
`kStatus_FLASH_RegionExecuteOnly`,
   
`kStatus_FLASH_ExecuteInRamFunctionNotReady`,
   
`kStatus_FLASH_PartitionStatusUpdateFailure`,
   
`kStatus_FLASH_SetFlexramAsEepromError`,
   
`kStatus_FLASH_RecoverFlexramAsRamError`,
   
`kStatus_FLASH_SetFlexramAsRamError` = MAKE\_STATUS(kStatusGroupFlashDriver, 13),
   
`kStatus_FLASH_RecoverFlexramAsEepromError`,
   
`kStatus_FLASH_CommandNotSupported` = MAKE\_STATUS(kStatusGroupFlashDriver, 15),
   
`kStatus_FLASH_SwapSystemNotInUninitialized`,
   
`kStatus_FLASH_SwapIndicatorAddressError`,
   
`kStatus_FLASH_ReadOnlyProperty` = MAKE\_STATUS(kStatusGroupFlashDriver, 18),
   
`kStatus_FLASH_InvalidPropertyValue`,
   
`kStatus_FLASH_InvalidSpeculationOption` }
   
*Flash driver status codes.*
- #define `kStatusGroupGeneric` 0
   
*Flash driver status group.*
- #define `kStatusGroupFlashDriver` 1
   
• #define `MAKE_STATUS`(group, code) (((group)\*100) + (code))
   
*Constructs a status code value from a group and a code number.*

## Flash API key

- enum `_flash_driver_api_keys` { `kFLASH_ApiEraseKey` = FOUR\_CHAR\_CODE('k', 'f', 'e', 'k') }
   
*Enumeration for Flash driver API keys.*
- #define `FOUR_CHAR_CODE`(a, b, c, d) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))
   
*Constructs the four character code for the Flash driver API key.*

## Initialization

- `status_t FLASH_Init (flash_config_t *config)`
  
*Initializes the global flash properties structure members.*
- `status_t FLASH_SetCallback (flash_config_t *config, flash_callback_t callback)`
  
*Sets the desired flash callback function.*
- `status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t *config)`
  
*Prepares flash execute-in-RAM functions.*

## Overview

## Erasing

- status\_t **FLASH\_EraseAll** (**flash\_config\_t** \*config, uint32\_t key)  
*Erases entire flash.*
- status\_t **FLASH\_Erase** (**flash\_config\_t** \*config, uint32\_t start, uint32\_t lengthInBytes, uint32\_t key)  
*Erases the flash sectors encompassed by parameters passed into function.*
- status\_t **FLASH\_EraseAllExecuteOnlySegments** (**flash\_config\_t** \*config, uint32\_t key)  
*Erases the entire flash, including protected sectors.*

## Programming

- status\_t **FLASH\_Program** (**flash\_config\_t** \*config, uint32\_t start, uint32\_t \*src, uint32\_t lengthInBytes)  
*Programs flash with data at locations passed in through parameters.*
- status\_t **FLASH\_ProgramOnce** (**flash\_config\_t** \*config, uint32\_t index, uint32\_t \*src, uint32\_t lengthInBytes)  
*Programs Program Once Field through parameters.*
- status\_t **FLASH\_ProgramSection** (**flash\_config\_t** \*config, uint32\_t start, uint32\_t \*src, uint32\_t lengthInBytes)  
*Programs flash with data at locations passed in through parameters via the Program Section command.*
- status\_t **FLASH\_EepromWrite** (**flash\_config\_t** \*config, uint32\_t start, uint8\_t \*src, uint32\_t lengthInBytes)  
*Programs the EEPROM with data at locations passed in through parameters.*

## Reading

- status\_t **FLASH\_ReadResource** (**flash\_config\_t** \*config, uint32\_t start, uint32\_t \*dst, uint32\_t lengthInBytes, **flash\_read\_resource\_option\_t** option)  
*Reads the resource with data at locations passed in through parameters.*
- status\_t **FLASH\_ReadOnce** (**flash\_config\_t** \*config, uint32\_t index, uint32\_t \*dst, uint32\_t lengthInBytes)  
*Reads the Program Once Field through parameters.*

## Security

- status\_t **FLASH\_GetSecurityState** (**flash\_config\_t** \*config, **flash\_security\_state\_t** \*state)  
*Returns the security state via the pointer passed into the function.*
- status\_t **FLASH\_SecurityBypass** (**flash\_config\_t** \*config, const uint8\_t \*backdoorKey)  
*Allows users to bypass security with a backdoor key.*

## Verification

- status\_t **FLASH\_VerifyEraseAll** (**flash\_config\_t** \*config, **flash\_margin\_value\_t** margin)  
*Verifies erasure of the entire flash at a specified margin level.*
- status\_t **FLASH\_VerifyErase** (**flash\_config\_t** \*config, uint32\_t start, uint32\_t lengthInBytes, **flash\_margin\_value\_t** margin)  
*Verifies an erasure of the desired flash area at a specified margin level.*
- status\_t **FLASH\_VerifyProgram** (**flash\_config\_t** \*config, uint32\_t start, uint32\_t lengthInBytes, const uint32\_t \*expectedData, **flash\_margin\_value\_t** margin, uint32\_t \*failedAddress, uint32\_t \*failedData)  
*Verifies a programmed flash area against expected data.*

- Verifies programming of the desired flash area at a specified margin level.
- status\_t `FLASH_VerifyEraseAllExecuteOnlySegments` (`flash_config_t` \*config, `flash_margin_value_t` margin)
 

*Verifies whether the program flash execute-only segments have been erased to the specified read margin level.*

## Protection

- status\_t `FLASH_IsProtected` (`flash_config_t` \*config, `uint32_t` start, `uint32_t` lengthInBytes, `flash_protection_state_t` \*protection\_state)
 

*Returns the protection state of the desired flash area via the pointer passed into the function.*
- status\_t `FLASH_IsExecuteOnly` (`flash_config_t` \*config, `uint32_t` start, `uint32_t` lengthInBytes, `flash_execute_only_access_state_t` \*access\_state)
 

*Returns the access state of the desired flash area via the pointer passed into the function.*

## Properties

- status\_t `FLASHGetProperty` (`flash_config_t` \*config, `flash_property_tag_t` whichProperty, `uint32_t` \*value)
 

*Returns the desired flash property.*
- status\_t `FLASH_SetProperty` (`flash_config_t` \*config, `flash_property_tag_t` whichProperty, `uint32_t` value)
 

*Sets the desired flash property.*

## FlexRAM

- status\_t `FLASH_SetFlexramFunction` (`flash_config_t` \*config, `flash flexram_function_option_t` option)
 

*Sets the FlexRAM function command.*

## FlexNVM

Configures the Swap function or checks the swap state of the Flash module.

Parameters

<code>config</code>	A pointer to the storage for the driver runtime state.
<code>address</code>	Address used to configure the flash Swap function.
<code>option</code>	The possible option used to configure Flash Swap function or check the flash Swap status

## Overview

<i>returnInfo</i>	A pointer to the data which is used to return the information of flash Swap.
-------------------	--

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_SwapIndicatorAddressError</i>	Swap indicator address is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

Swaps the lower half flash with the higher half flash.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>address</i>	Address used to configure the flash swap function
<i>option</i>	The possible option used to configure the Flash Swap function or check the flash Swap status.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.

<code>kStatus_FLASH_SwapIndicatorAddressError</code>	Swap indicator address is invalid.
<code>kStatus_FLASH_ExecuteInRamFunctionNotReady</code>	Execute-in-RAM function is not available.
<code>kStatus_FLASH_AccessError</code>	Invalid instruction codes and out-of bounds addresses.
<code>kStatus_FLASH_ProtectionViolation</code>	The program/erase operation is requested to execute on protected areas.
<code>kStatus_FLASH_CommandFailure</code>	Run-time error during command execution.
<code>kStatus_FLASH_SwapSystemNotInUninitialized</code>	Swap system is not in an uninitialized state.

- status\_t `FLASH_ProgramPartition` (`flash_config_t` \*config, `flash_partition_flexram_load_option_t` option, `uint32_t` eepromDataSizeCode, `uint32_t` flexnvmPartitionCode)
   
*Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.*

## Flash Protection Utilities

- status\_t `FLASH_PflashSetProtection` (`flash_config_t` \*config, `pflash_protection_status_t` \*protectStatus)
   
*Sets the PFlash Protection to the intended protection status.*
- status\_t `FLASH_PflashGetProtection` (`flash_config_t` \*config, `pflash_protection_status_t` \*protectStatus)
   
*Gets the PFlash protection status.*
- status\_t `FLASH_DflashSetProtection` (`flash_config_t` \*config, `uint8_t` protectStatus)
   
*Sets the DFlash protection to the intended protection status.*
- status\_t `FLASH_DflashGetProtection` (`flash_config_t` \*config, `uint8_t` \*protectStatus)
   
*Gets the DFlash protection status.*
- status\_t `FLASH_EepromSetProtection` (`flash_config_t` \*config, `uint8_t` protectStatus)
   
*Sets the EEPROM protection to the intended protection status.*
- status\_t `FLASH_EepromGetProtection` (`flash_config_t` \*config, `uint8_t` \*protectStatus)
   
*Gets the DFlash protection status.*

## 14.2 Data Structure Documentation

### 14.2.1 struct `flash_execute_in_ram_function_config_t`

#### Data Fields

- `uint32_t activeFunctionCount`
  
*Number of available execute-in-RAM functions.*
- `uint32_t * flashRunCommand`
  
*Execute-in-RAM function: flash\_run\_command.*
- `uint32_t * flashCommonBitOperation`

## Data Structure Documentation

*Execute-in-RAM function: flash\_common\_bit\_operation.*

### 14.2.1.0.0.29 Field Documentation

14.2.1.0.0.29.1 `uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount`

14.2.1.0.0.29.2 `uint32_t* flash_execute_in_ram_function_config_t::flashRunCommand`

14.2.1.0.0.29.3 `uint32_t* flash_execute_in_ram_function_config_t::flashCommonBitOperation`

## 14.2.2 `struct flash_swap_state_config_t`

### Data Fields

- `flash_swap_state_t flashSwapState`  
*The current Swap system status.*
- `flash_swap_block_status_t currentSwapBlockStatus`  
*The current Swap block status.*
- `flash_swap_block_status_t nextSwapBlockStatus`  
*The next Swap block status.*

### 14.2.2.0.0.30 Field Documentation

14.2.2.0.0.30.1 `flash_swap_state_t flash_swap_state_config_t::flashSwapState`

14.2.2.0.0.30.2 `flash_swap_block_status_t flash_swap_state_config_t::currentSwapBlockStatus`

14.2.2.0.0.30.3 `flash_swap_block_status_t flash_swap_state_config_t::nextSwapBlockStatus`

## 14.2.3 `struct flash_swap_ifr_field_config_t`

### Data Fields

- `uint16_t swapIndicatorAddress`  
*A Swap indicator address field.*
- `uint16_t swapEnableWord`  
*A Swap enable word field.*
- `uint8_t reserved0 [4]`  
*A reserved field.*
- `uint8_t reserved1 [2]`  
*A reserved field.*
- `uint16_t swapDisableWord`  
*A Swap disable word field.*
- `uint8_t reserved2 [4]`  
*A reserved field.*

**14.2.3.0.0.31 Field Documentation****14.2.3.0.0.31.1 uint16\_t flash\_swap\_ifr\_field\_config\_t::swapIndicatorAddress****14.2.3.0.0.31.2 uint16\_t flash\_swap\_ifr\_field\_config\_t::swapEnableWord****14.2.3.0.0.31.3 uint8\_t flash\_swap\_ifr\_field\_config\_t::reserved0[4]****14.2.3.0.0.31.4 uint8\_t flash\_swap\_ifr\_field\_config\_t::reserved1[2]****14.2.3.0.0.31.5 uint16\_t flash\_swap\_ifr\_field\_config\_t::swapDisableWord****14.2.3.0.0.31.6 uint8\_t flash\_swap\_ifr\_field\_config\_t::reserved2[4]****14.2.4 union flash\_swap\_ifr\_field\_data\_t****Data Fields**

- **uint32\_t flashSwapIfrData [2]**  
*A flash Swap IFR field data .*
- **flash\_swap\_ifr\_field\_config\_t flashSwapIfrField**  
*A flash Swap IFR field structure.*

**14.2.4.0.0.32 Field Documentation****14.2.4.0.0.32.1 uint32\_t flash\_swap\_ifr\_field\_data\_t::flashSwapIfrData[2]****14.2.4.0.0.32.2 flash\_swap\_ifr\_field\_config\_t flash\_swap\_ifr\_field\_data\_t::flashSwapIfrField****14.2.5 union pflash\_protection\_status\_low\_t****Data Fields**

- **uint32\_t protl32b**  
*PROT[31:0] .*
- **uint8\_t protsl**  
*PROTS[7:0] .*
- **uint8\_t protsh**  
*PROTS[15:8] .*

## Data Structure Documentation

### 14.2.5.0.0.33 Field Documentation

14.2.5.0.0.33.1 `uint32_t pflash_protection_status_low_t::protl32b`

14.2.5.0.0.33.2 `uint8_t pflash_protection_status_low_t::protsl`

14.2.5.0.0.33.3 `uint8_t pflash_protection_status_low_t::protsh`

## 14.2.6 `struct pflash_protection_status_t`

### Data Fields

- `pflash_protection_status_low_t valueLow32b`  
*PROT[31:0] or PROTS[15:0].*

### 14.2.6.0.0.34 Field Documentation

14.2.6.0.0.34.1 `pflash_protection_status_low_t pflash_protection_status_t::valueLow32b`

## 14.2.7 `struct flash_prefetch_speculation_status_t`

### Data Fields

- `flash_prefetch_speculation_option_t instructionOption`  
*Instruction speculation.*
- `flash_prefetch_speculation_option_t dataOption`  
*Data speculation.*

### 14.2.7.0.0.35 Field Documentation

14.2.7.0.0.35.1 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t-::instructionOption`

14.2.7.0.0.35.2 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::data-Option`

## 14.2.8 `struct flash_protection_config_t`

### Data Fields

- `uint32_t regionBase`  
*Base address of flash protection region.*
- `uint32_t regionSize`  
*size of flash protection region.*
- `uint32_t regionCount`  
*flash protection region count.*

**14.2.8.0.0.36 Field Documentation****14.2.8.0.0.36.1 uint32\_t flash\_protection\_config\_t::regionBase****14.2.8.0.0.36.2 uint32\_t flash\_protection\_config\_t::regionSize****14.2.8.0.0.36.3 uint32\_t flash\_protection\_config\_t::regionCount****14.2.9 struct flash\_access\_config\_t****Data Fields**

- **uint32\_t SegmentBase**  
*Base address of flash Execute-Only segment.*
- **uint32\_t SegmentSize**  
*size of flash Execute-Only segment.*
- **uint32\_t SegmentCount**  
*flash Execute-Only segment count.*

**14.2.9.0.0.37 Field Documentation****14.2.9.0.0.37.1 uint32\_t flash\_access\_config\_t::SegmentBase****14.2.9.0.0.37.2 uint32\_t flash\_access\_config\_t::SegmentSize****14.2.9.0.0.37.3 uint32\_t flash\_access\_config\_t::SegmentCount****14.2.10 struct flash\_operation\_config\_t****Data Fields**

- **uint32\_t convertedAddress**  
*A converted address for the current flash type.*
- **uint32\_t activeSectorSize**  
*A sector size of the current flash type.*
- **uint32\_t activeBlockSize**  
*A block size of the current flash type.*
- **uint32\_t blockWriteUnitSize**  
*The write unit size.*
- **uint32\_t sectorCmdAddressAlignment**  
*An erase sector command address alignment.*
- **uint32\_t partCmdAddressAlignment**  
*A program/verify part command address alignment.*
- **32\_t resourceCmdAddressAlignment**  
*A read resource command address alignment.*
- **uint32\_t checkCmdAddressAlignment**  
*A program check command address alignment.*

## Data Structure Documentation

### 14.2.10.0.0.38 Field Documentation

14.2.10.0.0.38.1 `uint32_t flash_operation_config_t::convertedAddress`

14.2.10.0.0.38.2 `uint32_t flash_operation_config_t::activeSectorSize`

14.2.10.0.0.38.3 `uint32_t flash_operation_config_t::activeBlockSize`

14.2.10.0.0.38.4 `uint32_t flash_operation_config_t::blockWriteUnitSize`

14.2.10.0.0.38.5 `uint32_t flash_operation_config_t::sectorCmdAddressAlignment`

14.2.10.0.0.38.6 `uint32_t flash_operation_config_t::partCmdAddressAlignment`

14.2.10.0.0.38.7 `uint32_t flash_operation_config_t::resourceCmdAddressAlignment`

14.2.10.0.0.38.8 `uint32_t flash_operation_config_t::checkCmdAddressAlignment`

### 14.2.11 `struct flash_config_t`

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

## Data Fields

- `uint32_t PFlashBlockBase`  
*A base address of the first PFlash block.*
- `uint32_t PFlashTotalSize`  
*The size of the combined PFlash block.*
- `uint8_t PFlashBlockCount`  
*A number of PFlash blocks.*
- `uint8_t FlashMemoryIndex`  
*0 - primary flash; 1 - secondary flash*
- `uint8_t FlashCacheControllerIndex`  
*0 - Controller for core 0; 1 - Controller for core 1*
- `uint8_t Reserved0`  
*Reserved field 0.*
- `uint32_t PFlashSectorSize`  
*The size in bytes of a sector of PFlash.*
- `flash_callback_t PFlashCallback`  
*The callback function for the flash API.*
- `uint32_t PFlashAccessSegmentSize`  
*A size in bytes of an access segment of PFlash.*
- `uint32_t PFlashAccessSegmentCount`  
*A number of PFlash access segments.*
- `uint32_t * flashExecuteInRamFunctionInfo`  
*An information structure of the flash execute-in-RAM function.*
- `uint32_t FlexRAMBlockBase`  
*For the FlexNVM device, this is the base address of the FlexRAM.*

- **uint32\_t FlexRAMTotalSize**  
*For the FlexNVM device, this is the size of the FlexRAM.*
- **uint32\_t DFlashBlockBase**  
*For the FlexNVM device, this is the base address of the D-Flash memory (FlexNVM memory)*
- **uint32\_t DFlashTotalSize**  
*For the FlexNVM device, this is the total size of the FlexNVM memory;*
- **uint32\_t EEepromTotalSize**  
*For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM.*

#### 14.2.11.0.0.39 Field Documentation

**14.2.11.0.0.39.1 uint32\_t flash\_config\_t::PFlashTotalSize**

**14.2.11.0.0.39.2 uint8\_t flash\_config\_t::PFlashBlockCount**

**14.2.11.0.0.39.3 uint32\_t flash\_config\_t::PFlashSectorSize**

**14.2.11.0.0.39.4 flash\_callback\_t flash\_config\_t::PFlashCallback**

**14.2.11.0.0.39.5 uint32\_t flash\_config\_t::PFlashAccessSegmentSize**

**14.2.11.0.0.39.6 uint32\_t flash\_config\_t::PFlashAccessSegmentCount**

**14.2.11.0.0.39.7 uint32\_t\* flash\_config\_t::flashExecuteInRamFunctionInfo**

**14.2.11.0.0.39.8 uint32\_t flash\_config\_t::FlexRAMBlockBase**

For the non-FlexNVM device, this is the base address of the acceleration RAM memory

**14.2.11.0.0.39.9 uint32\_t flash\_config\_t::FlexRAMTotalSize**

For the non-FlexNVM device, this is the size of the acceleration RAM memory

**14.2.11.0.0.39.10 uint32\_t flash\_config\_t::DFlashBlockBase**

For the non-FlexNVM device, this field is unused

**14.2.11.0.0.39.11 uint32\_t flash\_config\_t::DFlashTotalSize**

For the non-FlexNVM device, this field is unused

**14.2.11.0.0.39.12 uint32\_t flash\_config\_t::EEepromTotalSize**

For the non-FlexNVM device, this field is unused

## Enumeration Type Documentation

### 14.3 Macro Definition Documentation

14.3.1 `#define MAKE_VERSION( major, minor, bugfix ) (((major) << 16) | ((minor) << 8) | (bugfix))`

14.3.2 `#define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))`

Version 2.3.1.

14.3.3 `#define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1`

Enables the FlexNVM support by default.

14.3.4 `#define FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT 1`

Enables the secondary flash support by default.

14.3.5 `#define FLASH_DRIVER_IS_FLASH_RESIDENT 1`

Used for the flash resident application.

14.3.6 `#define FLASH_DRIVER_IS_EXPORTED 0`

Used for the KSDK application.

14.3.7 `#define kStatusGroupGeneric 0`

14.3.8 `#define MAKE_STATUS( group, code ) (((group)*100) + (code))`

14.3.9 `#define FOUR_CHAR_CODE( a, b, c, d ) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))`

### 14.4 Enumeration Type Documentation

#### 14.4.1 enum \_flash\_driver\_version\_constants

Enumerator

*kFLASH\_DriverVersionName* Flash driver version name.

- kFLASH\_DriverVersionMajor*** Major flash driver version.
- kFLASH\_DriverVersionMinor*** Minor flash driver version.
- kFLASH\_DriverVersionBugfix*** Bugfix for flash driver version.

#### 14.4.2 enum \_flash\_status

Enumerator

- kStatus\_FLASH\_Success*** API is executed successfully.
- kStatus\_FLASH\_InvalidArgument*** Invalid argument.
- kStatus\_FLASH\_SizeError*** Error size.
- kStatus\_FLASH\_AlignmentError*** Parameter is not aligned with the specified baseline.
- kStatus\_FLASH\_AddressError*** Address is out of range.
- kStatus\_FLASH\_AccessError*** Invalid instruction codes and out-of bound addresses.
- kStatus\_FLASH\_ProtectionViolation*** The program/erase operation is requested to execute on protected areas.
- kStatus\_FLASH\_CommandFailure*** Run-time error during command execution.
- kStatus\_FLASH\_UnknownProperty*** Unknown property.
- kStatus\_FLASH\_EraseKeyError*** API erase key is invalid.
- kStatus\_FLASH\_RegionExecuteOnly*** The current region is execute-only.
- kStatus\_FLASH\_ExecuteInRamFunctionNotReady*** Execute-in-RAM function is not available.
- kStatus\_FLASH\_PartitionStatusUpdateFailure*** Failed to update partition status.
- kStatus\_FLASH\_SetFlexramAsEepromError*** Failed to set FlexRAM as EEPROM.
- kStatus\_FLASH\_RecoverFlexramAsRamError*** Failed to recover FlexRAM as RAM.
- kStatus\_FLASH\_SetFlexramAsRamError*** Failed to set FlexRAM as RAM.
- kStatus\_FLASH\_RecoverFlexramAsEepromError*** Failed to recover FlexRAM as EEPROM.
- kStatus\_FLASH\_CommandNotSupported*** Flash API is not supported.
- kStatus\_FLASH\_SwapSystemNotInUninitialized*** Swap system is not in an uninitialized state.
- kStatus\_FLASH\_SwapIndicatorAddressError*** The swap indicator address is invalid.
- kStatus\_FLASH\_ReadOnlyProperty*** The flash property is read-only.
- kStatus\_FLASH\_InvalidPropertyValue*** The flash property value is out of range.
- kStatus\_FLASH\_InvalidSpeculationOption*** The option of flash prefetch speculation is invalid.

#### 14.4.3 enum \_flash\_driver\_api\_keys

Note

The resulting value is built with a byte order such that the string being readable in expected order when viewed in a hex editor, if the value is treated as a 32-bit little endian value.

Enumerator

- kFLASH\_ApiEraseKey*** Key value used to validate all flash erase APIs.

## Enumeration Type Documentation

### 14.4.4 enum flash\_margin\_value\_t

Enumerator

***kFLASH\_MarginValueNormal*** Use the 'normal' read level for 1s.

***kFLASH\_MarginValueUser*** Apply the 'User' margin to the normal read-1 level.

***kFLASH\_MarginValueFactory*** Apply the 'Factory' margin to the normal read-1 level.

***kFLASH\_MarginValueInvalid*** Not real margin level, Used to determine the range of valid margin level.

### 14.4.5 enum flash\_security\_state\_t

Enumerator

***kFLASH\_SecurityStateNotSecure*** Flash is not secure.

***kFLASH\_SecurityStateBackdoorEnabled*** Flash backdoor is enabled.

***kFLASH\_SecurityStateBackdoorDisabled*** Flash backdoor is disabled.

### 14.4.6 enum flash\_protection\_state\_t

Enumerator

***kFLASH\_ProtectionStateUnprotected*** Flash region is not protected.

***kFLASH\_ProtectionStateProtected*** Flash region is protected.

***kFLASH\_ProtectionStateMixed*** Flash is mixed with protected and unprotected region.

### 14.4.7 enum flash\_execute\_only\_access\_state\_t

Enumerator

***kFLASH\_AccessStateUnLimited*** Flash region is unlimited.

***kFLASH\_AccessStateExecuteOnly*** Flash region is execute only.

***kFLASH\_AccessStateMixed*** Flash is mixed with unlimited and execute only region.

### 14.4.8 enum flash\_property\_tag\_t

Enumerator

***kFLASH\_PropertyPflashSectorSize*** Pflash sector size property.

***kFLASH\_PropertyPflashTotalSize*** Pflash total size property.

*kFLASH\_PropertyPflashBlockSize* Pflash block size property.  
*kFLASH\_PropertyPflashBlockCount* Pflash block count property.  
*kFLASH\_PropertyPflashBlockBaseAddr* Pflash block base address property.  
*kFLASH\_PropertyPflashFacSupport* Pflash fac support property.  
*kFLASH\_PropertyPflashAccessSegmentSize* Pflash access segment size property.  
*kFLASH\_PropertyPflashAccessSegmentCount* Pflash access segment count property.  
*kFLASH\_PropertyFlexRamBlockBaseAddr* FlexRam block base address property.  
*kFLASH\_PropertyFlexRamTotalSize* FlexRam total size property.  
*kFLASH\_PropertyDflashSectorSize* Dflash sector size property.  
*kFLASH\_PropertyDflashTotalSize* Dflash total size property.  
*kFLASH\_PropertyDflashBlockSize* Dflash block size property.  
*kFLASH\_PropertyDflashBlockCount* Dflash block count property.  
*kFLASH\_PropertyDflashBlockBaseAddr* Dflash block base address property.  
*kFLASH\_PropertyEepromTotalSize* EEPROM total size property.  
*kFLASH\_PropertyFlashMemoryIndex* Flash memory index property.  
*kFLASH\_PropertyFlashCacheControllerIndex* Flash cache controller index property.

#### 14.4.9 enum \_flash\_execute\_in\_ram\_function\_constants

Enumerator

*kFLASH\_ExecuteInRamFunctionMaxSizeInWords* The maximum size of execute-in-RAM function.  
*kFLASH\_ExecuteInRamFunctionTotalNum* Total number of execute-in-RAM functions.

#### 14.4.10 enum flash\_read\_resource\_option\_t

Enumerator

*kFLASH\_ResourceOptionFlashIfr* Select code for Program flash 0 IFR, Program flash swap 0 IFR, Data flash 0 IFR.  
*kFLASH\_ResourceOptionVersionId* Select code for the version ID.

#### 14.4.11 enum \_flash\_read\_resource\_range

Enumerator

*kFLASH\_ResourceRangePflashIfrSizeInBytes* Pflash IFR size in byte.  
*kFLASH\_ResourceRangeVersionIdSizeInBytes* Version ID IFR size in byte.  
*kFLASH\_ResourceRangeVersionIdStart* Version ID IFR start address.  
*kFLASH\_ResourceRangeVersionIdEnd* Version ID IFR end address.

## Enumeration Type Documentation

*kFLASH\_ResourceRangePflashSwapIfrStart* Pflash swap IFR start address.  
*kFLASH\_ResourceRangePflashSwapIfrEnd* Pflash swap IFR end address.  
*kFLASH\_ResourceRangeDflashIfrStart* Dflash IFR start address.  
*kFLASH\_ResourceRangeDflashIfrEnd* Dflash IFR end address.

### 14.4.12 enum \_k3\_flash\_read\_once\_index

Enumerator

*kFLASH\_RecordIndexSwapAddr* Index of Swap indicator address.  
*kFLASH\_RecordIndexSwapEnable* Index of Swap system enable.  
*kFLASH\_RecordIndexSwapDisable* Index of Swap system disable.

### 14.4.13 enum flash\_flexram\_function\_option\_t

Enumerator

*kFLASH\_FlexramFunctionOptionAvailableAsRam* An option used to make FlexRAM available as RAM.  
*kFLASH\_FlexramFunctionOptionAvailableForEeprom* An option used to make FlexRAM available for EEPROM.

### 14.4.14 enum flash\_swap\_function\_option\_t

Enumerator

*kFLASH\_SwapFunctionOptionEnable* An option used to enable the Swap function.  
*kFLASH\_SwapFunctionOptionDisable* An option used to disable the Swap function.

### 14.4.15 enum flash\_swap\_control\_option\_t

Enumerator

*kFLASH\_SwapControlOptionInitializeSystem* An option used to initialize the Swap system.  
*kFLASH\_SwapControlOptionSetInUpdateState* An option used to set the Swap in an update state.  
*kFLASH\_SwapControlOptionSetIncompleteState* An option used to set the Swap in a complete state.  
*kFLASH\_SwapControlOptionReportStatus* An option used to report the Swap status.  
*kFLASH\_SwapControlOptionDisableSystem* An option used to disable the Swap status.

**14.4.16 enum flash\_swap\_state\_t**

Enumerator

***kFLASH\_SwapStateUninitialized*** Flash Swap system is in an uninitialized state.

***kFLASH\_SwapStateReady*** Flash Swap system is in a ready state.

***kFLASH\_SwapStateUpdate*** Flash Swap system is in an update state.

***kFLASH\_SwapStateUpdateErased*** Flash Swap system is in an updateErased state.

***kFLASH\_SwapStateComplete*** Flash Swap system is in a complete state.

***kFLASH\_SwapStateDisabled*** Flash Swap system is in a disabled state.

**14.4.17 enum flash\_swap\_block\_status\_t**

Enumerator

***kFLASH\_SwapBlockStatusLowerHalfProgramBlocksAtZero*** Swap block status is that lower half program block at zero.

***kFLASH\_SwapBlockStatusUpperHalfProgramBlocksAtZero*** Swap block status is that upper half program block at zero.

**14.4.18 enum flash\_partition\_flexram\_load\_option\_t**

Enumerator

***kFLASH\_PartitionFlexramLoadOptionLoadedWithValidEepromData*** FlexRAM is loaded with valid EEPROM data during reset sequence.

***kFLASH\_PartitionFlexramLoadOptionNotLoaded*** FlexRAM is not loaded during reset sequence.

**14.4.19 enum flash\_memory\_index\_t**

Enumerator

***kFLASH\_MemoryIndexPrimaryFlash*** Current flash memory is primary flash.

***kFLASH\_MemoryIndexSecondaryFlash*** Current flash memory is secondary flash.

**14.4.20 enum flash\_cache\_controller\_index\_t**

Enumerator

***kFLASH\_CacheControllerIndexForCore0*** Current flash cache controller is for core 0.

***kFLASH\_CacheControllerIndexForCore1*** Current flash cache controller is for core 1.

## Function Documentation

### 14.4.21 enum flash\_cache\_clear\_process\_t

Enumerator

*kFLASH\_CacheClearProcessPre* Pre flash cache clear process.

*kFLASH\_CacheClearProcessPost* Post flash cache clear process.

## 14.5 Function Documentation

### 14.5.1 status\_t FLASH\_Init ( flash\_config\_t \* config )

This function checks and initializes the Flash module for the other Flash APIs.

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
---------------	--

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

### 14.5.2 status\_t FLASH\_SetCallback ( flash\_config\_t \* config, flash\_callback\_t callback )

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
<i>callback</i>	A callback function to be stored in the driver.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.

#### 14.5.3 **status\_t FLASH\_PreparesExecuteInRamFunctions ( flash\_config\_t \* config )**

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
---------------	--

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.

#### 14.5.4 **status\_t FLASH\_EraseAll ( flash\_config\_t \* config, uint32\_t key )**

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
<i>key</i>	A value used to validate all flash erase APIs.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Erase-KeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.

## Function Documentation

<i>kStatus_FLASH_Access_Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

### 14.5.5 **status\_t FLASH\_Erase ( flash\_config\_t \* config, uint32\_t start, uint32\_t lengthInBytes, uint32\_t key )**

This function erases the appropriate number of flash sectors based on the desired start address and length.

#### Parameters

<i>config</i>	The pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words) to be erased. Must be word-aligned.
<i>key</i>	The value used to validate all flash erase APIs.

#### Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FLASH_AddressError</i>	The address is out of range.

<i>kStatus_FLASH_EraseKeyError</i>	The API erase key is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

#### 14.5.6 **status\_t FLASH\_EraseAllExecuteOnlySegments ( flash\_config\_t \* config, uint32\_t key )**

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
<i>key</i>	A value used to validate all flash erase APIs.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_EraseKeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

## Function Documentation

<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

Erases all program flash execute-only segments defined by the FXACC registers.

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
<i>key</i>	A value used to validate all flash erase APIs.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_EraseKeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

### 14.5.7 **status\_t FLASH\_Program ( flash\_config\_t \* config, uint32\_t start, uint32\_t \* src, uint32\_t lengthInBytes )**

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the flash.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with the specified baseline.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

#### 14.5.8 **status\_t FLASH\_ProgramOnce ( flash\_config\_t \* config, uint32\_t index, uint32\_t \* src, uint32\_t lengthInBytes )**

This function programs the Program Once Field with the desired data for a given flash area as determined by the index and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
---------------	--

## Function Documentation

<i>index</i>	The index indicating which area of the Program Once Field to be programmed.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the Program Once Field.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

### 14.5.9 **status\_t FLASH\_ProgramSection ( flash\_config\_t \* config, uint32\_t start, uint32\_t \* src, uint32\_t lengthInBytes )**

This function programs the flash memory with the desired data for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the flash.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_SetFlexramAsRamError</i>	Failed to set flexram as RAM.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_RecoverFlexramAsEepromError</i>	Failed to recover FlexRAM as EEPROM.

#### 14.5.10 **status\_t FLASH\_EepromWrite ( flash\_config\_t \* config, uint32\_t start, uint8\_t \* src, uint32\_t lengthInBytes )**

This function programs the emulated EEPROM with the desired data for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.

## Function Documentation

<i>src</i>	A pointer to the source buffer of data that is to be programmed into the flash.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_SetFlexramAsEepromError</i>	Failed to set flexram as eeprom.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_RecoverFlexramAsRamError</i>	Failed to recover the FlexRAM as RAM.

### 14.5.11 **status\_t FLASH\_ReadResource ( *flash\_config\_t \* config*, *uint32\_t start*, *uint32\_t \* dst*, *uint32\_t lengthInBytes*, *flash\_read\_resource\_option\_t option* )**

This function reads the flash memory with the desired location for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>dst</i>	A pointer to the destination buffer of data that is used to store data to be read.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be read. Must be word-aligned.

<i>option</i>	The resource option which indicates which area should be read back.
---------------	---

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with the specified baseline.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

#### 14.5.12 **status\_t FLASH\_ReadOnce ( flash\_config\_t \* config, uint32\_t index, uint32\_t \* dst, uint32\_t lengthInBytes )**

This function reads the read once feild with given index and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>index</i>	The index indicating the area of program once field to be read.
<i>dst</i>	A pointer to the destination buffer of data that is used to store data to be read.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
------------------------------	--------------------------------

## Function Documentation

<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

### 14.5.13 **status\_t FLASH\_GetSecurityState ( flash\_config\_t \* config, flash\_security\_state\_t \* state )**

This function retrieves the current flash security status, including the security enabling state and the backdoor key enabling state.

Parameters

<i>config</i>	A pointer to storage for the driver runtime state.
<i>state</i>	A pointer to the value returned for the current security status code:

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.

### 14.5.14 **status\_t FLASH\_SecurityBypass ( flash\_config\_t \* config, const uint8\_t \* backdoorKey )**

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>backdoorKey</i>	A pointer to the user buffer containing the backdoor key.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

#### 14.5.15 **status\_t FLASH\_VerifyEraseAll ( flash\_config\_t \* *config*, flash\_margin\_value\_t *margin* )**

This function checks whether the flash is erased to the specified read margin level.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>margin</i>	Read margin choice.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.

## Function Documentation

<i>kStatus_FLASH_Access_Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

### 14.5.16 **status\_t FLASH\_VerifyErase ( flash\_config\_t \* config, uint32\_t start, uint32\_t lengthInBytes, flash\_margin\_value\_t margin )**

This function checks the appropriate number of flash sectors based on the desired start address and length to check whether the flash is erased to the specified read margin level.

#### Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be verified. The start address does not need to be sector-aligned but must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<i>margin</i>	Read margin choice.

#### Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.

<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

**14.5.17 `status_t FLASH_VerifyProgram ( flash_config_t * config, uint32_t start, uint32_t lengthInBytes, const uint32_t * expectedData, flash_margin_value_t margin, uint32_t * failedAddress, uint32_t * failedData )`**

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

#### Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be verified. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<i>expectedData</i>	A pointer to the expected data that is to be verified against.
<i>margin</i>	Read margin choice.
<i>failedAddress</i>	A pointer to the returned failing address.
<i>failedData</i>	A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

#### Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.

## Function Documentation

<i>kStatus_FLASH_Address-Error</i>	Address is out of range.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_-CommandFailure</i>	Run-time error during the command execution.

### 14.5.18 **status\_t FLASH\_VerifyEraseAllExecuteOnlySegments ( flash\_config\_t \* config, flash\_margin\_value\_t margin )**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>margin</i>	Read margin choice.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_-CommandFailure</i>	Run-time error during the command execution.

**14.5.19 status\_t FLASH\_IsProtected ( flash\_config\_t \* *config*, uint32\_t *start*,  
uint32\_t *lengthInBytes*, flash\_protection\_state\_t \* *protection\_state* )**

This function retrieves the current flash protect status for a given flash area as determined by the start address and length.

## Function Documentation

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be checked. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words) to be checked. Must be word-aligned.
<i>protection_state</i>	A pointer to the value returned for the current protection status code for the desired flash area.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_AddressError</i>	The address is out of range.

### 14.5.20 **status\_t FLASH\_IsExecuteOnly ( *flash\_config\_t \* config, uint32\_t start, uint32\_t lengthInBytes, flash\_execute\_only\_access\_state\_t \* access\_state* )**

This function retrieves the current flash access status for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be checked. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be checked. Must be word-aligned.
<i>access_state</i>	A pointer to the value returned for the current access status code for the desired flash area.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	The parameter is not aligned to the specified baseline.
<i>kStatus_FLASH_AddressError</i>	The address is out of range.

#### 14.5.21 **status\_t FLASHGetProperty ( flash\_config\_t \* config, flash\_property\_tag\_t whichProperty, uint32\_t \* value )**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>whichProperty</i>	The desired property from the list of properties in enum flash_property_tag_t
<i>value</i>	A pointer to the value returned for the desired flash property.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_UnknownProperty</i>	An unknown property tag.

#### 14.5.22 **status\_t FLASH\_SetProperty ( flash\_config\_t \* config, flash\_property\_tag\_t whichProperty, uint32\_t value )**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>whichProperty</i>	The desired property from the list of properties in enum flash_property_tag_t

## Function Documentation

<i>value</i>	A to set for the desired flash property.
--------------	--

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_UnknownProperty</i>	An unknown property tag.
<i>kStatus_FLASH_InvalidPropertyValue</i>	An invalid property value.
<i>kStatus_FLASH_ReadOnlyProperty</i>	An read-only property tag.

### 14.5.23 **status\_t FLASH\_SetFlexramFunction ( flash\_config\_t \* config, flash\_flexram\_function\_option\_t option )**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>option</i>	The option used to set the work mode of FlexRAM.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.
-------------------------------------	--

**14.5.24 status\_t FLASH\_ProgramPartition ( *flash\_config\_t \* config, flash\_partition\_flexram\_load\_option\_t option, uint32\_t eepromDataSizeCode, uint32\_t flexnvmPartitionCode* )**

Parameters

<i>config</i>	Pointer to storage for the driver runtime state.
<i>option</i>	The option used to set FlexRAM load behavior during reset.
<i>eepromData-SizeCode</i>	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
<i>flexnvm-PartitionCode</i>	Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	Invalid argument is provided.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.

**14.5.25 status\_t FLASH\_PflashSetProtection ( *flash\_config\_t \* config, pflash\_protection\_status\_t \* protectStatus* )**

## Function Documentation

### Parameters

<i>config</i>	A pointer to storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the PFlash protection register. Each bit is corresponding to protection of 1/32(64) of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

### Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.

### 14.5.26 **status\_t FLASH\_PflashGetProtection ( flash\_config\_t \* config, pflash\_protection\_status\_t \* protectStatus )**

### Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	Protect status returned by the PFlash IP. Each bit is corresponding to the protection of 1/32(64) of the total PFlash. The least significant bit corresponds to the lowest address area of the PFlash. The most significant bit corresponds to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

### Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.

### 14.5.27 **status\_t FLASH\_DflashSetProtection ( flash\_config\_t \* config, uint8\_t protectStatus )**

## Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the DFlash protection register. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

## Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_CommandNotSupported</i>	Flash API is not supported.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.

**14.5.28 `status_t FLASH_DflashGetProtection ( flash_config_t * config, uint8_t * protectStatus )`**

## Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash, and so on. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

## Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.

## Function Documentation

<i>kStatus_FLASH_- CommandNotSupported</i>	Flash API is not supported.
--	-----------------------------

### 14.5.29 **status\_t FLASH\_EepromSetProtection ( flash\_config\_t \* config, uint8\_t protectStatus )**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the EEPROM protection register. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of EEPROM, and so on. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_- CommandNotSupported</i>	Flash API is not supported.
<i>kStatus_FLASH_- CommandFailure</i>	Run-time error during command execution.

### 14.5.30 **status\_t FLASH\_EepromGetProtection ( flash\_config\_t \* config, uint8\_t \* protectStatus )**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of the EEPROM. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_CommandNotSupported</i>	Flash API is not supported.

## Function Documentation

# Chapter 15

## FlexBus: External Bus Interface Driver

### 15.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Crossbar External Bus Interface (FlexBus) block of MCUXpresso SDK devices.

A multifunction external bus interface is provided on the device with a basic functionality to interface to slave-only devices. It can be directly connected to the following asynchronous or synchronous devices with little or no additional circuitry.

- External ROMs
- Flash memories
- Programmable logic devices
- Other simple target (slave) devices

For asynchronous devices, a simple chip-select based interface can be used. The FlexBus interface has up to six general purpose chip-selects, FB\_CS[5:0]. The number of chip selects available depends on the device and its pin configuration.

### 15.2 FlexBus functional operation

To configure the FlexBus driver, use one of the two ways to configure the `flexbus_config_t` structure.

1. Using the `FLEXBUS_GetDefaultConfig()` function.
2. Set parameters in the `flexbus_config_t` structure.

To initialize and configure the FlexBus driver, call the `FLEXBUS_Init()` function and pass a pointer to the `flexbus_config_t` structure.

To de-initialize the FlexBus driver, call the `FLEXBUS_Deinit()` function.

### 15.3 Typical use case and example

This example shows how to write/read to external memory (MRAM) by using the FlexBus module.

```
flexbus_config_t flexbusUserConfig;

FLEXBUS_GetDefaultConfig(&flexbusUserConfig); /* Gets the default configuration. */
/* Configure some parameters when using MRAM */
flexbusUserConfig.waitStates          = 2U;           /* Wait 2 states */
flexbusUserConfig.chipBaseAddress    = MRAM_START_ADDRESS; /* MRAM address for using
                                                               FlexBus */
flexbusUserConfig.chipBaseAddressMask = 7U;           /* 512 kilobytes memory
                                                               size */
FLEXBUS_Init(FB, &flexbusUserConfig); /* Initializes and configures the FlexBus module */

/* Do something */

FLEXBUS_Deinit(FB);
```

## Typical use case and example

## Data Structures

- struct `flexbus_config_t`  
*Configuration structure that the user needs to set. [More...](#)*

## Enumerations

- enum `flexbus_port_size_t`{  
  `kFLEXBUS_4Bytes` = 0x00U,  
  `kFLEXBUS_1Byte` = 0x01U,  
  `kFLEXBUS_2Bytes` = 0x02U }  
*Defines port size for FlexBus peripheral.*
- enum `flexbus_write_address_hold_t`{  
  `kFLEXBUS_Hold1Cycle` = 0x00U,  
  `kFLEXBUS_Hold2Cycles` = 0x01U,  
  `kFLEXBUS_Hold3Cycles` = 0x02U,  
  `kFLEXBUS_Hold4Cycles` = 0x03U }  
*Defines number of cycles to hold address and attributes for FlexBus peripheral.*
- enum `flexbus_read_address_hold_t`{  
  `kFLEXBUS_Hold1Or0Cycles` = 0x00U,  
  `kFLEXBUS_Hold2Or1Cycles` = 0x01U,  
  `kFLEXBUS_Hold3Or2Cycle` = 0x02U,  
  `kFLEXBUS_Hold4Or3Cycle` = 0x03U }  
*Defines number of cycles to hold address and attributes for FlexBus peripheral.*
- enum `flexbus_address_setup_t`{  
  `kFLEXBUS_FirstRisingEdge` = 0x00U,  
  `kFLEXBUS_SecondRisingEdge` = 0x01U,  
  `kFLEXBUS_ThirdRisingEdge` = 0x02U,  
  `kFLEXBUS_FourthRisingEdge` = 0x03U }  
*Address setup for FlexBus peripheral.*
- enum `flexbus_bytelane_shift_t`{  
  `kFLEXBUS_NotShifted` = 0x00U,  
  `kFLEXBUS_Shifted` = 0x01U }  
*Defines byte-lane shift for FlexBus peripheral.*
- enum `flexbus_multiplex_group1_t`{  
  `kFLEXBUS_MultiplexGroup1_FB_ALE` = 0x00U,  
  `kFLEXBUS_MultiplexGroup1_FB_CS1` = 0x01U,  
  `kFLEXBUS_MultiplexGroup1_FB_TS` = 0x02U }  
*Defines multiplex group1 valid signals.*
- enum `flexbus_multiplex_group2_t`{  
  `kFLEXBUS_MultiplexGroup2_FB_CS4` = 0x00U,  
  `kFLEXBUS_MultiplexGroup2_FB_TSIZ0` = 0x01U,  
  `kFLEXBUS_MultiplexGroup2_FB_BE_31_24` = 0x02U }  
*Defines multiplex group2 valid signals.*
- enum `flexbus_multiplex_group3_t`{  
  `kFLEXBUS_MultiplexGroup3_FB_CS5` = 0x00U,  
  `kFLEXBUS_MultiplexGroup3_FB_TSIZ1` = 0x01U,  
  `kFLEXBUS_MultiplexGroup3_FB_BE_23_16` = 0x02U }

- *Defines multiplex group3 valid signals.*  
enum **flexbus\_multiplex\_group4\_t** {  
  kFLEXBUS\_MultiplexGroup4\_FB\_TBST = 0x00U,  
  kFLEXBUS\_MultiplexGroup4\_FB\_CS2 = 0x01U,  
  kFLEXBUS\_MultiplexGroup4\_FB\_BE\_15\_8 = 0x02U }
- *Defines multiplex group4 valid signals.*  
enum **flexbus\_multiplex\_group5\_t** {  
  kFLEXBUS\_MultiplexGroup5\_FB\_TA = 0x00U,  
  kFLEXBUS\_MultiplexGroup5\_FB\_CS3 = 0x01U,  
  kFLEXBUS\_MultiplexGroup5\_FB\_BE\_7\_0 = 0x02U }
- *Defines multiplex group5 valid signals.*

## Driver version

- #define **FSL\_FLEXBUS\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 1))  
*Version 2.0.1.*

## FlexBus functional operation

- void **FLEXBUS\_Init** (FB\_Type \*base, const **flexbus\_config\_t** \*config)  
*Initializes and configures the FlexBus module.*
- void **FLEXBUS\_Deinit** (FB\_Type \*base)  
*De-initializes a FlexBus instance.*
- void **FLEXBUS\_GetDefaultConfig** (**flexbus\_config\_t** \*config)  
*Initializes the FlexBus configuration structure.*

## 15.4 Data Structure Documentation

### 15.4.1 struct **flexbus\_config\_t**

#### Data Fields

- uint8\_t **chip**  
*Chip FlexBus for validation.*
- uint8\_t **waitStates**  
*Value of wait states.*
- uint32\_t **chipBaseAddress**  
*Chip base address for using FlexBus.*
- uint32\_t **chipBaseAddressMask**  
*Chip base address mask.*
- bool **writeProtect**  
*Write protected.*
- bool **burstWrite**  
*Burst-Write enable.*
- bool **burstRead**  
*Burst-Read enable.*
- bool **byteEnableMode**  
*Byte-enable mode support.*
- bool **autoAcknowledge**

## Enumeration Type Documentation

- `bool extendTransferAddress`  
*Auto acknowledge setting.*
- `bool secondaryWaitStates`  
*Extend transfer start/extend address latch enable.*
- `flexbus_port_size_t portSize`  
*Secondary wait states number.*
- `flexbus_bytelane_shift_t byteLaneShift`  
*Port size of transfer.*
- `flexbus_write_address_hold_t writeAddressHold`  
*Byte-lane shift enable.*
- `flexbus_read_address_hold_t readAddressHold`  
*Write address hold or deselect option.*
- `flexbus_address_hold_t addressSetup`  
*Read address hold or deselect option.*
- `flexbus_multiplex_group1_t group1MultiplexControl`  
*Address setup setting.*
- `flexbus_multiplex_group2_t group2MultiplexControl`  
*FlexBus Signal Group 1 Multiplex control.*
- `flexbus_multiplex_group3_t group3MultiplexControl`  
*FlexBus Signal Group 2 Multiplex control.*
- `flexbus_multiplex_group4_t group4MultiplexControl`  
*FlexBus Signal Group 3 Multiplex control.*
- `flexbus_multiplex_group5_t group5MultiplexControl`  
*FlexBus Signal Group 4 Multiplex control.*
- `flexbus_multiplex_group6_t group6MultiplexControl`  
*FlexBus Signal Group 5 Multiplex control.*

## 15.5 Macro Definition Documentation

### 15.5.1 `#define FSL_FLEXBUS_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

## 15.6 Enumeration Type Documentation

### 15.6.1 `enum flexbus_port_size_t`

Enumerator

`kFLEXBUS_4Bytes` 32-bit port size

`kFLEXBUS_1Byte` 8-bit port size

`kFLEXBUS_2Bytes` 16-bit port size

### 15.6.2 `enum flexbus_write_address_hold_t`

Enumerator

`kFLEXBUS_Hold1Cycle` Hold address and attributes one cycles after FB\_CSn negates on writes.

`kFLEXBUS_Hold2Cycles` Hold address and attributes two cycles after FB\_CSn negates on writes.

`kFLEXBUS_Hold3Cycles` Hold address and attributes three cycles after FB\_CSn negates on writes.

***kFLEXBUS\_Hold4Cycles*** Hold address and attributes four cycles after FB\_CSn negates on writes.

### 15.6.3 enum flexbus\_read\_address\_hold\_t

Enumerator

***kFLEXBUS\_Hold1Or0Cycles*** Hold address and attributes 1 or 0 cycles on reads.

***kFLEXBUS\_Hold2Or1Cycles*** Hold address and attributes 2 or 1 cycles on reads.

***kFLEXBUS\_Hold3Or2Cycle*** Hold address and attributes 3 or 2 cycles on reads.

***kFLEXBUS\_Hold4Or3Cycle*** Hold address and attributes 4 or 3 cycles on reads.

### 15.6.4 enum flexbus\_address\_setup\_t

Enumerator

***kFLEXBUS\_FirstRisingEdge*** Assert FB\_CSn on first rising clock edge after address is asserted.

***kFLEXBUS\_SecondRisingEdge*** Assert FB\_CSn on second rising clock edge after address is asserted.

***kFLEXBUS\_ThirdRisingEdge*** Assert FB\_CSn on third rising clock edge after address is asserted.

***kFLEXBUS\_FourthRisingEdge*** Assert FB\_CSn on fourth rising clock edge after address is asserted.

### 15.6.5 enum flexbus\_bytelane\_shift\_t

Enumerator

***kFLEXBUS\_NotShifted*** Not shifted. Data is left-justified on FB\_AD

***kFLEXBUS\_Shifted*** Shifted. Data is right justified on FB\_AD

### 15.6.6 enum flexbus\_multiplex\_group1\_t

Enumerator

***kFLEXBUS\_MultiplexGroup1\_FB\_ALE*** FB\_ALE.

***kFLEXBUS\_MultiplexGroup1\_FB\_CS1*** FB\_CS1.

***kFLEXBUS\_MultiplexGroup1\_FB\_TS*** FB\_TS.

## Function Documentation

### 15.6.7 enum flexbus\_multiplex\_group2\_t

Enumerator

*kFLEXBUS\_MultiplexGroup2\_FB\_CS4* FB\_CS4.  
*kFLEXBUS\_MultiplexGroup2\_FB\_TSIZ0* FB\_TSIZ0.  
*kFLEXBUS\_MultiplexGroup2\_FB\_BE\_31\_24* FB\_BE\_31\_24.

### 15.6.8 enum flexbus\_multiplex\_group3\_t

Enumerator

*kFLEXBUS\_MultiplexGroup3\_FB\_CS5* FB\_CS5.  
*kFLEXBUS\_MultiplexGroup3\_FB\_TSIZ1* FB\_TSIZ1.  
*kFLEXBUS\_MultiplexGroup3\_FB\_BE\_23\_16* FB\_BE\_23\_16.

### 15.6.9 enum flexbus\_multiplex\_group4\_t

Enumerator

*kFLEXBUS\_MultiplexGroup4\_FB\_TBST* FB\_TBST.  
*kFLEXBUS\_MultiplexGroup4\_FB\_CS2* FB\_CS2.  
*kFLEXBUS\_MultiplexGroup4\_FB\_BE\_15\_8* FB\_BE\_15\_8.

### 15.6.10 enum flexbus\_multiplex\_group5\_t

Enumerator

*kFLEXBUS\_MultiplexGroup5\_FB\_TA* FB\_TA.  
*kFLEXBUS\_MultiplexGroup5\_FB\_CS3* FB\_CS3.  
*kFLEXBUS\_MultiplexGroup5\_FB\_BE\_7\_0* FB\_BE\_7\_0.

## 15.7 Function Documentation

### 15.7.1 void FLEXBUS\_Init ( FB\_Type \* *base*, const flexbus\_config\_t \* *config* )

This function enables the clock gate for FlexBus module. Only chip 0 is validated and set to known values. Other chips are disabled. Note that in this function, certain parameters, depending on external memories, must be set before using the [FLEXBUS\\_Init\(\)](#) function. This example shows how to set up the *uart\_state\_t* and the *flexbus\_config\_t* parameters and how to call the FLEXBUS\_Init function by passing in these parameters.

```

flexbus_config_t flexbusConfig;
FLEXBUS_GetDefaultConfig(&flexbusConfig);
flexbusConfig.waitStates          = 2U;
flexbusConfig.chipBaseAddress     = 0x60000000U;
flexbusConfig.chipBaseAddressMask = 7U;
FLEXBUS_Init(FB, &flexbusConfig);

```

## Parameters

<i>base</i>	FlexBus peripheral address.
<i>config</i>	Pointer to the configuration structure

**15.7.2 void FLEXBUS\_Deinit ( FB\_Type \* *base* )**

This function disables the clock gate of the FlexBus module clock.

## Parameters

<i>base</i>	FlexBus peripheral address.
-------------	-----------------------------

**15.7.3 void FLEXBUS\_GetDefaultConfig ( flexbus\_config\_t \* *config* )**

This function initializes the FlexBus configuration structure to default value. The default values are.

```

fbConfig->chip                  = 0;
fbConfig->writeProtect          = 0;
fbConfig->burstWrite            = 0;
fbConfig->burstRead             = 0;
fbConfig->byteEnableMode        = 0;
fbConfig->autoAcknowledge       = true;
fbConfig->extendTransferAddress = 0;
fbConfig->secondaryWaitStates   = 0;
fbConfig->byteLaneShift         = kFLEXBUS_NotShifted;
fbConfig->writeAddressHold      = kFLEXBUS_Hold1Cycle;
fbConfig->readAddressHold       = kFLEXBUS_Hold1Or0Cycles;
fbConfig->addressSetup          = kFLEXBUS_FirstRisingEdge;
fbConfig->portSize              = kFLEXBUS_1Byte;
fbConfig->group1MultiplexControl = kFLEXBUS_MultiplexGroup1_FB_ALE;
fbConfig->group2MultiplexControl = kFLEXBUS_MultiplexGroup2_FB_CS4 ;
fbConfig->group3MultiplexControl = kFLEXBUS_MultiplexGroup3_FB_CS5;
fbConfig->group4MultiplexControl = kFLEXBUS_MultiplexGroup4_FB_TBST;
fbConfig->group5MultiplexControl = kFLEXBUS_MultiplexGroup5_FB_TA;

```

## Function Documentation

Parameters

<i>config</i>	Pointer to the initialization structure.
---------------	--

See Also

[FLEXBUS\\_Init](#)

# Chapter 16

## FlexCAN: Flex Controller Area Network Driver

### 16.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Flex Controller Area Network (FlexCAN) module of MCUXpresso SDK devices.

### Modules

- [FlexCAN Driver](#)
- [FlexCAN eDMA Driver](#)

## FlexCAN Driver

### 16.2 FlexCAN Driver

#### 16.2.1 Overview

This section describes the programming interface of the FlexCAN driver. The FlexCAN driver configures FlexCAN module and provides functional and transactional interfaces to build the FlexCAN application.

#### 16.2.2 Typical use case

##### 16.2.2.1 Message Buffer Send Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t txFrame;

/* Init FlexCAN module. */
FLEXCAN_SetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enable FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the transmit message buffer. */
FLEXCAN_SetTxMbConfig(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, true);

/* Prepares the transmit frame for sending. */
txFrame.format = KFLEXCAN_FrameFormatStandard;
txFrame.type   = KFLEXCAN_FrameTypeData;
txFrame.id     = FLEXCAN_ID_STD(0x123);
txFrame.length = 8;
txFrame.dataWord0 = CAN_WORD0_DATA_BYTE_0(0x11) |
                   CAN_WORD0_DATA_BYTE_1(0x22) |
                   CAN_WORD0_DATA_BYTE_2(0x33) |
                   CAN_WORD0_DATA_BYTE_3(0x44);
txFrame.dataWord1 = CAN_WORD1_DATA_BYTE_4(0x55) |
                   CAN_WORD1_DATA_BYTE_5(0x66) |
                   CAN_WORD1_DATA_BYTE_6(0x77) |
                   CAN_WORD1_DATA_BYTE_7(0x88);
/* Writes a transmit message buffer to send a CAN Message. */
FLEXCAN_WriteTxMb(EXAMPLE_CAN, TX_MESSAGE_BUFFER_INDEX, &txFrame);

/* Waits until the transmit message buffer is empty. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX));

/* Cleans the transmit message buffer empty status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << TX_MESSAGE_BUFFER_INDEX);
```

##### 16.2.2.2 Message Buffer Receive Operation

```
flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module. */
FLEXCAN_SetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive message buffer. */
```

```

mbConfig.format      = KFLEXCAN_FrameFormatStandard;
mbConfig.type        = KFLEXCAN_FrameTypeData;
mbConfig.id          = FLEXCAN_ID_STD(0x123);
FLEXCAN_SetRxMbConfig(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &mbConfig, true);

/* Waits until the receive message buffer is full. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX));

/* Reads the received message from the receive message buffer. */
FLEXCAN_ReadRxMb(EXAMPLE_CAN, RX_MESSAGE_BUFFER_INDEX, &rxFrame);

/* Cleans the receive message buffer full status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, 1 << RX_MESSAGE_BUFFER_INDEX);

```

### 16.2.2.3 Receive FIFO Operation

```

uint32_t rxFifoFilter[] = {FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x321, 1, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 0, 0),
                           FLEXCAN_RX_FIFO_STD_FILTER_TYPE_A(0x123, 1, 0)};
;

flexcan_config_t flexcanConfig;
flexcan_frame_t rxFrame;

/* Initializes the FlexCAN module. */
FLEXCAN_GetDefaultConfig(&flexcanConfig);
FLEXCAN_Init(EXAMPLE_CAN, &flexcanConfig);

/* Enables the FlexCAN module. */
FLEXCAN_Enable(EXAMPLE_CAN, true);

/* Sets up the receive FIFO. */
rxFifoConfig.idFilterTable = rxFifoFilter;
rxFifoConfig.idFilterType  = KFLEXCAN_RxFifoFilterTypeA;
rxFifoConfig.idFilterNum   = sizeof(rxFifoFilter) / sizeof(rxFifoFilter[0]);
rxFifoConfig.priority     = KFLEXCAN_RxFifoPrioHigh;
FLEXCAN_SetRxFifoConfig(EXAMPLE_CAN, &rxFifoConfig, true);

/* Waits until the receive FIFO becomes available. */
while (!FLEXCAN_GetMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag));

/* Reads the message from the receive FIFO. */
FLEXCAN_ReadRxFifo(EXAMPLE_CAN, &rxFrame);

/* Cleans the receive FIFO available status. */
FLEXCAN_ClearMbStatusFlags(EXAMPLE_CAN, KFLEXCAN_RxFifoFrameAvlFlag);

```

## Data Structures

- struct [flexcan\\_frame\\_t](#)  
*FlexCAN message frame structure.* [More...](#)
- struct [flexcan\\_config\\_t](#)  
*FlexCAN module configuration structure.* [More...](#)
- struct [flexcan\\_timing\\_config\\_t](#)  
*FlexCAN protocol timing characteristic configuration structure.* [More...](#)
- struct [flexcan\\_rx\\_mb\\_config\\_t](#)  
*FlexCAN Receive Message Buffer configuration structure.* [More...](#)
- struct [flexcan\\_rx\\_fifo\\_config\\_t](#)

## FlexCAN Driver

- *FlexCAN Rx FIFO configuration structure.* [More...](#)
- struct **flexcan\_mb\_transfer\_t**  
*FlexCAN Message Buffer transfer.* [More...](#)
- struct **flexcan\_fifo\_transfer\_t**  
*FlexCAN Rx FIFO transfer.* [More...](#)
- struct **flexcan\_handle\_t**  
*FlexCAN handle structure.* [More...](#)

## Macros

- #define **FLEXCAN\_ID\_STD**(id) (((uint32\_t)((uint32\_t)(id)) << CAN\_ID\_STD\_SHIFT)) & CAN\_ID\_STD\_MASK)  
*FlexCAN Frame ID helper macro.*
- #define **FLEXCAN\_ID\_EXT**(id)  
*Extend Frame ID helper macro.*
- #define **FLEXCAN\_RX\_MB\_STD\_MASK**(id, rtr, ide)  
*FlexCAN Rx Message Buffer Mask helper macro.*
- #define **FLEXCAN\_RX\_MB\_EXT\_MASK**(id, rtr, ide)  
*Extend Rx Message Buffer Mask helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_A**(id, rtr, ide)  
*FlexCAN Rx FIFO Mask helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_B\_HIGH**(id, rtr, ide)  
*Standard Rx FIFO Mask helper macro Type B upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_B\_LOW**(id, rtr, ide)  
*Standard Rx FIFO Mask helper macro Type B lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_HIGH**(id) (((uint32\_t)(id) & 0x7F8) << 21)  
*Standard Rx FIFO Mask helper macro Type C upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_MID\_HIGH**(id) (((uint32\_t)(id) & 0x7F8) << 13)  
*Standard Rx FIFO Mask helper macro Type C mid-upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_MID\_LOW**(id) (((uint32\_t)(id) & 0x7F8) << 5)  
*Standard Rx FIFO Mask helper macro Type C mid-lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_LOW**(id) (((uint32\_t)(id) & 0x7F8) >> 3)  
*Standard Rx FIFO Mask helper macro Type C lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_A**(id, rtr, ide)  
*Extend Rx FIFO Mask helper macro Type A helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_B\_HIGH**(id, rtr, ide)  
*Extend Rx FIFO Mask helper macro Type B upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_B\_LOW**(id, rtr, ide)  
*Extend Rx FIFO Mask helper macro Type B lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_HIGH**(id) ((FLEXCAN\_ID\_EXT(id) & 0x1FE00000) << 3)  
*Extend Rx FIFO Mask helper macro Type C upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_MID\_HIGH**(id)  
*Extend Rx FIFO Mask helper macro Type C mid-upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_MID\_LOW**(id)  
*Extend Rx FIFO Mask helper macro Type C mid-lower part helper macro.*

- #define **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_LOW**(id) ((FLEXCAN\_ID\_EXT(id) & 0x1FE00000) >> 21)
 

*Extend Rx FIFO Mask helper macro Type C lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_A**(id, rtr, ide) **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_A**(id, rtr, ide)
 

*FlexCAN Rx FIFO Filter helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_B\_HIGH**(id, rtr, ide)
 

*Standard Rx FIFO Filter helper macro Type B upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_B\_LOW**(id, rtr, ide)
 

*Standard Rx FIFO Filter helper macro Type B lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_HIGH**(id)
 

*Standard Rx FIFO Filter helper macro Type C upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_MID\_HIGH**(id)
 

*Standard Rx FIFO Filter helper macro Type C mid-upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_MID\_LOW**(id)
 

*Standard Rx FIFO Filter helper macro Type C mid-lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_LOW**(id) **FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_LOW**(id)
 

*Standard Rx FIFO Filter helper macro Type C lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_FILTER\_TYPE\_A**(id, rtr, ide) **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_A**(id, rtr, ide)
 

*Extend Rx FIFO Filter helper macro Type A helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_FILTER\_TYPE\_B\_HIGH**(id, rtr, ide)
 

*Extend Rx FIFO Filter helper macro Type B upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_FILTER\_TYPE\_B\_LOW**(id, rtr, ide)
 

*Extend Rx FIFO Filter helper macro Type B lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_FILTER\_TYPE\_C\_HIGH**(id) **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_HIGH**(id)
 

*Extend Rx FIFO Filter helper macro Type C upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_FILTER\_TYPE\_C\_MID\_HIGH**(id)
 

*Extend Rx FIFO Filter helper macro Type C mid-upper part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_FILTER\_TYPE\_C\_MID\_LOW**(id)
 

*Extend Rx FIFO Filter helper macro Type C mid-lower part helper macro.*
- #define **FLEXCAN\_RX\_FIFO\_EXT\_FILTER\_TYPE\_C\_LOW**(id) **FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_LOW**(id)
 

*Extend Rx FIFO Filter helper macro Type C lower part helper macro.*

## Typedefs

- typedef void(\* **flexcan\_transfer\_callback\_t** )(CAN\_Type \*base, flexcan\_handle\_t \*handle, status\_t status, uint32\_t result, void \*userData)
 

*FlexCAN transfer callback function.*

### Enumerations

- enum `_flexcan_status` {  
    kStatus\_FLEXCAN\_TxBusy = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 0),  
    kStatus\_FLEXCAN\_TxIdle = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 1),  
    kStatus\_FLEXCAN\_TxSwitchToRx,  
    kStatus\_FLEXCAN\_RxBusy = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 3),  
    kStatus\_FLEXCAN\_RxIdle = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 4),  
    kStatus\_FLEXCAN\_RxOverflow = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 5),  
    kStatus\_FLEXCAN\_RxFifoBusy = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 6),  
    kStatus\_FLEXCAN\_RxFifoIdle = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 7),  
    kStatus\_FLEXCAN\_RxFifoOverflow = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 8),  
    kStatus\_FLEXCAN\_RxFifoWarning = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 9),  
    kStatus\_FLEXCAN\_ErrorStatus = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 10),  
    kStatus\_FLEXCAN\_UnHandled = MAKE\_STATUS(kStatusGroup\_FLEXCAN, 11) }  
        *FlexCAN transfer status.*
- enum `flexcan_frame_format_t` {  
    kFLEXCAN\_FrameFormatStandard = 0x0U,  
    kFLEXCAN\_FrameFormatExtend = 0x1U }  
        *FlexCAN frame format.*
- enum `flexcan_frame_type_t` {  
    kFLEXCAN\_FrameTypeData = 0x0U,  
    kFLEXCAN\_FrameTypeRemote = 0x1U }  
        *FlexCAN frame type.*
- enum `flexcan_clock_source_t` {  
    kFLEXCAN\_ClkSrcOsc = 0x0U,  
    kFLEXCAN\_ClkSrcPeri = 0x1U }  
        *FlexCAN clock source.*
- enum `flexcan_rx_fifo_filter_type_t` {  
    kFLEXCAN\_RxFifoFilterTypeA = 0x0U,  
    kFLEXCAN\_RxFifoFilterTypeB,  
    kFLEXCAN\_RxFifoFilterTypeC,  
    kFLEXCAN\_RxFifoFilterTypeD = 0x3U }  
        *FlexCAN Rx Fifo Filter type.*
- enum `flexcan_rx_fifo_priority_t` {  
    kFLEXCAN\_RxFifoPrioLow = 0x0U,  
    kFLEXCAN\_RxFifoPrioHigh = 0x1U }  
        *FlexCAN Rx FIFO priority.*
- enum `_flexcan_interrupt_enable` {  
    kFLEXCAN\_BusOffInterruptEnable = CAN\_CTRL1\_BOFFMSK\_MASK,  
    kFLEXCAN\_ErrorInterruptEnable = CAN\_CTRL1\_ERRMSK\_MASK,  
    kFLEXCAN\_RxWarningInterruptEnable = CAN\_CTRL1\_RWRNMSK\_MASK,  
    kFLEXCAN\_TxWarningInterruptEnable = CAN\_CTRL1\_TWRNMSK\_MASK,  
    kFLEXCAN\_WakeUpInterruptEnable = CAN\_MCR\_WAKMSK\_MASK }  
        *FlexCAN interrupt configuration structure, default settings all disabled.*
- enum `_flexcan_flags` {

```

kFLEXCAN_SynchFlag = CAN_ESR1_SYNCH_MASK,
kFLEXCAN_TxWarningIntFlag = CAN_ESR1_TWRNINT_MASK,
kFLEXCAN_RxWarningIntFlag = CAN_ESR1_RWRNINT_MASK,
kFLEXCAN_TxErrorWarningFlag = CAN_ESR1_TXWRN_MASK,
kFLEXCAN_RxErrorWarningFlag = CAN_ESR1_RXWRN_MASK,
kFLEXCAN_IdleFlag = CAN_ESR1_IDLE_MASK,
kFLEXCAN_FaultConfinementFlag = CAN_ESR1_FLTCONF_MASK,
kFLEXCAN_TransmittingFlag = CAN_ESR1_TX_MASK,
kFLEXCAN_ReceivingFlag = CAN_ESR1_RX_MASK,
kFLEXCAN_BusOffIntFlag = CAN_ESR1_BOFFINT_MASK,
kFLEXCAN_ErrorIntFlag = CAN_ESR1_ERRINT_MASK,
kFLEXCAN_WakeUpIntFlag = CAN_ESR1_WAKINT_MASK,
kFLEXCAN_ErrorFlag }

```

*FlexCAN status flags.*

- enum `_flexcan_error_flags` {

```

kFLEXCAN_StuffingError = CAN_ESR1_STFERR_MASK,
kFLEXCAN_FormError = CAN_ESR1_FRMERR_MASK,
kFLEXCAN_CrcError = CAN_ESR1_CRCERR_MASK,
kFLEXCAN_AckError = CAN_ESR1_ACKERR_MASK,
kFLEXCAN_Bit0Error = CAN_ESR1_BIT0ERR_MASK,
kFLEXCAN_Bit1Error = CAN_ESR1_BIT1ERR_MASK }

```

*FlexCAN error status flags.*

- enum `_flexcan_rx_fifo_flags` {

```

kFLEXCAN_RxFifoOverflowFlag = CAN_IFLAG1_BUF7I_MASK,
kFLEXCAN_RxFifoWarningFlag = CAN_IFLAG1_BUF6I_MASK,
kFLEXCAN_RxFifoFrameAvlFlag = CAN_IFLAG1_BUF5I_MASK }

```

*FlexCAN Rx FIFO status flags.*

## Driver version

- #define `FLEXCAN_DRIVER_VERSION` (MAKE\_VERSION(2, 2, 0))  
*FlexCAN driver version 2.2.0.*

## Initialization and deinitialization

- void `FLEXCAN_Init` (CAN\_Type \*base, const `flexcan_config_t` \*config, uint32\_t sourceClock\_Hz)  
*Initializes a FlexCAN instance.*
- void `FLEXCAN_Deinit` (CAN\_Type \*base)  
*De-initializes a FlexCAN instance.*
- void `FLEXCAN_GetDefaultConfig` (`flexcan_config_t` \*config)  
*Gets the default configuration structure.*

## FlexCAN Driver

### Configuration.

- void [FLEXCAN\\_SetTimingConfig](#) (CAN\_Type \*base, const [flexcan\\_timing\\_config\\_t](#) \*config)  
*Sets the FlexCAN protocol timing characteristic.*
- void [FLEXCAN\\_SetRxMbGlobalMask](#) (CAN\_Type \*base, uint32\_t mask)  
*Sets the FlexCAN receive message buffer global mask.*
- void [FLEXCAN\\_SetRxFifoGlobalMask](#) (CAN\_Type \*base, uint32\_t mask)  
*Sets the FlexCAN receive FIFO global mask.*
- void [FLEXCAN\\_SetRxIndividualMask](#) (CAN\_Type \*base, uint8\_t maskIdx, uint32\_t mask)  
*Sets the FlexCAN receive individual mask.*
- void [FLEXCAN\\_SetTxMbConfig](#) (CAN\_Type \*base, uint8\_t mbIdx, bool enable)  
*Configures a FlexCAN transmit message buffer.*
- void [FLEXCAN\\_SetRxMbConfig](#) (CAN\_Type \*base, uint8\_t mbIdx, const [flexcan\\_rx\\_mb\\_config\\_t](#) \*config, bool enable)  
*Configures a FlexCAN Receive Message Buffer.*
- void [FLEXCAN\\_SetRxFifoConfig](#) (CAN\_Type \*base, const [flexcan\\_rx\\_fifo\\_config\\_t](#) \*config, bool enable)  
*Configures the FlexCAN Rx FIFO.*

### Status

- static uint32\_t [FLEXCAN\\_GetStatusFlags](#) (CAN\_Type \*base)  
*Gets the FlexCAN module interrupt flags.*
- static void [FLEXCAN\\_ClearStatusFlags](#) (CAN\_Type \*base, uint32\_t mask)  
*Clears status flags with the provided mask.*
- static void [FLEXCAN\\_GetBusErrCount](#) (CAN\_Type \*base, uint8\_t \*txErrBuf, uint8\_t \*rxErrBuf)  
*Gets the FlexCAN Bus Error Counter value.*
- static uint32\_t [FLEXCAN\\_GetMbStatusFlags](#) (CAN\_Type \*base, uint32\_t mask)  
*Gets the FlexCAN Message Buffer interrupt flags.*
- static void [FLEXCAN\\_ClearMbStatusFlags](#) (CAN\_Type \*base, uint32\_t mask)  
*Clears the FlexCAN Message Buffer interrupt flags.*

### Interrupts

- static void [FLEXCAN\\_EnableInterrupts](#) (CAN\_Type \*base, uint32\_t mask)  
*Enables FlexCAN interrupts according to the provided mask.*
- static void [FLEXCAN\\_DisableInterrupts](#) (CAN\_Type \*base, uint32\_t mask)  
*Disables FlexCAN interrupts according to the provided mask.*
- static void [FLEXCAN\\_EnableMbInterrupts](#) (CAN\_Type \*base, uint32\_t mask)  
*Enables FlexCAN Message Buffer interrupts.*
- static void [FLEXCAN\\_DisableMbInterrupts](#) (CAN\_Type \*base, uint32\_t mask)  
*Disables FlexCAN Message Buffer interrupts.*

### Bus Operations

- static void [FLEXCAN\\_Enable](#) (CAN\_Type \*base, bool enable)  
*Enables or disables the FlexCAN module operation.*

- status\_t **FLEXCAN\_WriteTxMb** (CAN\_Type \*base, uint8\_t mbIdx, const flexcan\_frame\_t \*txFrame)  
*Writes a FlexCAN Message to the Transmit Message Buffer.*
- status\_t **FLEXCAN\_ReadRxMb** (CAN\_Type \*base, uint8\_t mbIdx, flexcan\_frame\_t \*rxFrame)  
*Reads a FlexCAN Message from Receive Message Buffer.*
- status\_t **FLEXCAN\_ReadRxFifo** (CAN\_Type \*base, flexcan\_frame\_t \*rxFrame)  
*Reads a FlexCAN Message from Rx FIFO.*

## Transactional

- status\_t **FLEXCAN\_TransferSendBlocking** (CAN\_Type \*base, uint8\_t mbIdx, flexcan\_frame\_t \*txFrame)  
*Performs a polling send transaction on the CAN bus.*
- status\_t **FLEXCAN\_TransferReceiveBlocking** (CAN\_Type \*base, uint8\_t mbIdx, flexcan\_frame\_t \*rxFrame)  
*Performs a polling receive transaction on the CAN bus.*
- status\_t **FLEXCAN\_TransferReceiveFifoBlocking** (CAN\_Type \*base, flexcan\_frame\_t \*rxFrame)  
*Performs a polling receive transaction from Rx FIFO on the CAN bus.*
- void **FLEXCAN\_TransferCreateHandle** (CAN\_Type \*base, flexcan\_handle\_t \*handle, flexcan\_transfer\_callback\_t callback, void \*userData)  
*Initializes the FlexCAN handle.*
- status\_t **FLEXCAN\_TransferSendNonBlocking** (CAN\_Type \*base, flexcan\_handle\_t \*handle, flexcan\_mb\_transfer\_t \*xfer)  
*Sends a message using IRQ.*
- status\_t **FLEXCAN\_TransferReceiveNonBlocking** (CAN\_Type \*base, flexcan\_handle\_t \*handle, flexcan\_mb\_transfer\_t \*xfer)  
*Receives a message using IRQ.*
- status\_t **FLEXCAN\_TransferReceiveFifoNonBlocking** (CAN\_Type \*base, flexcan\_handle\_t \*handle, flexcan\_fifo\_transfer\_t \*xfer)  
*Receives a message from Rx FIFO using IRQ.*
- void **FLEXCAN\_TransferAbortSend** (CAN\_Type \*base, flexcan\_handle\_t \*handle, uint8\_t mbIdx)  
*Aborts the interrupt driven message send process.*
- void **FLEXCAN\_TransferAbortReceive** (CAN\_Type \*base, flexcan\_handle\_t \*handle, uint8\_t mbIdx)  
*Aborts the interrupt driven message receive process.*
- void **FLEXCAN\_TransferAbortReceiveFifo** (CAN\_Type \*base, flexcan\_handle\_t \*handle)  
*Aborts the interrupt driven message receive from Rx FIFO process.*
- void **FLEXCAN\_TransferHandleIRQ** (CAN\_Type \*base, flexcan\_handle\_t \*handle)  
*FlexCAN IRQ handle function.*

### 16.2.3 Data Structure Documentation

#### 16.2.3.1 struct flexcan\_frame\_t

##### 16.2.3.1.0.40 Field Documentation

16.2.3.1.0.40.1 uint32\_t flexcan\_frame\_t::timestamp

16.2.3.1.0.40.2 uint32\_t flexcan\_frame\_t::length

16.2.3.1.0.40.3 uint32\_t flexcan\_frame\_t::type

16.2.3.1.0.40.4 uint32\_t flexcan\_frame\_t::format

16.2.3.1.0.40.5 uint32\_t flexcan\_frame\_t::\_\_pad0\_\_

16.2.3.1.0.40.6 uint32\_t flexcan\_frame\_t::idhit

16.2.3.1.0.40.7 uint32\_t flexcan\_frame\_t::id

16.2.3.1.0.40.8 uint32\_t flexcan\_frame\_t::dataWord0

16.2.3.1.0.40.9 uint32\_t flexcan\_frame\_t::dataWord1

16.2.3.1.0.40.10 uint8\_t flexcan\_frame\_t::dataByte3

16.2.3.1.0.40.11 uint8\_t flexcan\_frame\_t::dataByte2

16.2.3.1.0.40.12 uint8\_t flexcan\_frame\_t::dataByte1

16.2.3.1.0.40.13 uint8\_t flexcan\_frame\_t::dataByte0

16.2.3.1.0.40.14 uint8\_t flexcan\_frame\_t::dataByte7

16.2.3.1.0.40.15 uint8\_t flexcan\_frame\_t::dataByte6

16.2.3.1.0.40.16 uint8\_t flexcan\_frame\_t::dataByte5

16.2.3.1.0.40.17 uint8\_t flexcan\_frame\_t::dataByte4

#### 16.2.3.2 struct flexcan\_config\_t

##### Data Fields

- uint32\_t baudRate  
*FlexCAN baud rate in bps.*
- flexcan\_clock\_source\_t clkSrc  
*Clock source for FlexCAN Protocol Engine.*
- uint8\_t maxMbNum  
*The maximum number of Message Buffers used by user.*

- bool `enableLoopBack`  
*Enable or Disable Loop Back Self Test Mode.*
- bool `enableSelfWakeup`  
*Enable or Disable Self Wakeup Mode.*
- bool `enableIndividMask`  
*Enable or Disable Rx Individual Mask.*

#### 16.2.3.2.0.41 Field Documentation

16.2.3.2.0.41.1 `uint32_t flexcan_config_t::baudRate`

16.2.3.2.0.41.2 `flexcan_clock_source_t flexcan_config_t::clkSrc`

16.2.3.2.0.41.3 `uint8_t flexcan_config_t::maxMbNum`

16.2.3.2.0.41.4 `bool flexcan_config_t::enableLoopBack`

16.2.3.2.0.41.5 `bool flexcan_config_t::enableSelfWakeup`

16.2.3.2.0.41.6 `bool flexcan_config_t::enableIndividMask`

#### 16.2.3.3 struct `flexcan_timing_config_t`

##### Data Fields

- `uint8_t preDivider`  
*Clock Pre-scaler Division Factor.*
- `uint8_t rJumpwidth`  
*Re-sync Jump Width.*
- `uint8_t phaseSeg1`  
*Phase Segment 1.*
- `uint8_t phaseSeg2`  
*Phase Segment 2.*
- `uint8_t propSeg`  
*Propagation Segment.*

## FlexCAN Driver

### 16.2.3.3.0.42 Field Documentation

16.2.3.3.0.42.1 `uint8_t flexcan_timing_config_t::preDivider`

16.2.3.3.0.42.2 `uint8_t flexcan_timing_config_t::rJumpwidth`

16.2.3.3.0.42.3 `uint8_t flexcan_timing_config_t::phaseSeg1`

16.2.3.3.0.42.4 `uint8_t flexcan_timing_config_t::phaseSeg2`

16.2.3.3.0.42.5 `uint8_t flexcan_timing_config_t::propSeg`

### 16.2.3.4 `struct flexcan_rx_mb_config_t`

This structure is used as the parameter of [FLEXCAN\\_SetRxMbConfig\(\)](#) function. The [FLEXCAN\\_SetRxMbConfig\(\)](#) function is used to configure FlexCAN Receive Message Buffer. The function abort previous receiving process, clean the Message Buffer and activate the Rx Message Buffer using given Message Buffer setting.

#### Data Fields

- `uint32_t id`  
*CAN Message Buffer Frame Identifier, should be set using [FLEXCAN\\_ID\\_EXT\(\)](#) or [FLEXCAN\\_ID\\_STD\(\)](#) macro.*
- `flexcan_frame_format_t format`  
*CAN Frame Identifier format(Standard or Extend).*
- `flexcan_frame_type_t type`  
*CAN Frame Type(Data or Remote).*

### 16.2.3.4.0.43 Field Documentation

16.2.3.4.0.43.1 `uint32_t flexcan_rx_mb_config_t::id`

16.2.3.4.0.43.2 `flexcan_frame_format_t flexcan_rx_mb_config_t::format`

16.2.3.4.0.43.3 `flexcan_frame_type_t flexcan_rx_mb_config_t::type`

### 16.2.3.5 `struct flexcan_rx_fifo_config_t`

#### Data Fields

- `uint32_t * idFilterTable`  
*Pointer to the FlexCAN Rx FIFO identifier filter table.*
- `uint8_t idFilterNum`  
*The quantity of filter elements.*
- `flexcan_rx_fifo_filter_type_t idFilterType`  
*The FlexCAN Rx FIFO Filter type.*
- `flexcan_rx_fifo_priority_t priority`  
*The FlexCAN Rx FIFO receive priority.*

**16.2.3.5.0.44 Field Documentation****16.2.3.5.0.44.1 uint32\_t\* flexcan\_rx\_fifo\_config\_t::idFilterTable****16.2.3.5.0.44.2 uint8\_t flexcan\_rx\_fifo\_config\_t::idFilterNum****16.2.3.5.0.44.3 flexcan\_rx\_fifo\_filter\_type\_t flexcan\_rx\_fifo\_config\_t::idFilterType****16.2.3.5.0.44.4 flexcan\_rx\_fifo\_priority\_t flexcan\_rx\_fifo\_config\_t::priority****16.2.3.6 struct flexcan\_mb\_transfer\_t****Data Fields**

- **flexcan\_frame\_t \* frame**  
*The buffer of CAN Message to be transfer.*
- **uint8\_t mbIdx**  
*The index of Message buffer used to transfer Message.*

**16.2.3.6.0.45 Field Documentation****16.2.3.6.0.45.1 flexcan\_frame\_t\* flexcan\_mb\_transfer\_t::frame****16.2.3.6.0.45.2 uint8\_t flexcan\_mb\_transfer\_t::mbIdx****16.2.3.7 struct flexcan\_fifo\_transfer\_t****Data Fields**

- **flexcan\_frame\_t \* frame**  
*The buffer of CAN Message to be received from Rx FIFO.*

**16.2.3.7.0.46 Field Documentation****16.2.3.7.0.46.1 flexcan\_frame\_t\* flexcan\_fifo\_transfer\_t::frame****16.2.3.8 struct \_flexcan\_handle**

FlexCAN handle structure definition.

**Data Fields**

- **flexcan\_transfer\_callback\_t callback**  
*Callback function.*
- **void \* userData**  
*FlexCAN callback function parameter.*
- **flexcan\_frame\_t \*volatile mbFrameBuf [CAN\_WORD1\_COUNT]**  
*The buffer for received data from Message Buffers.*
- **flexcan\_frame\_t \*volatile rxFifoFrameBuf**  
*The buffer for received data from Rx FIFO.*

## FlexCAN Driver

- volatile uint8\_t **mbState** [CAN\_WORD1\_COUNT]  
*Message Buffer transfer state.*
- volatile uint8\_t **rxFifoState**  
*Rx FIFO transfer state.*

### 16.2.3.8.0.47 Field Documentation

16.2.3.8.0.47.1 **flexcan\_transfer\_callback\_t flexcan\_handle\_t::callback**

16.2.3.8.0.47.2 **void\* flexcan\_handle\_t::userData**

16.2.3.8.0.47.3 **flexcan\_frame\_t\* volatile flexcan\_handle\_t::mbFrameBuf[CAN\_WORD1\_COUNT]**

16.2.3.8.0.47.4 **flexcan\_frame\_t\* volatile flexcan\_handle\_t::rx\_fifoFrameBuf**

16.2.3.8.0.47.5 **volatile uint8\_t flexcan\_handle\_t::mbState[CAN\_WORD1\_COUNT]**

16.2.3.8.0.47.6 **volatile uint8\_t flexcan\_handle\_t::rxFifoState**

### 16.2.4 Macro Definition Documentation

16.2.4.1 **#define FLEXCAN\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 0))**

16.2.4.2 **#define FLEXCAN\_ID\_STD( id ) (((uint32\_t)((uint32\_t)(id)) << CAN\_ID\_STD\_SHIFT) & CAN\_ID\_STD\_MASK)**

Standard Frame ID helper macro.

16.2.4.3 **#define FLEXCAN\_ID\_EXT( id )**

**Value:**

```
((uint32_t)((uint32_t)(id)) << CAN_ID_EXT_SHIFT) & \  
 (CAN_ID_EXT_MASK | CAN_ID_STD_MASK))
```

16.2.4.4 **#define FLEXCAN\_RX\_MB\_STD\_MASK( id, rtr, ide )**

**Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \  
 FLEXCAN_ID_STD(id))
```

Standard Rx Message Buffer Mask helper macro.

**16.2.4.5 #define FLEXCAN\_RX\_MB\_EXT\_MASK( *id*, *rtr*, *ide* )****Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
    FLEXCAN_ID_EXT(id))
```

**16.2.4.6 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_A( *id*, *rtr*, *ide* )****Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
    (FLEXCAN_ID_STD(id) << 1))
```

Standard Rx FIFO Mask helper macro Type A helper macro.

**16.2.4.7 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_B\_HIGH( *id*, *rtr*, *ide* )****Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \
    ((uint32_t)(id) & 0x7FF) << 19))
```

**16.2.4.8 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_B\_LOW( *id*, *rtr*, *ide* )****Value:**

```
((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \
    ((uint32_t)(id) & 0x7FF) << 3))
```

**16.2.4.9 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_HIGH( *id* ) (((uint32\_t)(id) & 0x7F8) << 21)****16.2.4.10 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_MID\_HIGH( *id* ) (((uint32\_t)(id) & 0x7F8) << 13)****16.2.4.11 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_MID\_LOW( *id* ) (((uint32\_t)(id) & 0x7F8) << 5)****16.2.4.12 #define FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_LOW( *id* ) (((uint32\_t)(id) & 0x7F8) >> 3)****16.2.4.13 #define FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_A( *id*, *rtr*, *ide* )****Value:**

## FlexCAN Driver

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \  
  (FLEXCAN_ID_EXT(id) << 1))
```

### 16.2.4.14 #define FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_B\_HIGH( id, rtr, ide )

**Value:**

```
((uint32_t)((uint32_t)(rtr) << 31) | (uint32_t)((uint32_t)(ide) << 30)) | \  
  (FLEXCAN_ID_EXT(id) & 0x1FFF8000) << 1))
```

### 16.2.4.15 #define FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_B\_LOW( id, rtr, ide )

**Value:**

```
((uint32_t)((uint32_t)(rtr) << 15) | (uint32_t)((uint32_t)(ide) << 14)) | \  
  (FLEXCAN_ID_EXT(id) & 0x1FFF8000) >> \  
  15))
```

### 16.2.4.16 #define FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_HIGH( id ) ((FLEXCAN\_ID\_EXT(id) & 0x1FE00000) << 3)

### 16.2.4.17 #define FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_MID\_HIGH( id )

**Value:**

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> \  
  5))
```

### 16.2.4.18 #define FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_MID\_LOW( id )

**Value:**

```
((FLEXCAN_ID_EXT(id) & 0x1FE00000) >> \  
  13))
```

### 16.2.4.19 #define FLEXCAN\_RX\_FIFO\_EXT\_MASK\_TYPE\_C\_LOW( id ) ((FLEXCAN\_ID\_EXT(id) & 0x1FE00000) >> 21)

### 16.2.4.20 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_A( id, rtr, ide ) FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_A(id, rtr, ide)

Standard Rx FIFO Filter helper macro Type A helper macro.

**16.2.4.21 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_B\_HIGH( *id*, *rtr*, *ide* )****Value:**

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_HIGH(           \
    id, rtr, ide)
```

**16.2.4.22 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_B\_LOW( *id*, *rtr*, *ide* )****Value:**

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_B_LOW(           \
    id, rtr, ide)
```

**16.2.4.23 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_HIGH( *id* )****Value:**

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_HIGH(           \
    id)
```

**16.2.4.24 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_MID\_HIGH( *id* )****Value:**

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_HIGH(       \
    id)
```

**16.2.4.25 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_MID\_LOW( *id* )****Value:**

```
FLEXCAN_RX_FIFO_STD_MASK_TYPE_C_MID_LOW(         \
    id)
```

**16.2.4.26 #define FLEXCAN\_RX\_FIFO\_STD\_FILTER\_TYPE\_C\_LOW( *id* )  
 ) FLEXCAN\_RX\_FIFO\_STD\_MASK\_TYPE\_C\_LOW(id)**

\

## FlexCAN Driver

- 16.2.4.27 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_A( id, rtr, ide ) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_A(id, rtr, ide)`
- 16.2.4.28 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_HIGH( id, rtr, ide )`

**Value:**

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_HIGH(           \
    id, rtr, ide)
```

- 16.2.4.29 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_B_LOW( id, rtr, ide )`

**Value:**

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_B_LOW(           \
    id, rtr, ide)
```

- 16.2.4.30 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_HIGH( id ) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_HIGH(id)`

\

- 16.2.4.31 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_HIGH( id )`

**Value:**

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_HIGH(       \
    id)
```

- 16.2.4.32 `#define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_MID_LOW( id )`

**Value:**

```
FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_MID_LOW(        \
    id)
```

```
16.2.4.33 #define FLEXCAN_RX_FIFO_EXT_FILTER_TYPE_C_LOW( id
    ) FLEXCAN_RX_FIFO_EXT_MASK_TYPE_C_LOW(id)
```

## 16.2.5 Typedef Documentation

**16.2.5.1 `typedef void(* flexcan_transfer_callback_t)(CAN_Type *base, flexcan_handle_t *handle, status_t status, uint32_t result, void *userData)`**

The FlexCAN transfer callback returns a value from the underlying layer. If the status equals to kStatus\_FLEXCAN\_ErrorStatus, the result parameter is the Content of FlexCAN status register which can be used to get the working status(or error status) of FlexCAN module. If the status equals to other FlexCAN Message Buffer transfer status, the result is the index of Message Buffer that generate transfer event. If the status equals to other FlexCAN Message Buffer transfer status, the result is meaningless and should be Ignored.

## 16.2.6 Enumeration Type Documentation

**16.2.6.1 `enum _flexcan_status`**

Enumerator

- kStatus\_FLEXCAN\_TxBusy* Tx Message Buffer is Busy.
- kStatus\_FLEXCAN\_TxIdle* Tx Message Buffer is Idle.
- kStatus\_FLEXCAN\_TxSwitchToRx* Remote Message is send out and Message buffer changed to Receive one.
- kStatus\_FLEXCAN\_RxBusy* Rx Message Buffer is Busy.
- kStatus\_FLEXCAN\_RxIdle* Rx Message Buffer is Idle.
- kStatus\_FLEXCAN\_RxOverflow* Rx Message Buffer is Overflowed.
- kStatus\_FLEXCAN\_RxFifoBusy* Rx Message FIFO is Busy.
- kStatus\_FLEXCAN\_RxFifoIdle* Rx Message FIFO is Idle.
- kStatus\_FLEXCAN\_RxFifoOverflow* Rx Message FIFO is overflowed.
- kStatus\_FLEXCAN\_RxFifoWarning* Rx Message FIFO is almost overflowed.
- kStatus\_FLEXCAN\_ErrorStatus* FlexCAN Module Error and Status.
- kStatus\_FLEXCAN\_UnHandled* UnHadled Interrupt asserted.

**16.2.6.2 `enum flexcan_frame_format_t`**

Enumerator

- kFLEXCAN\_FrameFormatStandard* Standard frame format attribute.
- kFLEXCAN\_FrameFormatExtend* Extend frame format attribute.

## FlexCAN Driver

### 16.2.6.3 enum flexcan\_frame\_type\_t

Enumerator

*kFLEXCAN\_FrameTypeData* Data frame type attribute.

*kFLEXCAN\_FrameTypeRemote* Remote frame type attribute.

### 16.2.6.4 enum flexcan\_clock\_source\_t

Enumerator

*kFLEXCAN\_ClkSrcOsc* FlexCAN Protocol Engine clock from Oscillator.

*kFLEXCAN\_ClkSrcPeri* FlexCAN Protocol Engine clock from Peripheral Clock.

### 16.2.6.5 enum flexcan\_rx\_fifo\_filter\_type\_t

Enumerator

*kFLEXCAN\_RxFifoFilterTypeA* One full ID (standard and extended) per ID Filter element.

*kFLEXCAN\_RxFifoFilterTypeB* Two full standard IDs or two partial 14-bit ID slices per ID Filter Table element.

*kFLEXCAN\_RxFifoFilterTypeC* Four partial 8-bit Standard or extended ID slices per ID Filter Table element.

*kFLEXCAN\_RxFifoFilterTypeD* All frames rejected.

### 16.2.6.6 enum flexcan\_rx\_fifo\_priority\_t

The matching process starts from the Rx MB(or Rx FIFO) with higher priority. If no MB(or Rx FIFO filter) is satisfied, the matching process goes on with the Rx FIFO(or Rx MB) with lower priority.

Enumerator

*kFLEXCAN\_RxFifoPrioLow* Matching process start from Rx Message Buffer first.

*kFLEXCAN\_RxFifoPrioHigh* Matching process start from Rx FIFO first.

### 16.2.6.7 enum \_flexcan\_interrupt\_enable

This structure contains the settings for all of the FlexCAN Module interrupt configurations. Note: FlexCAN Message Buffers and Rx FIFO have their own interrupts.

Enumerator

*kFLEXCAN\_BusOffInterruptEnable* Bus Off interrupt.

***kFLEXCAN\_ErrorInterruptEnable*** Error interrupt.  
***kFLEXCAN\_RxWarningInterruptEnable*** Rx Warning interrupt.  
***kFLEXCAN\_TxWarningInterruptEnable*** Tx Warning interrupt.  
***kFLEXCAN\_WakeUpInterruptEnable*** Wake Up interrupt.

#### 16.2.6.8 enum \_flexcan\_flags

This provides constants for the FlexCAN status flags for use in the FlexCAN functions. Note: The CPU read action clears FLEXCAN\_ErrorFlag, therefore user need to read FLEXCAN\_ErrorFlag and distinguish which error is occur using [\\_flexcan\\_error\\_flags](#) enumerations.

Enumerator

***kFLEXCAN\_SynchFlag*** CAN Synchronization Status.  
***kFLEXCAN\_TxWarningIntFlag*** Tx Warning Interrupt Flag.  
***kFLEXCAN\_RxWarningIntFlag*** Rx Warning Interrupt Flag.  
***kFLEXCAN\_TxErrorWarningFlag*** Tx Error Warning Status.  
***kFLEXCAN\_RxErrorWarningFlag*** Rx Error Warning Status.  
***kFLEXCAN\_IdleFlag*** CAN IDLE Status Flag.  
***kFLEXCAN\_FaultConfinementFlag*** Fault Confinement State Flag.  
***kFLEXCAN\_TransmittingFlag*** FlexCAN In Transmission Status.  
***kFLEXCAN\_ReceivingFlag*** FlexCAN In Reception Status.  
***kFLEXCAN\_BusOffIntFlag*** Bus Off Interrupt Flag.  
***kFLEXCAN\_ErrorIntFlag*** Error Interrupt Flag.  
***kFLEXCAN\_WakeUpIntFlag*** Wake-Up Interrupt Flag.  
***kFLEXCAN\_ErrorFlag*** All FlexCAN Error Status.

#### 16.2.6.9 enum \_flexcan\_error\_flags

The FlexCAN Error Status enumerations is used to report current error of the FlexCAN bus. This enumerations should be used with KFLEXCAN\_ErrorFlag in [\\_flexcan\\_flags](#) enumerations to determine which error is generated.

Enumerator

***kFLEXCAN\_StuffingError*** Stuffing Error.  
***kFLEXCAN\_FormError*** Form Error.  
***kFLEXCAN\_CrcError*** Cyclic Redundancy Check Error.  
***kFLEXCAN\_AckError*** Received no ACK on transmission.  
***kFLEXCAN\_Bit0Error*** Unable to send dominant bit.  
***kFLEXCAN\_Bit1Error*** Unable to send recessive bit.

## FlexCAN Driver

### 16.2.6.10 enum \_flexcan\_rx\_fifo\_flags

The FlexCAN Rx FIFO Status enumerations are used to determine the status of the Rx FIFO. Because Rx FIFO occupy the MB0 ~ MB7 (Rx Fifo filter also occupies more Message Buffer space), Rx FIFO status flags are mapped to the corresponding Message Buffer status flags.

Enumerator

- kFLEXCAN\_RxFifoOverflowFlag* Rx FIFO overflow flag.
- kFLEXCAN\_RxFifoWarningFlag* Rx FIFO almost full flag.
- kFLEXCAN\_RxFifoFrameAvlFlag* Frames available in Rx FIFO flag.

### 16.2.7 Function Documentation

#### 16.2.7.1 void FLEXCAN\_Init ( CAN\_Type \* *base*, const flexcan\_config\_t \* *config*, uint32\_t *sourceClock\_Hz* )

This function initializes the FlexCAN module with user-defined settings. This example shows how to set up the *flexcan\_config\_t* parameters and how to call the FLEXCAN\_Init function by passing in these parameters.

```
*     flexcan_config_t flexcanConfig;
*     flexcanConfig.clkSrc          = kFLEXCAN_ClkSrcOsc;
*     flexcanConfig.baudRate        = 125000U;
*     flexcanConfig.maxMbNum        = 16;
*     flexcanConfig.enableLoopBack  = false;
*     flexcanConfig.enableSelfWakeup = false;
*     flexcanConfig.enableIndividMask = false;
*     flexcanConfig.enableDoze      = false;
*     FLEXCAN_Init(CAN0, &flexcanConfig, 8000000UL);
*
```

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>config</i>	Pointer to the user-defined configuration structure.
<i>sourceClock_- Hz</i>	FlexCAN Protocol Engine clock source frequency in Hz.

#### 16.2.7.2 void FLEXCAN\_Deinit ( CAN\_Type \* *base* )

This function disables the FlexCAN module clock and sets all register values to the reset value.

Parameters

<i>base</i>	FlexCAN peripheral base address.
-------------	----------------------------------

#### 16.2.7.3 void FLEXCAN\_GetDefaultConfig ( flexcan\_config\_t \* *config* )

This function initializes the FlexCAN configuration structure to default values. The default values are as follows. *flexcanConfig->clkSrc* = KFLEXCAN\_ClkSrcOsc; *flexcanConfig->baudRate* = 125000U; *flexcanConfig->maxMbNum* = 16; *flexcanConfig->enableLoopBack* = false; *flexcanConfig->enableSelfWakeup* = false; *flexcanConfig->enableIndividMask* = false; *flexcanConfig->enableDoze* = false;

Parameters

<i>config</i>	Pointer to the FlexCAN configuration structure.
---------------	---

#### 16.2.7.4 void FLEXCAN\_SetTimingConfig ( CAN\_Type \* *base*, const flexcan\_timing\_config\_t \* *config* )

This function gives user settings to CAN bus timing characteristic. The function is for an experienced user. For less experienced users, call the [FLEXCAN\\_Init\(\)](#) and fill the baud rate field with a desired value. This provides the default timing characteristics to the module.

Note that calling [FLEXCAN\\_SetTimingConfig\(\)](#) overrides the baud rate set in [FLEXCAN\\_Init\(\)](#).

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>config</i>	Pointer to the timing configuration structure.

#### 16.2.7.5 void FLEXCAN\_SetRxMbGlobalMask ( CAN\_Type \* *base*, uint32\_t *mask* )

This function sets the global mask for the FlexCAN message buffer in a matching process. The configuration is only effective when the Rx individual mask is disabled in the [FLEXCAN\\_Init\(\)](#).

Parameters

<i>base</i>	FlexCAN peripheral base address.
-------------	----------------------------------

## FlexCAN Driver

<i>mask</i>	Rx Message Buffer Global Mask value.
-------------	--------------------------------------

### 16.2.7.6 void FLEXCAN\_SetRxFifoGlobalMask ( CAN\_Type \* *base*, uint32\_t *mask* )

This function sets the global mask for FlexCAN FIFO in a matching process.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	Rx Fifo Global Mask value.

### 16.2.7.7 void FLEXCAN\_SetRxIndividualMask ( CAN\_Type \* *base*, uint8\_t *maskIdx*, uint32\_t *mask* )

This function sets the individual mask for the FlexCAN matching process. The configuration is only effective when the Rx individual mask is enabled in the [FLEXCAN\\_Init\(\)](#). If the Rx FIFO is disabled, the individual mask is applied to the corresponding Message Buffer. If the Rx FIFO is enabled, the individual mask for Rx FIFO occupied Message Buffer is applied to the Rx Filter with the same index. Note that only the first 32 individual masks can be used as the Rx FIFO filter mask.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>maskIdx</i>	The Index of individual Mask.
<i>mask</i>	Rx Individual Mask value.

### 16.2.7.8 void FLEXCAN\_SetTxMbConfig ( CAN\_Type \* *base*, uint8\_t *mbIdx*, bool *enable* )

This function aborts the previous transmission, cleans the Message Buffer, and configures it as a Transmit Message Buffer.

Parameters

<i>base</i>	FlexCAN peripheral base address.
-------------	----------------------------------

<i>mbIdx</i>	The Message Buffer index.
<i>enable</i>	Enable/disable Tx Message Buffer. <ul style="list-style-type: none"> <li>• true: Enable Tx Message Buffer.</li> <li>• false: Disable Tx Message Buffer.</li> </ul>

#### 16.2.7.9 void FLEXCAN\_SetRxMbConfig ( CAN\_Type \* *base*, uint8\_t *mbIdx*, const flexcan\_rx\_mb\_config\_t \* *config*, bool *enable* )

This function cleans a FlexCAN build-in Message Buffer and configures it as a Receive Message Buffer.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mbIdx</i>	The Message Buffer index.
<i>config</i>	Pointer to the FlexCAN Message Buffer configuration structure.
<i>enable</i>	Enable/disable Rx Message Buffer. <ul style="list-style-type: none"> <li>• true: Enable Rx Message Buffer.</li> <li>• false: Disable Rx Message Buffer.</li> </ul>

#### 16.2.7.10 void FLEXCAN\_SetRxFifoConfig ( CAN\_Type \* *base*, const flexcan\_rx\_fifo\_config\_t \* *config*, bool *enable* )

This function configures the Rx FIFO with given Rx FIFO configuration.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>config</i>	Pointer to the FlexCAN Rx FIFO configuration structure.
<i>enable</i>	Enable/disable Rx FIFO. <ul style="list-style-type: none"> <li>• true: Enable Rx FIFO.</li> <li>• false: Disable Rx FIFO.</li> </ul>

#### 16.2.7.11 static uint32\_t FLEXCAN\_GetStatusFlags ( CAN\_Type \* *base* ) [inline], [static]

This function gets all FlexCAN status flags. The flags are returned as the logical OR value of the enumerators [\\_flexcan\\_flags](#). To check the specific status, compare the return value with enumerators in [\\_flexcan-](#)

flags.

Parameters

<i>base</i>	FlexCAN peripheral base address.
-------------	----------------------------------

Returns

FlexCAN status flags which are ORed by the enumerators in the `_flexcan_flags`.

#### 16.2.7.12 static void FLEXCAN\_ClearStatusFlags ( CAN\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function clears the FlexCAN status flags with a provided mask. An automatically cleared flag can't be cleared by this function.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The status flags to be cleared, it is logical OR value of <code>_flexcan_flags</code> .

#### 16.2.7.13 static void FLEXCAN\_GetBusErrCount ( CAN\_Type \* *base*, uint8\_t \* *txErrBuf*, uint8\_t \* *rxErrBuf* ) [inline], [static]

This function gets the FlexCAN Bus Error Counter value for both Tx and Rx direction. These values may be needed in the upper layer error handling.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>txErrBuf</i>	Buffer to store Tx Error Counter value.
<i>rxErrBuf</i>	Buffer to store Rx Error Counter value.

#### 16.2.7.14 static uint32\_t FLEXCAN\_GetMbStatusFlags ( CAN\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function gets the interrupt flags of a given Message Buffers.

## FlexCAN Driver

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

Returns

The status of given Message Buffers.

### 16.2.7.15 static void FLEXCAN\_ClearMbStatusFlags ( CAN\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function clears the interrupt flags of a given Message Buffers.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

### 16.2.7.16 static void FLEXCAN\_EnableInterrupts ( CAN\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function enables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see [\\_flexcan\\_interrupt\\_enable](#).

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of <a href="#">_flexcan_interrupt_enable</a> .

### 16.2.7.17 static void FLEXCAN\_DisableInterrupts ( CAN\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function disables the FlexCAN interrupts according to the provided mask. The mask is a logical OR of enumeration members, see [\\_flexcan\\_interrupt\\_enable](#).

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The interrupts to disable. Logical OR of <a href="#">_flexcan_interrupt_enable</a> .

#### 16.2.7.18 static void FLEXCAN\_EnableMbInterrupts ( CAN\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function enables the interrupts of given Message Buffers.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

#### 16.2.7.19 static void FLEXCAN\_DisableMbInterrupts ( CAN\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function disables the interrupts of given Message Buffers.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mask</i>	The ORed FlexCAN Message Buffer mask.

#### 16.2.7.20 static void FLEXCAN\_Enable ( CAN\_Type \* *base*, bool *enable* ) [inline], [static]

This function enables or disables the FlexCAN module.

Parameters

<i>base</i>	FlexCAN base pointer.
<i>enable</i>	true to enable, false to disable.

#### 16.2.7.21 status\_t FLEXCAN\_WriteTxMb ( CAN\_Type \* *base*, uint8\_t *mbIdx*, const flexcan\_frame\_t \* *txFrame* )

This function writes a CAN Message to the specified Transmit Message Buffer and changes the Message Buffer state to start CAN Message transmit. After that the function returns immediately.

## FlexCAN Driver

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mbIdx</i>	The FlexCAN Message Buffer index.
<i>txFrame</i>	Pointer to CAN message frame to be sent.

Return values

<i>kStatus_Success</i>	- Write Tx Message Buffer Successfully.
<i>kStatus_Fail</i>	- Tx Message Buffer is currently in use.

### 16.2.7.22 status\_t FLEXCAN\_ReadRxMb ( CAN\_Type \* *base*, uint8\_t *mbIdx*, flexcan\_frame\_t \* *rxFrame* )

This function reads a CAN message from a specified Receive Message Buffer. The function fills a receive CAN message frame structure with just received data and activates the Message Buffer again. The function returns immediately.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>mbIdx</i>	The FlexCAN Message Buffer index.
<i>rxFrame</i>	Pointer to CAN message frame structure for reception.

Return values

<i>kStatus_Success</i>	- Rx Message Buffer is full and has been read successfully.
<i>kStatus_FLEXCAN_Overflow</i>	- Rx Message Buffer is already overflowed and has been read successfully.
<i>kStatus_Fail</i>	- Rx Message Buffer is empty.

### 16.2.7.23 status\_t FLEXCAN\_ReadRxFifo ( CAN\_Type \* *base*, flexcan\_frame\_t \* *rxFrame* )

This function reads a CAN message from the FlexCAN build-in Rx FIFO.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>rxFrame</i>	Pointer to CAN message frame structure for reception.

Return values

<i>kStatus_Success</i>	- Read Message from Rx FIFO successfully.
<i>kStatus_Fail</i>	- Rx FIFO is not enabled.

#### 16.2.7.24 status\_t FLEXCAN\_TransferSendBlocking ( CAN\_Type \* *base*, uint8\_t *mbIdx*, flexcan\_frame\_t \* *txFrame* )

Note that a transfer handle does not need to be created before calling this API.

Parameters

<i>base</i>	FlexCAN peripheral base pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.
<i>txFrame</i>	Pointer to CAN message frame to be sent.

Return values

<i>kStatus_Success</i>	- Write Tx Message Buffer Successfully.
<i>kStatus_Fail</i>	- Tx Message Buffer is currently in use.

#### 16.2.7.25 status\_t FLEXCAN\_TransferReceiveBlocking ( CAN\_Type \* *base*, uint8\_t *mbIdx*, flexcan\_frame\_t \* *rxFrame* )

Note that a transfer handle does not need to be created before calling this API.

Parameters

<i>base</i>	FlexCAN peripheral base pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.

## FlexCAN Driver

<i>rxFrame</i>	Pointer to CAN message frame structure for reception.
----------------	---

Return values

<i>kStatus_Success</i>	- Rx Message Buffer is full and has been read successfully.
<i>kStatus_FLEXCAN_RxOverflow</i>	- Rx Message Buffer is already overflowed and has been read successfully.
<i>kStatus_Fail</i>	- Rx Message Buffer is empty.

### 16.2.7.26 **status\_t FLEXCAN\_TransferReceiveFifoBlocking ( CAN\_Type \* *base*, flexcan\_frame\_t \* *rxFrame* )**

Note that a transfer handle does not need to be created before calling this API.

Parameters

<i>base</i>	FlexCAN peripheral base pointer.
<i>rxFrame</i>	Pointer to CAN message frame structure for reception.

Return values

<i>kStatus_Success</i>	- Read Message from Rx FIFO successfully.
<i>kStatus_Fail</i>	- Rx FIFO is not enabled.

### 16.2.7.27 **void FLEXCAN\_TransferCreateHandle ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle*, flexcan\_transfer\_callback\_t *callback*, void \* *userData* )**

This function initializes the FlexCAN handle, which can be used for other FlexCAN transactional APIs. Usually, for a specified FlexCAN instance, call this API once to get the initialized handle.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>callback</i>	The callback function.

<i>userData</i>	The parameter of the callback function.
-----------------	---

### 16.2.7.28 status\_t FLEXCAN\_TransferSendNonBlocking ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle*, flexcan\_mb\_transfer\_t \* *xfer* )

This function sends a message using IRQ. This is a non-blocking function, which returns right away. When messages have been sent out, the send callback function is called.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>xfer</i>	FlexCAN Message Buffer transfer structure. See the <a href="#">flexcan_mb_transfer_t</a> .

Return values

<i>kStatus_Success</i>	Start Tx Message Buffer sending process successfully.
<i>kStatus_Fail</i>	Write Tx Message Buffer failed.
<i>kStatus_FLEXCAN_Tx-Busy</i>	Tx Message Buffer is in use.

### 16.2.7.29 status\_t FLEXCAN\_TransferReceiveNonBlocking ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle*, flexcan\_mb\_transfer\_t \* *xfer* )

This function receives a message using IRQ. This is non-blocking function, which returns right away. When the message has been received, the receive callback function is called.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>xfer</i>	FlexCAN Message Buffer transfer structure. See the <a href="#">flexcan_mb_transfer_t</a> .

Return values

## FlexCAN Driver

<i>kStatus_Success</i>	- Start Rx Message Buffer receiving process successfully.
<i>kStatus_FLEXCAN_Rx-Busy</i>	- Rx Message Buffer is in use.

### 16.2.7.30 **status\_t FLEXCAN\_TransferReceiveFifoNonBlocking ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle*, flexcan\_fifo\_transfer\_t \* *xfer* )**

This function receives a message using IRQ. This is a non-blocking function, which returns right away. When all messages have been received, the receive callback function is called.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>xfer</i>	FlexCAN Rx FIFO transfer structure. See the <a href="#">flexcan_fifo_transfer_t</a> .

Return values

<i>kStatus_Success</i>	- Start Rx FIFO receiving process successfully.
<i>kStatus_FLEXCAN_Rx-FifoBusy</i>	- Rx FIFO is currently in use.

### 16.2.7.31 **void FLEXCAN\_TransferAbortSend ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle*, uint8\_t *mbIdx* )**

This function aborts the interrupt driven message send process.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.

### 16.2.7.32 **void FLEXCAN\_TransferAbortReceive ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle*, uint8\_t *mbIdx* )**

This function aborts the interrupt driven message receive process.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.
<i>mbIdx</i>	The FlexCAN Message Buffer index.

#### **16.2.7.33 void FLEXCAN\_TransferAbortReceiveFifo ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle* )**

This function aborts the interrupt driven message receive from Rx FIFO process.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.

#### **16.2.7.34 void FLEXCAN\_TransferHandleIRQ ( CAN\_Type \* *base*, flexcan\_handle\_t \* *handle* )**

This function handles the FlexCAN Error, the Message Buffer, and the Rx FIFO IRQ request.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	FlexCAN handle pointer.

## FlexCAN eDMA Driver

### 16.3 FlexCAN eDMA Driver

#### 16.3.1 Overview

#### Data Structures

- struct [flexcan\\_edma\\_handle\\_t](#)  
*FlexCAN eDMA handle.* [More...](#)

#### TypeDefs

- [typedef void\(\\* flexcan\\_edma\\_transfer\\_callback\\_t \)](#)(CAN\_Type \*base, flexcan\_edma\_handle\_t \*handle, status\_t status, void \*userData)  
*FlexCAN transfer callback function.*

#### eDMA transactional

- [void FLEXCAN\\_TransferCreateHandleEDMA](#) (CAN\_Type \*base, flexcan\_edma\_handle\_t \*handle, [flexcan\\_edma\\_transfer\\_callback\\_t](#) callback, void \*userData, [edma\\_handle\\_t](#) \*rxFifoEdmaHandle)  
*Initializes the FlexCAN handle, which is used in transactional functions.*
- [status\\_t FLEXCAN\\_TransferReceiveFifoEDMA](#) (CAN\_Type \*base, flexcan\_edma\_handle\_t \*handle, [flexcan\\_fifo\\_transfer\\_t](#) \*xfer)  
*Receives the CAN Message from the Rx FIFO using eDMA.*
- [void FLEXCAN\\_TransferAbortReceiveFifoEDMA](#) (CAN\_Type \*base, flexcan\_edma\_handle\_t \*handle)  
*Aborts the receive process which used eDMA.*

#### 16.3.2 Data Structure Documentation

##### 16.3.2.1 struct \_flexcan\_edma\_handle

#### Data Fields

- [flexcan\\_edma\\_transfer\\_callback\\_t](#) **callback**  
*Callback function.*
- [void \\*](#) **userData**  
*FlexCAN callback function parameter.*
- [edma\\_handle\\_t \\*](#) **rxFifoEdmaHandle**  
*The EDMA Rx FIFO channel used.*
- [volatile uint8\\_t](#) **rxFifoState**  
*Rx FIFO transfer state.*

#### 16.3.2.1.0.48 Field Documentation

16.3.2.1.0.48.1 `flexcan_edma_transfer_callback_t flexcan_edma_handle_t::callback`

16.3.2.1.0.48.2 `void* flexcan_edma_handle_t::userData`

16.3.2.1.0.48.3 `edma_handle_t* flexcan_edma_handle_t::rxFifoEdmaHandle`

16.3.2.1.0.48.4 `volatile uint8_t flexcan_edma_handle_t::rxFifoState`

#### 16.3.3 Typedef Documentation

16.3.3.1 `typedef void(* flexcan_edma_transfer_callback_t)(CAN_Type *base,  
flexcan_edma_handle_t *handle, status_t status, void *userData)`

#### 16.3.4 Function Documentation

16.3.4.1 `void FLEXCAN_TransferCreateHandleEDMA ( CAN_Type * base,  
flexcan_edma_handle_t * handle, flexcan_edma_transfer_callback_t callback,  
void * userData, edma_handle_t * rxFifoEdmaHandle )`

## FlexCAN eDMA Driver

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	Pointer to flexcan_edma_handle_t structure.
<i>callback</i>	The callback function.
<i>userData</i>	The parameter of the callback function.
<i>rxFifoEdmaHandle</i>	User-requested DMA handle for Rx FIFO DMA transfer.

### 16.3.4.2 status\_t FLEXCAN\_TransferReceiveFifoEDMA ( CAN\_Type \* *base*, flexcan\_edma\_handle\_t \* *handle*, flexcan\_fifo\_transfer\_t \* *xfer* )

This function receives the CAN Message using eDMA. This is a non-blocking function, which returns right away. After the CAN Message is received, the receive callback function is called.

Parameters

<i>base</i>	FlexCAN peripheral base address.
<i>handle</i>	Pointer to flexcan_edma_handle_t structure.
<i>xfer</i>	FlexCAN Rx FIFO EDMA transfer structure, see <a href="#">flexcan_fifo_transfer_t</a> .

Return values

<i>kStatus_Success</i>	if succeed, others failed.
<i>kStatus_FLEXCAN_Rx-FifoBusy</i>	Previous transfer ongoing.

### 16.3.4.3 void FLEXCAN\_TransferAbortReceiveFifoEDMA ( CAN\_Type \* *base*, flexcan\_edma\_handle\_t \* *handle* )

This function aborts the receive process which used eDMA.

Parameters

<i>base</i>	FlexCAN peripheral base address.
-------------	----------------------------------

<i>handle</i>	Pointer to flexcan_edma_handle_t structure.
---------------	---



# Chapter 17

## FTM: FlexTimer Driver

### 17.1 Overview

The MCUXpresso SDK provides a driver for the FlexTimer Module (FTM) of MCUXpresso SDK devices.

### 17.2 Function groups

The FTM driver supports the generation of PWM signals, input capture, dual edge capture, output compare, and quadrature decoder modes. The driver also supports configuring each of the FTM fault inputs.

#### 17.2.1 Initialization and deinitialization

The function [FTM\\_Init\(\)](#) initializes the FTM with specified configurations. The function [FTM\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the FTM for the requested register update mode for registers with buffers. It also sets up the FTM's fault operation mode and FTM behavior in the BDM mode.

The function [FTM\\_Deinit\(\)](#) disables the FTM counter and turns off the module clock.

#### 17.2.2 PWM Operations

The function [FTM\\_SetupPwm\(\)](#) sets up FTM channels for the PWM output. The function sets up the PWM signal properties for multiple channels. Each channel has its own duty cycle and level-mode specified. However, the same PWM period and PWM mode is applied to all channels requesting the PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle).

The function [FTM\\_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular FTM channel.

The function [FTM\\_UpdateChnlEdgeLevelSelect\(\)](#) updates the level select bits of a particular FTM channel. This can be used to disable the PWM output when making changes to the PWM signal.

#### 17.2.3 Input capture operations

The function [FTM\\_SetupInputCapture\(\)](#) sets up an FTM channel for the input capture. The user can specify the capture edge and a filter value to be used when processing the input signal.

The function [FTM\\_SetupDualEdgeCapture\(\)](#) can be used to measure the pulse width of a signal. A channel pair is used during capture with the input signal coming through a channel n. The user can specify whether

## Register Update

to use one-shot or continuous capture, the capture edge for each channel, and any filter value to be used when processing the input signal.

### 17.2.4 Output compare operations

The function [FTM\\_SetupOutputCompare\(\)](#) sets up an FTM channel for the output comparison. The user can specify the channel output on a successful comparison and a comparison value.

### 17.2.5 Quad decode

The function [FTM\\_SetupQuadDecode\(\)](#) sets up FTM channels 0 and 1 for quad decoding. The user can specify the quad decoding mode, polarity, and filter properties for each input signal.

### 17.2.6 Fault operation

The function [FTM\\_SetupFault\(\)](#) sets up the properties for each fault. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

## 17.3 Register Update

Some of the FTM registers have buffers. The driver supports various methods to update these registers with the content of the register buffer. The registers can be updated using the PWM synchronized loading or an intermediate point loading. The update mechanism for register with buffers can be specified through the following fields available in the configuration structure.

```
uint32_t pwmSyncMode;  
uint32_t reloadPoints;
```

Multiple PWM synchronization update modes can be used by providing an OR'ed list of options available in the enumeration [ftm\\_pwm\\_sync\\_method\\_t](#) to the pwmSyncMode field.

When using an intermediate reload points, the PWM synchronization is not required. Multiple reload points can be used by providing an OR'ed list of options available in the enumeration [ftm\\_reload\\_point\\_t](#) to the reloadPoints field.

The driver initialization function sets up the appropriate bits in the FTM module based on the register update options selected.

If software PWM synchronization is used, the below function can be used to initiate a software trigger.

```
FTM_SetSoftwareTrigger(FTM0, true)
```

## 17.4 Typical use case

### 17.4.1 PWM output

Output a PWM signal on two FTM channels with different duty cycles. Periodically update the PWM signal duty cycle.

```
int main(void)
{
    bool brightnessUp = true; /* Indicates whether LEDs are brighter or dimmer. */
    ftm_config_t ftmInfo;
    uint8_t updatedDutycycle = 0U;
    ftm_chnl_pwm_signal_param_t ftmParam[2];

    /* Configures the FTM parameters with frequency 24 kHz */
    ftmParam[0].chnlNumber = (ftm_chnl_t)BOARD_FIRST_FTM_CHANNEL;
    ftmParam[0].level = kFTM_LowTrue;
    ftmParam[0].dutyCyclePercent = 0U;
    ftmParam[0].firstEdgeDelayPercent = 0U;

    ftmParam[1].chnlNumber = (ftm_chnl_t)BOARD_SECOND_FTM_CHANNEL;
    ftmParam[1].level = kFTM_LowTrue;
    ftmParam[1].dutyCyclePercent = 0U;
    ftmParam[1].firstEdgeDelayPercent = 0U;

    FTM_GetDefaultConfig(&ftmInfo);

    /* Initializes the FTM module. */
    FTM_Init(BOARD_FTM_BASEADDR, &ftmInfo);

    FTM_SetupPwm(BOARD_FTM_BASEADDR, ftmParam, 2U,
                 kFTM_EdgeAlignedPwm, 24000U, FTM_SOURCE_CLOCK);
    FTM_StartTimer(BOARD_FTM_BASEADDR, kFTM_SystemClock);

    while (1)
    {
        /* Delays to check whether the LED brightness has changed. */
        delay();

        if (brightnessUp)
        {
            /* Increases the duty cycle until it reaches a limited value. */
            if (++updatedDutycycle == 100U)
            {
                brightnessUp = false;
            }
        }
        else
        {
            /* Decreases the duty cycle until it reaches a limited value. */
            if (--updatedDutycycle == 0U)
            {
                brightnessUp = true;
            }
        }

        /* Starts the PWM mode with an updated duty cycle. */
        FTM_UpdatePwmDutycycle(BOARD_FTM_BASEADDR, (
            ftm_chnl_t)BOARD_FIRST_FTM_CHANNEL, kFTM_EdgeAlignedPwm,
            updatedDutycycle);
        FTM_UpdatePwmDutycycle(BOARD_FTM_BASEADDR, (
            ftm_chnl_t)BOARD_SECOND_FTM_CHANNEL, kFTM_EdgeAlignedPwm,
            updatedDutycycle);
        /* Software trigger to update registers. */
        FTM_SetSoftwareTrigger(BOARD_FTM_BASEADDR, true);
    }
}
```

## Typical use case

## Data Structures

- struct [ftm\\_chnl\\_pwm\\_signal\\_param\\_t](#)  
*Options to configure a FTM channel's PWM signal.* [More...](#)
- struct [ftm\\_dual\\_edge\\_capture\\_param\\_t](#)  
*FlexTimer dual edge capture parameters.* [More...](#)
- struct [ftm\\_phase\\_params\\_t](#)  
*FlexTimer quadrature decode phase parameters.* [More...](#)
- struct [ftm\\_fault\\_param\\_t](#)  
*Structure is used to hold the parameters to configure a FTM fault.* [More...](#)
- struct [ftm\\_config\\_t](#)  
*FTM configuration structure.* [More...](#)

## Enumerations

- enum [ftm\\_chnl\\_t](#) {  
  kFTM\_Chnl\_0 = 0U,  
  kFTM\_Chnl\_1,  
  kFTM\_Chnl\_2,  
  kFTM\_Chnl\_3,  
  kFTM\_Chnl\_4,  
  kFTM\_Chnl\_5,  
  kFTM\_Chnl\_6,  
  kFTM\_Chnl\_7 }  
*List of FTM channels.*
- enum [ftm\\_fault\\_input\\_t](#) {  
  kFTM\_Fault\_0 = 0U,  
  kFTM\_Fault\_1,  
  kFTM\_Fault\_2,  
  kFTM\_Fault\_3 }  
*List of FTM faults.*
- enum [ftm\\_pwm\\_mode\\_t](#) {  
  kFTM\_EdgeAlignedPwm = 0U,  
  kFTM\_CenterAlignedPwm,  
  kFTM\_CombinedPwm }  
*FTM PWM operation modes.*
- enum [ftm\\_pwm\\_level\\_select\\_t](#) {  
  kFTM\_NoPwmSignal = 0U,  
  kFTM\_LowTrue,  
  kFTM\_HighTrue }  
*FTM PWM output pulse mode: high-true, low-true or no output.*
- enum [ftm\\_output\\_compare\\_mode\\_t](#) {  
  kFTM\_NoOutputSignal = (1U << FTM\_CnSC\_MSA\_SHIFT),  
  kFTM\_ToggleOnMatch = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (1U << FTM\_CnSC\_ELSA\_S-  
  HIFT)),  
  kFTM\_ClearOnMatch = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (2U << FTM\_CnSC\_ELSA\_SH-  
  IFT)),  
  kFTM\_SetOnMatch = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (3U << FTM\_CnSC\_ELSA\_SHIF-

- T)) }
- FlexTimer output compare mode.*
- enum `ftm_input_capture_edge_t` {
   
kFTM\_RisingEdge = (1U << FTM\_CnSC\_ELSA\_SHIFT),
   
kFTM\_FallingEdge = (2U << FTM\_CnSC\_ELSA\_SHIFT),
   
kFTM\_RiseAndFallEdge = (3U << FTM\_CnSC\_ELSA\_SHIFT) }
- FlexTimer input capture edge.*
- enum `ftm_dual_edge_capture_mode_t` {
   
kFTM\_OneShot = 0U,
   
kFTM\_Continuous = (1U << FTM\_CnSC\_MSA\_SHIFT) }
- FlexTimer dual edge capture modes.*
- enum `ftm_quad_decode_mode_t` {
   
kFTM\_QualPhaseEncode = 0U,
   
kFTM\_QualCountAndDir }
- FlexTimer quadrature decode modes.*
- enum `ftm_phase_polarity_t` {
   
kFTM\_QualPhaseNormal = 0U,
   
kFTM\_QualPhaseInvert }
- FlexTimer quadrature phase polarities.*
- enum `ftm_deadtime_prescale_t` {
   
kFTM\_Deadtime\_Prescale\_1 = 1U,
   
kFTM\_Deadtime\_Prescale\_4,
   
kFTM\_Deadtime\_Prescale\_16 }
- FlexTimer pre-scaler factor for the dead time insertion.*
- enum `ftm_clock_source_t` {
   
kFTM\_SystemClock = 1U,
   
kFTM\_FixedClock,
   
kFTM\_ExternalClock }
- FlexTimer clock source selection.*
- enum `ftm_clock_prescale_t` {
   
kFTM\_Prescale\_Divide\_1 = 0U,
   
kFTM\_Prescale\_Divide\_2,
   
kFTM\_Prescale\_Divide\_4,
   
kFTM\_Prescale\_Divide\_8,
   
kFTM\_Prescale\_Divide\_16,
   
kFTM\_Prescale\_Divide\_32,
   
kFTM\_Prescale\_Divide\_64,
   
kFTM\_Prescale\_Divide\_128 }
- FlexTimer pre-scaler factor selection for the clock source.*
- enum `ftm_bdm_mode_t` {
   
kFTM\_BdmMode\_0 = 0U,
   
kFTM\_BdmMode\_1,
   
kFTM\_BdmMode\_2,
   
kFTM\_BdmMode\_3 }
- Options for the FlexTimer behaviour in BDM Mode.*
- enum `ftm_fault_mode_t` {

## Typical use case

```
kFTM_Fault_Disable = 0U,  
kFTM_Fault_EvenChnls,  
kFTM_Fault_AllChnlsMan,  
kFTM_Fault_AllChnlsAuto }
```

*Options for the FTM fault control mode.*

- enum `ftm_external_trigger_t` {  
    kFTM\_Chnl0Trigger = (1U << 4),  
    kFTM\_Chnl1Trigger = (1U << 5),  
    kFTM\_Chnl2Trigger = (1U << 0),  
    kFTM\_Chnl3Trigger = (1U << 1),  
    kFTM\_Chnl4Trigger = (1U << 2),  
    kFTM\_Chnl5Trigger = (1U << 3),  
    kFTM\_Chnl6Trigger,  
    kFTM\_Chnl7Trigger,  
    kFTM\_InitTrigger = (1U << 6),  
    kFTM\_ReloadInitTrigger = (1U << 7) }

*FTM external trigger options.*

- enum `ftm_pwm_sync_method_t` {  
    kFTM\_SoftwareTrigger = FTM\_SYNC\_SWSYNC\_MASK,  
    kFTM\_HardwareTrigger\_0 = FTM\_SYNC\_TRIG0\_MASK,  
    kFTM\_HardwareTrigger\_1 = FTM\_SYNC\_TRIG1\_MASK,  
    kFTM\_HardwareTrigger\_2 = FTM\_SYNC\_TRIG2\_MASK }

*FlexTimer PWM sync options to update registers with buffer.*

- enum `ftm_reload_point_t` {  
    kFTM\_Chnl0Match = (1U << 0),  
    kFTM\_Chnl1Match = (1U << 1),  
    kFTM\_Chnl2Match = (1U << 2),  
    kFTM\_Chnl3Match = (1U << 3),  
    kFTM\_Chnl4Match = (1U << 4),  
    kFTM\_Chnl5Match = (1U << 5),  
    kFTM\_Chnl6Match = (1U << 6),  
    kFTM\_Chnl7Match = (1U << 7),  
    kFTM\_CntMax = (1U << 8),  
    kFTM\_CntMin = (1U << 9),  
    kFTM\_HalfCycMatch = (1U << 10) }

*FTM options available as loading point for register reload.*

- enum `ftm_interrupt_enable_t` {

```

kFTM_Chnl0InterruptEnable = (1U << 0),
kFTM_Chnl1InterruptEnable = (1U << 1),
kFTM_Chnl2InterruptEnable = (1U << 2),
kFTM_Chnl3InterruptEnable = (1U << 3),
kFTM_Chnl4InterruptEnable = (1U << 4),
kFTM_Chnl5InterruptEnable = (1U << 5),
kFTM_Chnl6InterruptEnable = (1U << 6),
kFTM_Chnl7InterruptEnable = (1U << 7),
kFTM_FaultInterruptEnable = (1U << 8),
kFTM_TimeOverflowInterruptEnable = (1U << 9),
kFTM_ReloadInterruptEnable = (1U << 10) }

```

*List of FTM interrupts.*

- enum `ftm_status_flags_t` {

```

kFTM_Chnl0Flag = (1U << 0),
kFTM_Chnl1Flag = (1U << 1),
kFTM_Chnl2Flag = (1U << 2),
kFTM_Chnl3Flag = (1U << 3),
kFTM_Chnl4Flag = (1U << 4),
kFTM_Chnl5Flag = (1U << 5),
kFTM_Chnl6Flag = (1U << 6),
kFTM_Chnl7Flag = (1U << 7),
kFTM_FaultFlag = (1U << 8),
kFTM_TimeOverflowFlag = (1U << 9),
kFTM_ChnlTriggerFlag = (1U << 10),
kFTM_ReloadFlag = (1U << 11) }

```

*List of FTM flags.*

- enum `_ftm_quad_decoder_flags` {

```

kFTM_QquadDecoderCountingIncreaseFlag = FTM_QDCTRL_QUADIR_MASK,
kFTM_QquadDecoderCountingOverflowOnTopFlag = FTM_QDCTRL_TOFDIR_MASK }

```

*List of FTM Quad Decoder flags.*

## Functions

- void `FTM_SetupFault` (FTM\_Type \*base, `ftm_fault_input_t` faultNumber, const `ftm_fault_param_t` \*faultParams)
 

*Sets up the working of the FTM fault protection.*
- static void `FTM_SetGlobalTimeBaseOutputEnable` (FTM\_Type \*base, bool enable)
 

*Enables or disables the FTM global time base signal generation to other FTMs.*
- static void `FTM_SetOutputMask` (FTM\_Type \*base, `ftm_chnl_t` chnlNumber, bool mask)
 

*Sets the FTM peripheral timer channel output mask.*
- static void `FTM_SetSoftwareTrigger` (FTM\_Type \*base, bool enable)
 

*Enables or disables the FTM software trigger for PWM synchronization.*
- static void `FTM_SetWriteProtection` (FTM\_Type \*base, bool enable)
 

*Enables or disables the FTM write protection.*

## Driver version

- #define `FSL_FTM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 2)`)

## Typical use case

Version 2.0.2.

## Initialization and deinitialization

- status\_t **FTM\_Init** (FTM\_Type \*base, const **ftm\_config\_t** \*config)  
*Ungates the FTM clock and configures the peripheral for basic operation.*
- void **FTM\_Deinit** (FTM\_Type \*base)  
*Gates the FTM clock.*
- void **FTM\_GetDefaultConfig** (**ftm\_config\_t** \*config)  
*Fills in the FTM configuration structure with the default settings.*

## Channel mode operations

- status\_t **FTM\_SetupPwm** (FTM\_Type \*base, const **ftm\_chnl\_pwm\_signal\_param\_t** \*chnlParams, uint8\_t numOfChnls, **ftm\_pwm\_mode\_t** mode, uint32\_t pwmFreq\_Hz, uint32\_t srcClock\_Hz)  
*Configures the PWM signal parameters.*
- void **FTM\_UpdatePwmDutyCycle** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_pwm\_mode\_t** currentPwmMode, uint8\_t dutyCyclePercent)  
*Updates the duty cycle of an active PWM signal.*
- void **FTM\_UpdateChnlEdgeLevelSelect** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, uint8\_t level)  
*Updates the edge level selection for a channel.*
- void **FTM\_SetupInputCapture** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_input\_capture\_edge\_t** captureMode, uint32\_t filterValue)  
*Enables capturing an input signal on the channel using the function parameters.*
- void **FTM\_SetupOutputCompare** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_output\_compare\_mode\_t** compareMode, uint32\_t compareValue)  
*Configures the FTM to generate timed pulses.*
- void **FTM\_SetupDualEdgeCapture** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, const **ftm\_dual\_edge\_capture\_param\_t** \*edgeParam, uint32\_t filterValue)  
*Configures the dual edge capture mode of the FTM.*

## Interrupt Interface

- void **FTM\_EnableInterrupts** (FTM\_Type \*base, uint32\_t mask)  
*Enables the selected FTM interrupts.*
- void **FTM\_DisableInterrupts** (FTM\_Type \*base, uint32\_t mask)  
*Disables the selected FTM interrupts.*
- uint32\_t **FTM\_GetEnabledInterrupts** (FTM\_Type \*base)  
*Gets the enabled FTM interrupts.*

## Status Interface

- uint32\_t **FTM\_GetStatusFlags** (FTM\_Type \*base)  
*Gets the FTM status flags.*
- void **FTM\_ClearStatusFlags** (FTM\_Type \*base, uint32\_t mask)  
*Clears the FTM status flags.*

## Read and write the timer period

- static void **FTM\_SetTimerPeriod** (FTM\_Type \*base, uint32\_t ticks)

- Sets the timer period in units of ticks.
- static uint32\_t **FTM\_GetCurrentTimerCount** (FTM\_Type \*base)  
Reads the current timer counting value.

## Timer Start and Stop

- static void **FTM\_StartTimer** (FTM\_Type \*base, **ftm\_clock\_source\_t** clockSource)  
Starts the FTM counter.
- static void **FTM\_StopTimer** (FTM\_Type \*base)  
Stops the FTM counter.

## Software output control

- static void **FTM\_SetSoftwareCtrlEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool value)  
Enables or disables the channel software output control.
- static void **FTM\_SetSoftwareCtrlVal** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool value)  
Sets the channel software output control value.

## Channel pair operations

- static void **FTM\_SetFaultControlEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
This function enables/disables the fault control in a channel pair.
- static void **FTM\_SetDeadTimeEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
This function enables/disables the dead time insertion in a channel pair.
- static void **FTM\_SetComplementaryEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
This function enables/disables complementary mode in a channel pair.
- static void **FTM\_SetInvertEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)  
This function enables/disables inverting control in a channel pair.

## Quad Decoder

- void **FTM\_SetupQuadDecode** (FTM\_Type \*base, const **ftm\_phase\_params\_t** \*phaseAParams, const **ftm\_phase\_params\_t** \*phaseBParams, **ftm\_quad\_decode\_mode\_t** quadMode)  
Configures the parameters and activates the quadrature decoder mode.
- static uint32\_t **FTM\_GetQuadDecoderFlags** (FTM\_Type \*base)  
Gets the FTM Quad Decoder flags.
- static void **FTM\_SetQuadDecoderModuloValue** (FTM\_Type \*base, uint32\_t startValue, uint32\_t overValue)  
Sets the modulo values for Quad Decoder.
- static uint32\_t **FTM\_GetQuadDecoderCounterValue** (FTM\_Type \*base)  
Gets the current Quad Decoder counter value.
- static void **FTM\_ClearQuadDecoderCounterValue** (FTM\_Type \*base)  
Clears the current Quad Decoder counter value.

## Data Structure Documentation

### 17.5 Data Structure Documentation

#### 17.5.1 struct ftm\_chnl\_pwm\_signal\_param\_t

##### Data Fields

- **ftm\_chnl\_t chnlNumber**  
*The channel/channel pair number.*
- **ftm\_pwm\_level\_select\_t level**  
*PWM output active level select.*
- **uint8\_t dutyCyclePercent**  
*PWM pulse width, value should be between 0 to 100 0 = inactive signal(0% duty cycle)...*
- **uint8\_t firstEdgeDelayPercent**  
*Used only in combined PWM mode to generate an asymmetrical PWM.*

##### 17.5.1.0.0.49 Field Documentation

###### 17.5.1.0.0.49.1 ftm\_chnl\_t ftm\_chnl\_pwm\_signal\_param\_t::chnlNumber

In combined mode, this represents the channel pair number.

###### 17.5.1.0.0.49.2 ftm\_pwm\_level\_select\_t ftm\_chnl\_pwm\_signal\_param\_t::level

###### 17.5.1.0.0.49.3 uint8\_t ftm\_chnl\_pwm\_signal\_param\_t::dutyCyclePercent

100 = always active signal (100% duty cycle).

###### 17.5.1.0.0.49.4 uint8\_t ftm\_chnl\_pwm\_signal\_param\_t::firstEdgeDelayPercent

Specifies the delay to the first edge in a PWM period. If unsure leave as 0; Should be specified as a percentage of the PWM period

### 17.5.2 struct ftm\_dual\_edge\_capture\_param\_t

##### Data Fields

- **ftm\_dual\_edge\_capture\_mode\_t mode**  
*Dual Edge Capture mode.*
- **ftm\_input\_capture\_edge\_t currChanEdgeMode**  
*Input capture edge select for channel n.*
- **ftm\_input\_capture\_edge\_t nextChanEdgeMode**  
*Input capture edge select for channel n+1.*

### 17.5.3 struct `ftm_phase_params_t`

#### Data Fields

- bool `enablePhaseFilter`  
*True: enable phase filter; false: disable filter.*
- uint32\_t `phaseFilterVal`  
*Filter value, used only if phase filter is enabled.*
- `ftm_phase_polarity_t phasePolarity`  
*Phase polarity.*

### 17.5.4 struct `ftm_fault_param_t`

#### Data Fields

- bool `enableFaultInput`  
*True: Fault input is enabled; false: Fault input is disabled.*
- bool `faultLevel`  
*True: Fault polarity is active low; in other words, '0' indicates a fault; False: Fault polarity is active high.*
- bool `useFaultFilter`  
*True: Use the filtered fault signal; False: Use the direct path from fault input.*

### 17.5.5 struct `ftm_config_t`

This structure holds the configuration settings for the FTM peripheral. To initialize this structure to reasonable defaults, call the [FTM\\_GetDefaultConfig\(\)](#) function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

#### Data Fields

- `ftm_clock_prescale_t prescale`  
*FTM clock prescale value.*
- `ftm_bdm_mode_t bdmMode`  
*FTM behavior in BDM mode.*
- uint32\_t `pwmSyncMode`  
*Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration [ftm\\_pwm\\_sync\\_method\\_t](#).*
- uint32\_t `reloadPoints`  
*FTM reload points; When using this, the PWM synchronization is not required.*
- `ftm_fault_mode_t faultMode`  
*FTM fault control mode.*
- uint8\_t `faultFilterValue`  
*Fault input filter value.*

## Enumeration Type Documentation

- **ftm\_deadtime\_prescale\_t deadTimePrescale**  
*The dead time prescalar value.*
- **uint32\_t deadTimeValue**  
*The dead time value deadTimeValue's available range is 0-1023 when register has DTVALEX, otherwise its available range is 0-63.*
- **uint32\_t extTriggers**  
*External triggers to enable.*
- **uint8\_t chnlInitState**  
*Defines the initialization value of the channels in OUTINT register.*
- **uint8\_t chnlPolarity**  
*Defines the output polarity of the channels in POL register.*
- **bool useGlobalTimeBase**  
*True: Use of an external global time base is enabled; False: disabled.*

### 17.5.5.0.0.50 Field Documentation

#### 17.5.5.0.0.50.1 uint32\_t ftm\_config\_t::pwmSyncMode

#### 17.5.5.0.0.50.2 uint32\_t ftm\_config\_t::reloadPoints

Multiple reload points can be used by providing an OR'ed list of options available in enumeration [ftm\\_reload\\_point\\_t](#).

#### 17.5.5.0.0.50.3 uint32\_t ftm\_config\_t::deadTimeValue

#### 17.5.5.0.0.50.4 uint32\_t ftm\_config\_t::extTriggers

Multiple trigger sources can be enabled by providing an OR'ed list of options available in enumeration [ftm\\_external\\_trigger\\_t](#).

## 17.6 Enumeration Type Documentation

### 17.6.1 enum ftm\_chnl\_t

Note

Actual number of available channels is SoC dependent

Enumerator

- kFTM\_Chnl\_0** FTM channel number 0.
- kFTM\_Chnl\_1** FTM channel number 1.
- kFTM\_Chnl\_2** FTM channel number 2.
- kFTM\_Chnl\_3** FTM channel number 3.
- kFTM\_Chnl\_4** FTM channel number 4.
- kFTM\_Chnl\_5** FTM channel number 5.
- kFTM\_Chnl\_6** FTM channel number 6.
- kFTM\_Chnl\_7** FTM channel number 7.

## 17.6.2 enum ftm\_fault\_input\_t

Enumerator

- kFTM\_Fault\_0* FTM fault 0 input pin.
- kFTM\_Fault\_1* FTM fault 1 input pin.
- kFTM\_Fault\_2* FTM fault 2 input pin.
- kFTM\_Fault\_3* FTM fault 3 input pin.

## 17.6.3 enum ftm\_pwm\_mode\_t

Enumerator

- kFTM\_EdgeAlignedPwm* Edge-aligned PWM.
- kFTM\_CenterAlignedPwm* Center-aligned PWM.
- kFTM\_CombinedPwm* Combined PWM.

## 17.6.4 enum ftm\_pwm\_level\_select\_t

Enumerator

- kFTM\_NoPwmSignal* No PWM output on pin.
- kFTM\_LowTrue* Low true pulses.
- kFTM\_HighTrue* High true pulses.

## 17.6.5 enum ftm\_output\_compare\_mode\_t

Enumerator

- kFTM\_NoOutputSignal* No channel output when counter reaches CnV.
- kFTM\_ToggleOnMatch* Toggle output.
- kFTM\_ClearOnMatch* Clear output.
- kFTM\_SetOnMatch* Set output.

## 17.6.6 enum ftm\_input\_capture\_edge\_t

Enumerator

- kFTM\_RisingEdge* Capture on rising edge only.
- kFTM\_FallingEdge* Capture on falling edge only.
- kFTM\_RiseAndFallEdge* Capture on rising or falling edge.

## Enumeration Type Documentation

### 17.6.7 enum ftm\_dual\_edge\_capture\_mode\_t

Enumerator

*kFTM\_OneShot* One-shot capture mode.

*kFTM\_Continuous* Continuous capture mode.

### 17.6.8 enum ftm\_quad\_decode\_mode\_t

Enumerator

*kFTM\_QuadPhaseEncode* Phase A and Phase B encoding mode.

*kFTM\_QuadCountAndDir* Count and direction encoding mode.

### 17.6.9 enum ftm\_phase\_polarity\_t

Enumerator

*kFTM\_QuadPhaseNormal* Phase input signal is not inverted.

*kFTM\_QuadPhaseInvert* Phase input signal is inverted.

### 17.6.10 enum ftm\_deadtime\_prescale\_t

Enumerator

*kFTM\_Deadtime\_Prescale\_1* Divide by 1.

*kFTM\_Deadtime\_Prescale\_4* Divide by 4.

*kFTM\_Deadtime\_Prescale\_16* Divide by 16.

### 17.6.11 enum ftm\_clock\_source\_t

Enumerator

*kFTM\_SystemClock* System clock selected.

*kFTM\_FixedClock* Fixed frequency clock.

*kFTM\_ExternalClock* External clock.

### 17.6.12 enum ftm\_clock\_prescale\_t

Enumerator

- kFTM\_Prescale\_Divide\_1* Divide by 1.
- kFTM\_Prescale\_Divide\_2* Divide by 2.
- kFTM\_Prescale\_Divide\_4* Divide by 4.
- kFTM\_Prescale\_Divide\_8* Divide by 8.
- kFTM\_Prescale\_Divide\_16* Divide by 16.
- kFTM\_Prescale\_Divide\_32* Divide by 32.
- kFTM\_Prescale\_Divide\_64* Divide by 64.
- kFTM\_Prescale\_Divide\_128* Divide by 128.

### 17.6.13 enum ftm\_bdm\_mode\_t

Enumerator

- kFTM\_BdmMode\_0* FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
- kFTM\_BdmMode\_1* FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
- kFTM\_BdmMode\_2* FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
- kFTM\_BdmMode\_3* FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode.

### 17.6.14 enum ftm\_fault\_mode\_t

Enumerator

- kFTM\_Fault\_Disable* Fault control is disabled for all channels.
- kFTM\_Fault\_EvenChnls* Enabled for even channels only(0,2,4,6) with manual fault clearing.
- kFTM\_Fault\_AllChnlsMan* Enabled for all channels with manual fault clearing.
- kFTM\_Fault\_AllChnlsAuto* Enabled for all channels with automatic fault clearing.

### 17.6.15 enum ftm\_external\_trigger\_t

## Enumeration Type Documentation

Note

Actual available external trigger sources are SoC-specific

Enumerator

- kFTM\_Chnl0Trigger*** Generate trigger when counter equals chnl 0 CnV reg.
- kFTM\_Chnl1Trigger*** Generate trigger when counter equals chnl 1 CnV reg.
- kFTM\_Chnl2Trigger*** Generate trigger when counter equals chnl 2 CnV reg.
- kFTM\_Chnl3Trigger*** Generate trigger when counter equals chnl 3 CnV reg.
- kFTM\_Chnl4Trigger*** Generate trigger when counter equals chnl 4 CnV reg.
- kFTM\_Chnl5Trigger*** Generate trigger when counter equals chnl 5 CnV reg.
- kFTM\_Chnl6Trigger*** Available on certain SoC's, generate trigger when counter equals chnl 6 CnV reg.
- kFTM\_Chnl7Trigger*** Available on certain SoC's, generate trigger when counter equals chnl 7 CnV reg.
- kFTM\_InitTrigger*** Generate Trigger when counter is updated with CNTIN.
- kFTM\_ReloadInitTrigger*** Available on certain SoC's, trigger on reload point.

### 17.6.16 enum ftm\_pwm\_sync\_method\_t

Enumerator

- kFTM\_SoftwareTrigger*** Software triggers PWM sync.
- kFTM\_HardwareTrigger\_0*** Hardware trigger 0 causes PWM sync.
- kFTM\_HardwareTrigger\_1*** Hardware trigger 1 causes PWM sync.
- kFTM\_HardwareTrigger\_2*** Hardware trigger 2 causes PWM sync.

### 17.6.17 enum ftm\_reload\_point\_t

Note

Actual available reload points are SoC-specific

Enumerator

- kFTM\_Chnl0Match*** Channel 0 match included as a reload point.
- kFTM\_Chnl1Match*** Channel 1 match included as a reload point.
- kFTM\_Chnl2Match*** Channel 2 match included as a reload point.
- kFTM\_Chnl3Match*** Channel 3 match included as a reload point.
- kFTM\_Chnl4Match*** Channel 4 match included as a reload point.
- kFTM\_Chnl5Match*** Channel 5 match included as a reload point.
- kFTM\_Chnl6Match*** Channel 6 match included as a reload point.
- kFTM\_Chnl7Match*** Channel 7 match included as a reload point.

***kFTM\_CntMax*** Use in up-down count mode only, reload when counter reaches the maximum value.

***kFTM\_CntMin*** Use in up-down count mode only, reload when counter reaches the minimum value.

***kFTM\_HalfCycMatch*** Available on certain SoC's, half cycle match reload point.

### 17.6.18 enum ftm\_interrupt\_enable\_t

Note

Actual available interrupts are SoC-specific

Enumerator

***kFTM\_Chnl0InterruptEnable*** Channel 0 interrupt.

***kFTM\_Chnl1InterruptEnable*** Channel 1 interrupt.

***kFTM\_Chnl2InterruptEnable*** Channel 2 interrupt.

***kFTM\_Chnl3InterruptEnable*** Channel 3 interrupt.

***kFTM\_Chnl4InterruptEnable*** Channel 4 interrupt.

***kFTM\_Chnl5InterruptEnable*** Channel 5 interrupt.

***kFTM\_Chnl6InterruptEnable*** Channel 6 interrupt.

***kFTM\_Chnl7InterruptEnable*** Channel 7 interrupt.

***kFTM\_FaultInterruptEnable*** Fault interrupt.

***kFTM\_TimeOverflowInterruptEnable*** Time overflow interrupt.

***kFTM\_ReloadInterruptEnable*** Reload interrupt; Available only on certain SoC's.

### 17.6.19 enum ftm\_status\_flags\_t

Note

Actual available flags are SoC-specific

Enumerator

***kFTM\_Chnl0Flag*** Channel 0 Flag.

***kFTM\_Chnl1Flag*** Channel 1 Flag.

***kFTM\_Chnl2Flag*** Channel 2 Flag.

***kFTM\_Chnl3Flag*** Channel 3 Flag.

***kFTM\_Chnl4Flag*** Channel 4 Flag.

***kFTM\_Chnl5Flag*** Channel 5 Flag.

***kFTM\_Chnl6Flag*** Channel 6 Flag.

***kFTM\_Chnl7Flag*** Channel 7 Flag.

***kFTM\_FaultFlag*** Fault Flag.

## Function Documentation

*kFTM\_TimeOverflowFlag* Time overflow Flag.

*kFTM\_ChnlTriggerFlag* Channel trigger Flag.

*kFTM\_ReloadFlag* Reload Flag; Available only on certain SoC's.

### 17.6.20 enum \_ftm\_quad\_decoder\_flags

Enumerator

*kFTM\_QuadDecoderCountingIncreaseFlag* Counting direction is increasing (FTM counter increment), or the direction is decreasing.

*kFTM\_QuadDecoderCountingOverflowOnTopFlag* Indicates if the TOF bit was set on the top or the bottom of counting.

## 17.7 Function Documentation

### 17.7.1 status\_t FTM\_Init ( FTM\_Type \* *base*, const ftm\_config\_t \* *config* )

Note

This API should be called at the beginning of the application which is using the FTM driver.

Parameters

<i>base</i>	FTM peripheral base address
<i>config</i>	Pointer to the user configuration structure.

Returns

kStatus\_Success indicates success; Else indicates failure.

### 17.7.2 void FTM\_Deinit ( FTM\_Type \* *base* )

Parameters

<i>base</i>	FTM peripheral base address
-------------	-----------------------------

### 17.7.3 void FTM\_GetDefaultConfig ( ftm\_config\_t \* *config* )

The default values are:

```

* config->prescale = kFTM_Prescale_Divide_1;
* config->bdmMode = kFTM_BdmMode_0;
* config->pwmSyncMode = kFTM_SoftwareTrigger;
* config->reloadPoints = 0;
* config->faultMode = kFTM_Fault_Disable;
* config->faultFilterValue = 0;
* config->deadTimePrescale = kFTM_Deadtime_Prescale_1;
* config->deadTimeValue = 0;
* config->extTriggers = 0;
* config->chnlInitState = 0;
* config->chnlPolarity = 0;
* config->useGlobalTimeBase = false;
*

```

## Parameters

<i>config</i>	Pointer to the user configuration structure.
---------------	--

#### 17.7.4 status\_t FTM\_SetupPwm ( **FTM\_Type** \* *base*, **const ftm\_chnl\_pwm\_signal\_param\_t** \* *chnlParams*, **uint8\_t** *numOfChnls*, **ftm\_pwm\_mode\_t** *mode*, **uint32\_t** *pwmFreq\_Hz*, **uint32\_t** *srcClock\_Hz* )

Call this function to configure the PWM signal period, mode, duty cycle, and edge. Use this function to configure all FTM channels that are used to output a PWM signal.

## Parameters

<i>base</i>	FTM peripheral base address
<i>chnlParams</i>	Array of PWM channel parameters to configure the channel(s)
<i>numOfChnls</i>	Number of channels to configure; This should be the size of the array passed in
<i>mode</i>	PWM operation mode, options available in enumeration <b>ftm_pwm_mode_t</b>
<i>pwmFreq_Hz</i>	PWM signal frequency in Hz
<i>srcClock_Hz</i>	FTM counter clock in Hz

## Returns

kStatus\_Success if the PWM setup was successful kStatus\_Error on failure

#### 17.7.5 void FTM\_UpdatePwmDutycycle ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlNumber*, **ftm\_pwm\_mode\_t** *currentPwmMode*, **uint8\_t** *dutyCyclePercent* )

## Function Documentation

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel/channel pair number. In combined mode, this represents the channel pair number
<i>currentPwm-Mode</i>	The current PWM mode set during PWM setup
<i>dutyCycle-Percent</i>	New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

### **17.7.6 void FTM\_UpdateChnlEdgeLevelSelect ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, uint8\_t *level* )**

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel number
<i>level</i>	The level to be set to the ELSnB:ELSnA field; Valid values are 00, 01, 10, 11. See the Kinetis SoC reference manual for details about this field.

### **17.7.7 void FTM\_SetupInputCapture ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, ftm\_input\_capture\_edge\_t *captureMode*, uint32\_t *filterValue* )**

When the edge specified in the captureMode argument occurs on the channel, the FTM counter is captured into the CnV register. The user has to read the CnV register separately to get this value. The filter function is disabled if the filterVal argument passed in is 0. The filter function is available only for channels 0, 1, 2, 3.

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel number
<i>captureMode</i>	Specifies which edge to capture

<i>filterValue</i>	Filter value, specify 0 to disable filter. Available only for channels 0-3.
--------------------	---

### 17.7.8 void FTM\_SetupOutputCompare ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlNumber*, **ftm\_output\_compare\_mode\_t** *compareMode*, **uint32\_t** *compareValue* )

When the FTM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel number
<i>compareMode</i>	Action to take on the channel output when the compare condition is met
<i>compareValue</i>	Value to be programmed in the CnV register.

### 17.7.9 void FTM\_SetupDualEdgeCapture ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlPairNumber*, **const ftm\_dual\_edge\_capture\_param\_t** \* *edgeParam*, **uint32\_t** *filterValue* )

This function sets up the dual edge capture mode on a channel pair. The capture edge for the channel pair and the capture mode (one-shot or continuous) is specified in the parameter argument. The filter function is disabled if the filterVal argument passed is zero. The filter function is available only on channels 0 and 2. The user has to read the channel CnV registers separately to get the capture values.

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>edgeParam</i>	Sets up the dual edge capture function
<i>filterValue</i>	Filter value, specify 0 to disable filter. Available only for channel pair 0 and 1.

### 17.7.10 void FTM\_SetupFault ( **FTM\_Type** \* *base*, **ftm\_fault\_input\_t** *faultNumber*, **const ftm\_fault\_param\_t** \* *faultParams* )

FTM can have up to 4 fault inputs. This function sets up fault parameters, fault level, and a filter.

## Function Documentation

Parameters

<i>base</i>	FTM peripheral base address
<i>faultNumber</i>	FTM fault to configure.
<i>faultParams</i>	Parameters passed in to set up the fault

### **17.7.11 void FTM\_EnableInterrupts ( FTM\_Type \* *base*, uint32\_t *mask* )**

Parameters

<i>base</i>	FTM peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">ftm_interrupt_enable_t</a>

### **17.7.12 void FTM\_DisableInterrupts ( FTM\_Type \* *base*, uint32\_t *mask* )**

Parameters

<i>base</i>	FTM peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">ftm_interrupt_enable_t</a>

### **17.7.13 uint32\_t FTM\_GetEnabledInterrupts ( FTM\_Type \* *base* )**

Parameters

<i>base</i>	FTM peripheral base address
-------------	-----------------------------

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [ftm\\_interrupt\\_enable\\_t](#)

### **17.7.14 uint32\_t FTM\_GetStatusFlags ( FTM\_Type \* *base* )**

Parameters

<i>base</i>	FTM peripheral base address
-------------	-----------------------------

Returns

The status flags. This is the logical OR of members of the enumeration [ftm\\_status\\_flags\\_t](#)

### 17.7.15 void FTM\_ClearStatusFlags ( **FTM\_Type** \* *base*, **uint32\_t** *mask* )

Parameters

<i>base</i>	FTM peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration <a href="#">ftm_status_flags_t</a>

### 17.7.16 static void FTM\_SetTimerPeriod ( **FTM\_Type** \* *base*, **uint32\_t** *ticks* ) [**inline**], [**static**]

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note

1. This API allows the user to use the FTM module as a timer. Do not mix usage of this API with FTM's PWM setup API's.
2. Call the utility macros provided in the `fsl_common.h` to convert usec or msec to ticks.

Parameters

<i>base</i>	FTM peripheral base address
<i>ticks</i>	A timer period in units of ticks, which should be equal or greater than 1.

### 17.7.17 static **uint32\_t** FTM\_GetCurrentTimerCount ( **FTM\_Type** \* *base* ) [**inline**], [**static**]

This function returns the real-time timer counting value in a range from 0 to a timer period.

## Function Documentation

Note

Call the utility macros provided in the fsl\_common.h to convert ticks to usec or msec.

Parameters

<i>base</i>	FTM peripheral base address
-------------	-----------------------------

Returns

The current counter value in ticks

### 17.7.18 static void FTM\_StartTimer ( FTM\_Type \* *base*, ftm\_clock\_source\_t *clockSource* ) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>clockSource</i>	FTM clock source; After the clock source is set, the counter starts running.

### 17.7.19 static void FTM\_StopTimer ( FTM\_Type \* *base* ) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
-------------	-----------------------------

### 17.7.20 static void FTM\_SetSoftwareCtrlEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, bool *value* ) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	Channel to be enabled or disabled

<i>value</i>	true: channel output is affected by software output control false: channel output is unaffected by software output control
--------------	---

### 17.7.21 static void FTM\_SetSoftwareCtrlVal ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, bool *value* ) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address.
<i>chnlNumber</i>	Channel to be configured
<i>value</i>	true to set 1, false to set 0

### 17.7.22 static void FTM\_SetGlobalTimeBaseOutputEnable ( FTM\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>enable</i>	true to enable, false to disable

### 17.7.23 static void FTM\_SetOutputMask ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, bool *mask* ) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	Channel to be configured
<i>mask</i>	true: masked, channel is forced to its inactive state; false: unmasked

### 17.7.24 static void FTM\_SetFaultControlEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: Enable fault control for this channel pair; false: No fault control

**17.7.25 static void FTM\_SetDeadTimeEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: Insert dead time in this channel pair; false: No dead time inserted

**17.7.26 static void FTM\_SetComplementaryEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: enable complementary mode; false: disable complementary mode

**17.7.27 static void FTM\_SetInvertEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: enable inverting; false: disable inverting

**17.7.28 void FTM\_SetupQuadDecode ( FTM\_Type \* *base*, const ftm\_phase\_params\_t \* *phaseAParams*, const ftm\_phase\_params\_t \* *phaseBParams*, ftm\_quad\_decode\_mode\_t *quadMode* )**

Parameters

<i>base</i>	FTM peripheral base address
<i>phaseAParams</i>	Phase A configuration parameters
<i>phaseBParams</i>	Phase B configuration parameters
<i>quadMode</i>	Selects encoding mode used in quadrature decoder mode

**17.7.29 static uint32\_t FTM\_GetQuadDecoderFlags ( FTM\_Type \* *base* ) [inline], [static]**

Parameters

<i>base</i>	FTM peripheral base address.
-------------	------------------------------

Returns

Flag mask of FTM Quad Decoder, see [\\_ftm\\_quad\\_decoder\\_flags](#).

**17.7.30 static void FTM\_SetQuadDecoderModuloValue ( FTM\_Type \* *base*, uint32\_t *startValue*, uint32\_t *overValue* ) [inline], [static]**

The modulo values configure the minimum and maximum values that the Quad decoder counter can reach. After the counter goes over, the counter value goes to the other side and decrease/increase again.

## Function Documentation

Parameters

<i>base</i>	FTM peripheral base address.
<i>startValue</i>	The low limit value for Quad Decoder counter.
<i>overValue</i>	The high limit value for Quad Decoder counter.

**17.7.31 static uint32\_t FTM\_GetQuadDecoderCounterValue ( FTM\_Type \* *base* )  
[inline], [static]**

Parameters

<i>base</i>	FTM peripheral base address.
-------------	------------------------------

Returns

Current quad Decoder counter value.

**17.7.32 static void FTM\_ClearQuadDecoderCounterValue ( FTM\_Type \* *base* )  
[inline], [static]**

The counter is set as the initial value.

Parameters

<i>base</i>	FTM peripheral base address.
-------------	------------------------------

**17.7.33 static void FTM\_SetSoftwareTrigger ( FTM\_Type \* *base*, bool *enable* )  
[inline], [static]**

Parameters

<i>base</i>	FTM peripheral base address
<i>enable</i>	true: software trigger is selected, false: software trigger is not selected

**17.7.34 static void FTM\_SetWriteProtection ( FTM\_Type \* *base*, bool *enable* )  
[inline], [static]**

## Parameters

<i>base</i>	FTM peripheral base address
<i>enable</i>	true: Write-protection is enabled, false: Write-protection is disabled

## Function Documentation

# Chapter 18

## GPIO: General-Purpose Input/Output Driver

### 18.1 Overview

#### Modules

- FGPIO Driver
- GPIO Driver

#### Data Structures

- struct `gpio_pin_config_t`  
*The GPIO pin configuration structure.* [More...](#)

#### Enumerations

- enum `gpio_pin_direction_t` {  
  `kGPIO_DigitalInput` = 0U,  
  `kGPIO_DigitalOutput` = 1U }  
*GPIO direction definition.*

#### Driver version

- #define `FSL_GPIO_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 1)`)  
*GPIO driver version 2.1.1.*

### 18.2 Data Structure Documentation

#### 18.2.1 struct `gpio_pin_config_t`

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the `outputConfig` unused. Note that in some use cases, the corresponding port property should be configured in advance with the [PORT\\_SetPinConfig\(\)](#).

#### Data Fields

- `gpio_pin_direction_t pinDirection`  
*GPIO direction, input or output.*
- `uint8_t outputLogic`  
*Set a default output logic, which has no use in input.*

## Enumeration Type Documentation

### 18.3 Macro Definition Documentation

18.3.1 `#define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))`

### 18.4 Enumeration Type Documentation

#### 18.4.1 `enum gpio_pin_direction_t`

Enumerator

*kGPIO\_DigitalInput* Set current pin as digital input.

*kGPIO\_DigitalOutput* Set current pin as digital output.

## 18.5 GPIO Driver

### 18.5.1 Overview

The MCUXpresso SDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of MCUXpresso SDK devices.

### 18.5.2 Typical use case

#### 18.5.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};

/* Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

#### 18.5.2.2 Input Operation

```
/* Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_GPIO_PIN,
                           kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};

/* Sets the input pin configuration */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```

## GPIO Configuration

- void **GPIO\_PinInit** (GPIO\_Type \*base, uint32\_t pin, const **gpio\_pin\_config\_t** \*config)  
*Initializes a GPIO pin used by the board.*

## GPIO Output Operations

- static void **GPIO\_WritePinOutput** (GPIO\_Type \*base, uint32\_t pin, uint8\_t output)  
*Sets the output level of the multiple GPIO pins to the logic 1 or 0.*
- static void **GPIO\_SetPinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 1.*
- static void **GPIO\_ClearPinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 0.*
- static void **GPIO\_TogglePinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Reverses the current output logic of the multiple GPIO pins.*

### GPIO Input Operations

- static uint32\_t [GPIO\\_ReadPinInput](#) (GPIO\_Type \*base, uint32\_t pin)  
*Reads the current input value of the GPIO port.*

### GPIO Interrupt

- uint32\_t [GPIO\\_GetPinsInterruptFlags](#) (GPIO\_Type \*base)  
*Reads the GPIO port interrupt status flag.*
- void [GPIO\\_ClearPinsInterruptFlags](#) (GPIO\_Type \*base, uint32\_t mask)  
*Clears multiple GPIO pin interrupt status flags.*

### 18.5.3 Function Documentation

#### 18.5.3.1 void [GPIO\\_PinInit](#) ( **GPIO\_Type** \* *base*, **uint32\_t** *pin*, **const gpio\_pin\_config\_t** \* *config* )

To initialize the GPIO, define a pin configuration, as either input or output, in the user file. Then, call the [GPIO\\_PinInit\(\)](#) function.

This is an example to define an input pin or an output pin configuration.

```
* // Define a digital input pin configuration,
* gpio_pin_config_t config =
* {
*   kGPIO_DigitalInput,
*   0,
* }
* //Define a digital output pin configuration,
* gpio_pin_config_t config =
* {
*   kGPIO_DigitalOutput,
*   0,
* }
```

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>pin</i>	GPIO port pin number
<i>config</i>	GPIO pin configuration pointer

#### 18.5.3.2 static void [GPIO\\_WritePinOutput](#) ( **GPIO\_Type** \* *base*, **uint32\_t** *pin*, **uint8\_t** *output* ) [inline], [static]

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>pin</i>	GPIO pin number
<i>output</i>	GPIO pin output logic level. <ul style="list-style-type: none"> <li>• 0: corresponding pin output low-logic level.</li> <li>• 1: corresponding pin output high-logic level.</li> </ul>

**18.5.3.3 static void GPIO\_SetPinsOutput ( **GPIO\_Type** \* *base*, **uint32\_t** *mask* )  
[inline], [static]**

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

**18.5.3.4 static void GPIO\_ClearPinsOutput ( **GPIO\_Type** \* *base*, **uint32\_t** *mask* )  
[inline], [static]**

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

**18.5.3.5 static void GPIO\_TogglePinsOutput ( **GPIO\_Type** \* *base*, **uint32\_t** *mask* )  
[inline], [static]**

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

**18.5.3.6 static **uint32\_t** GPIO\_ReadPinInput ( **GPIO\_Type** \* *base*, **uint32\_t** *pin* )  
[inline], [static]**

## GPIO Driver

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>pin</i>	GPIO pin number

Return values

<i>GPIO</i>	port input value <ul style="list-style-type: none"><li>• 0: corresponding pin input low-logic level.</li><li>• 1: corresponding pin input high-logic level.</li></ul>
-------------	---

### 18.5.3.7 `uint32_t GPIO_GetPinsInterruptFlags ( GPIO_Type * base )`

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
-------------	--

Return values

<i>The</i>	current GPIO port interrupt status flag, for example, 0x00010001 means the pin 0 and 17 have the interrupt.
------------	---

### 18.5.3.8 `void GPIO_ClearPinsInterruptFlags ( GPIO_Type * base, uint32_t mask )`

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

## 18.6 FGPIO Driver

This chapter describes the programming interface of the FGPIO driver. The FGPIO driver configures the FGPIO module and provides a functional interface to build the GPIO application.

Note

FGPIO (Fast GPIO) is only available in a few MCUs. FGPIO and GPIO share the same peripheral but use different registers. FGPIO is closer to the core than the regular GPIO and it's faster to read and write.

### 18.6.1 Typical use case

#### 18.6.1.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};

/* Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

#### 18.6.1.2 Input Operation

```
/* Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_FGPIO_PIN,
                           kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};

/* Sets the input pin configuration */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```



# **Chapter 19**

## **I2C: Inter-Integrated Circuit Driver**

### **19.1 Overview**

#### **Modules**

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C eDMA Driver

## I2C Driver

### 19.2 I2C Driver

#### 19.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of MCUXpresso SDK devices.

The I2C driver includes functional APIs and transactional APIs.

Functional APIs target the low-level APIs. Functional APIs can be used for the I2C master/slave initialization/configuration/operation for optimization/customization purpose. Using the functional APIs requires knowing the I2C master peripheral and how to organize functional APIs to meet the application requirements. The I2C functional operation groups provide the functional APIs set.

Transactional APIs target the high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code using the functional APIs or accessing the hardware registers.

Transactional APIs support asynchronous transfer. This means that the functions [I2C\\_MasterTransferNonBlocking\(\)](#) set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

#### 19.2.2 Typical use case

##### 19.2.2.1 Master Operation in functional method

```
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];

/* Gets the default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

/* Sends a start and a slave address. */
I2C_MasterStart(EXAMPLE_I2C_MASTER_BASEADDR, 7-bit slave address,
    kI2C_Write/kI2C_Read);

/* Waits for the sent out address. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR)) & kI2C_IntPendingFlag))
{
}

if(status & kI2C_ReceiveNakFlag)
{
    return kStatus_I2C_Nak;
}

result = I2C_MasterWriteBlocking(EXAMPLE_I2C_MASTER_BASEADDR, txBuff, BUFFER_SIZE,
    kI2C_TransferDefaultFlag);

if(result)
```

```
{
    return result;
}
```

### 19.2.2.2 Master Operation in interrupt transactional method

```
i2c_master_handle_t g_m_handle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* Gets a default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

I2C_MasterTransferCreateHandle(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle,
    i2c_master_callback, NULL);
I2C_MasterTransferNonBlocking(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle, &masterXfer);

/* Waits for a transfer to be completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

### 19.2.2.3 Master Operation in DMA transactional method

```
i2c_master_dma_handle_t g_m_dma_handle;
dma_handle_t dmaHandle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
```

## I2C Driver

```
{  
    g_MasterCompletionFlag = true;  
}  
}  
  
/* Gets the default configuration for the master. */  
I2C_MasterGetDefaultConfig(&masterConfig);  
  
/* Initializes the I2C master. */  
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);  
  
masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;  
masterXfer.direction = kI2C_Write;  
masterXfer.subaddress = NULL;  
masterXfer.subaddressSize = 0;  
masterXfer.data = txBuff;  
masterXfer.dataSize = BUFFER_SIZE;  
masterXfer.flags = kI2C_TransferDefaultFlag;  
  
DMAMGR_RequestChannel((dma_request_source_t)DMA_REQUEST_SRC, 0, &dmaHandle);  
  
I2C_MasterTransferCreateHandleDMA(EXAMPLE_I2C_MASTER_BASEADDR, &  
    g_m_dma_handle, i2c_master_callback, NULL, &dmaHandle);  
I2C_MasterTransferDMA(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_dma_handle, &masterXfer);  
  
/* Wait for transfer completed. */  
while (!g_MasterCompletionFlag)  
{  
}  
g_MasterCompletionFlag = false;
```

### 19.2.2.4 Slave Operation in functional method

```
i2c_slave_config_t slaveConfig;  
uint8_t status;  
status_t result = kStatus_Success;  
  
I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit  
    addressing mode*/  
slaveConfig.slaveAddr = 7-bit address  
slaveConfig.addressingMode = kI2C_Address7bit/  
    kI2C_RangeMatch;  
I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig, I2C_SLAVE_CLK);  
  
/* Waits for an address match. */  
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_SLAVE_BASEADDR)) & kI2C_AddressMatchFlag))  
{  
}  
  
/* A slave transmits; master is reading from the slave. */  
if (status & kI2C_TransferDirectionFlag)  
{  
    result = I2C_SlaveWriteBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, txBuff, BUFFER_SIZE);  
}  
else  
{  
    I2C_SlaveReadBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, rxBuff, BUFFER_SIZE);  
}  
  
return result;
```

### 19.2.2.5 Slave Operation in interrupt transactional method

```
i2c_slave_config_t slaveConfig;
i2c_slave_handle_t g_s_handle;
volatile bool g_SlaveCompletionFlag = false;

static void i2c_slave_callback(I2C_Type *base, i2c_slave_transfer_t *xfer, void *
                               userData)
{
    switch (xfer->event)
    {
        /* Transmit request */
        case kI2C_SlaveTransmitEvent:
            /* Update information for transmit process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Receives request */
        case kI2C_SlaveReceiveEvent:
            /* Update information for received process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Transfer is done */
        case kI2C_SlaveCompletionEvent:
            g_SlaveCompletionFlag = true;
            break;

        default:
            g_SlaveCompletionFlag = true;
            break;
    }
}

I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit
                                             addressing mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/
                           kI2C_RangeMatch;

I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig, I2C_SLAVE_CLK);

I2C_SlaveTransferCreateHandle(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
                             i2c_slave_callback, NULL);

I2C_SlaveTransferNonBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
                            kI2C_SlaveCompletionEvent);

/* Waits for a transfer to be completed. */
while (!g_SlaveCompletionFlag)
{
}
g_SlaveCompletionFlag = false;
```

## Data Structures

- struct [i2c\\_master\\_config\\_t](#)  
*I2C master user configuration.* [More...](#)
- struct [i2c\\_slave\\_config\\_t](#)  
*I2C slave user configuration.* [More...](#)
- struct [i2c\\_master\\_transfer\\_t](#)

## I2C Driver

- struct `i2c_master_handle_t`  
*I2C master handle structure.* [More...](#)
- struct `i2c_slave_transfer_t`  
*I2C slave transfer structure.* [More...](#)
- struct `i2c_slave_handle_t`  
*I2C slave handle structure.* [More...](#)

## Typedefs

- typedef void(\* `i2c_master_transfer_callback_t`)  
(I2C\_Type \*base, i2c\_master\_handle\_t \*handle,  
status\_t status, void \*userData)  
*I2C master transfer callback typedef.*
- typedef void(\* `i2c_slave_transfer_callback_t`)  
(I2C\_Type \*base, i2c\_slave\_transfer\_t \*xfer, void  
\*userData)  
*I2C slave transfer callback typedef.*

## Enumerations

- enum `_i2c_status` {  
  kStatus\_I2C\_Busy = MAKE\_STATUS(kStatusGroup\_I2C, 0),  
  kStatus\_I2C\_Idle = MAKE\_STATUS(kStatusGroup\_I2C, 1),  
  kStatus\_I2C\_Nak = MAKE\_STATUS(kStatusGroup\_I2C, 2),  
  kStatus\_I2C\_ArbitrationLost = MAKE\_STATUS(kStatusGroup\_I2C, 3),  
  kStatus\_I2C\_Timeout = MAKE\_STATUS(kStatusGroup\_I2C, 4),  
  kStatus\_I2C\_Addr\_Nak = MAKE\_STATUS(kStatusGroup\_I2C, 5) }  
*I2C status return codes.*
- enum `_i2c_flags` {  
  kI2C\_ReceiveNakFlag = I2C\_S\_RXAK\_MASK,  
  kI2C\_IntPendingFlag = I2C\_S\_IICIF\_MASK,  
  kI2C\_TransferDirectionFlag = I2C\_S\_SRW\_MASK,  
  kI2C\_RangeAddressMatchFlag = I2C\_S\_RAM\_MASK,  
  kI2C\_ArbitrationLostFlag = I2C\_S\_ARBL\_MASK,  
  kI2C\_BusBusyFlag = I2C\_S\_BUSY\_MASK,  
  kI2C\_AddressMatchFlag = I2C\_S\_IAAS\_MASK,  
  kI2C\_TransferCompleteFlag = I2C\_S\_TCF\_MASK,  
  kI2C\_StopDetectFlag = I2C\_FLT\_STOPF\_MASK << 8,  
  kI2C\_StartDetectFlag = I2C\_FLT\_STARTF\_MASK << 8 }  
*I2C peripheral flags.*
- enum `_i2c_interrupt_enable` {  
  kI2C\_GlobalInterruptEnable = I2C\_C1\_IICIE\_MASK,  
  kI2C\_StartStopDetectInterruptEnable = I2C\_FLT\_SSIE\_MASK }  
*I2C feature interrupt source.*
- enum `i2c_direction_t` {  
  kI2C\_Write = 0x0U,

- ```

kI2C_Read = 0x1U }

The direction of master and slave transfers.
• enum i2c_slave_address_mode_t {
    kI2C_Address7bit = 0x0U,
    kI2C_RangeMatch = 0X2U }

Addressing mode.
• enum _i2c_master_transfer_flags {
    kI2C_TransferDefaultFlag = 0x0U,
    kI2C_TransferNoStartFlag = 0x1U,
    kI2C_TransferRepeatedStartFlag = 0x2U,
    kI2C_TransferNoStopFlag = 0x4U }

I2C transfer control flag.
• enum i2c_slave_transfer_event_t {
    kI2C_SlaveAddressMatchEvent = 0x01U,
    kI2C_SlaveTransmitEvent = 0x02U,
    kI2C_SlaveReceiveEvent = 0x04U,
    kI2C_SlaveTransmitAckEvent = 0x08U,
    kI2C_SlaveStartEvent = 0x10U,
    kI2C_SlaveCompletionEvent = 0x20U,
    kI2C_SlaveGenaralcallEvent = 0x40U,
    kI2C_SlaveAllEvents }

Set of events sent to the callback for nonblocking slave transfers.

```

## Driver version

- #define FSL\_I2C\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 3))  
*I2C driver version 2.0.3.*

## Initialization and deinitialization

- void I2C\_MasterInit (I2C\_Type \*base, const i2c\_master\_config\_t \*masterConfig, uint32\_t srcClock\_Hz)  
*Initializes the I2C peripheral.*
- void I2C\_SlaveInit (I2C\_Type \*base, const i2c\_slave\_config\_t \*slaveConfig, uint32\_t srcClock\_Hz)  
*Initializes the I2C peripheral.*
- void I2C\_MasterDeinit (I2C\_Type \*base)  
*De-initializes the I2C master peripheral.*
- void I2C\_SlaveDeinit (I2C\_Type \*base)  
*De-initializes the I2C slave peripheral.*
- void I2C\_MasterGetDefaultConfig (i2c\_master\_config\_t \*masterConfig)  
*Sets the I2C master configuration structure to default values.*
- void I2C\_SlaveGetDefaultConfig (i2c\_slave\_config\_t \*slaveConfig)  
*Sets the I2C slave configuration structure to default values.*
- static void I2C\_Enable (I2C\_Type \*base, bool enable)  
*Enables or disables the I2C peripheral operation.*

## I2C Driver

### Status

- `uint32_t I2C_MasterGetStatusFlags (I2C_Type *base)`  
*Gets the I2C status flags.*
- `static uint32_t I2C_SlaveGetStatusFlags (I2C_Type *base)`  
*Gets the I2C status flags.*
- `static void I2C_MasterClearStatusFlags (I2C_Type *base, uint32_t statusMask)`  
*Clears the I2C status flag state.*
- `static void I2C_SlaveClearStatusFlags (I2C_Type *base, uint32_t statusMask)`  
*Clears the I2C status flag state.*

### Interrupts

- `void I2C_EnableInterrupts (I2C_Type *base, uint32_t mask)`  
*Enables I2C interrupt requests.*
- `void I2C_DisableInterrupts (I2C_Type *base, uint32_t mask)`  
*Disables I2C interrupt requests.*

### DMA Control

- `static void I2C_EnableDMA (I2C_Type *base, bool enable)`  
*Enables/disables the I2C DMA interrupt.*
- `static uint32_t I2C_GetDataRegAddr (I2C_Type *base)`  
*Gets the I2C tx/rx data register address.*

### Bus Operations

- `void I2C_MasterSetBaudRate (I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)`  
*Sets the I2C master transfer baud rate.*
- `status_t I2C_MasterStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)`  
*Sends a START on the I2C bus.*
- `status_t I2C_MasterStop (I2C_Type *base)`  
*Sends a STOP signal on the I2C bus.*
- `status_t I2C_MasterRepeatedStart (I2C_Type *base, uint8_t address, i2c_direction_t direction)`  
*Sends a REPEATED START on the I2C bus.*
- `status_t I2C_MasterWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize, uint32_t flags)`  
*Performs a polling send transaction on the I2C bus.*
- `status_t I2C_MasterReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize, uint32_t flags)`  
*Performs a polling receive transaction on the I2C bus.*
- `status_t I2C_SlaveWriteBlocking (I2C_Type *base, const uint8_t *txBuff, size_t txSize)`  
*Performs a polling send transaction on the I2C bus.*
- `void I2C_SlaveReadBlocking (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)`  
*Performs a polling receive transaction on the I2C bus.*
- `status_t I2C_MasterTransferBlocking (I2C_Type *base, i2c_master_transfer_t *xfer)`  
*Performs a master polling transfer on the I2C bus.*

## Transactional

- void `I2C_MasterTransferCreateHandle` (I2C\_Type \*base, i2c\_master\_handle\_t \*handle, `i2c_master_transfer_callback_t` callback, void \*userData)
 

*Initializes the I2C handle which is used in transactional functions.*
- status\_t `I2C_MasterTransferNonBlocking` (I2C\_Type \*base, i2c\_master\_handle\_t \*handle, `i2c_master_transfer_t` \*xfer)
 

*Performs a master interrupt non-blocking transfer on the I2C bus.*
- status\_t `I2C_MasterTransferGetCount` (I2C\_Type \*base, i2c\_master\_handle\_t \*handle, size\_t \*count)
 

*Gets the master transfer status during a interrupt non-blocking transfer.*
- void `I2C_MasterTransferAbort` (I2C\_Type \*base, i2c\_master\_handle\_t \*handle)
 

*Aborts an interrupt non-blocking transfer early.*
- void `I2C_MasterTransferHandleIRQ` (I2C\_Type \*base, void \*i2cHandle)
 

*Master interrupt handler.*
- void `I2C_SlaveTransferCreateHandle` (I2C\_Type \*base, i2c\_slave\_handle\_t \*handle, `i2c_slave_transfer_callback_t` callback, void \*userData)
 

*Initializes the I2C handle which is used in transactional functions.*
- status\_t `I2C_SlaveTransferNonBlocking` (I2C\_Type \*base, i2c\_slave\_handle\_t \*handle, uint32\_t eventMask)
 

*Starts accepting slave transfers.*
- void `I2C_SlaveTransferAbort` (I2C\_Type \*base, i2c\_slave\_handle\_t \*handle)
 

*Aborts the slave transfer.*
- status\_t `I2C_SlaveTransferGetCount` (I2C\_Type \*base, i2c\_slave\_handle\_t \*handle, size\_t \*count)
 

*Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.*
- void `I2C_SlaveTransferHandleIRQ` (I2C\_Type \*base, void \*i2cHandle)
 

*Slave interrupt handler.*

### 19.2.3 Data Structure Documentation

#### 19.2.3.1 struct i2c\_master\_config\_t

##### Data Fields

- bool `enableMaster`

*Enables the I2C peripheral at initialization time.*
- bool `enableStopHold`

*Controls the stop hold enable.*
- uint32\_t `baudRate_Bps`

*Baud rate configuration of I2C peripheral.*
- uint8\_t `glitchFilterWidth`

*Controls the width of the glitch.*

### 19.2.3.1.0.51 Field Documentation

19.2.3.1.0.51.1 `bool i2c_master_config_t::enableMaster`

19.2.3.1.0.51.2 `bool i2c_master_config_t::enableStopHold`

19.2.3.1.0.51.3 `uint32_t i2c_master_config_t::baudRate_Bps`

19.2.3.1.0.51.4 `uint8_t i2c_master_config_t::glitchFilterWidth`

### 19.2.3.2 `struct i2c_slave_config_t`

#### Data Fields

- `bool enableSlave`

*Enables the I2C peripheral at initialization time.*

- `bool enableGeneralCall`

*Enables the general call addressing mode.*

- `bool enableWakeUp`

*Enables/disables waking up MCU from low-power mode.*

- `bool enableBaudRateCtl`

*Enables/disables independent slave baud rate on SCL in very fast I2C modes.*

- `uint16_t slaveAddress`

*A slave address configuration.*

- `uint16_t upperAddress`

*A maximum boundary slave address used in a range matching mode.*

- `i2c_slave_address_mode_t addressingMode`

*An addressing mode configuration of i2c\_slave\_address\_mode\_config\_t.*

- `uint32_t sclStopHoldTime_ns`

*the delay from the rising edge of SCL (I2C clock) to the rising edge of SDA (I2C data) while SCL is high (stop condition), SDA hold time and SCL start hold time are also configured according to the SCL stop hold time.*

### 19.2.3.2.0.52 Field Documentation

19.2.3.2.0.52.1 `bool i2c_slave_config_t::enableSlave`

19.2.3.2.0.52.2 `bool i2c_slave_config_t::enableGeneralCall`

19.2.3.2.0.52.3 `bool i2c_slave_config_t::enableWakeUp`

19.2.3.2.0.52.4 `bool i2c_slave_config_t::enableBaudRateCtl`

19.2.3.2.0.52.5 `uint16_t i2c_slave_config_t::slaveAddress`

19.2.3.2.0.52.6 `uint16_t i2c_slave_config_t::upperAddress`

19.2.3.2.0.52.7 `i2c_slave_address_mode_t i2c_slave_config_t::addressingMode`

19.2.3.2.0.52.8 `uint32_t i2c_slave_config_t::sclStopHoldTime_ns`

### 19.2.3.3 struct i2c\_master\_transfer\_t

#### Data Fields

- `uint32_t flags`  
*A transfer flag which controls the transfer.*
- `uint8_t slaveAddress`  
*7-bit slave address.*
- `i2c_direction_t direction`  
*A transfer direction, read or write.*
- `uint32_t subaddress`  
*A sub address.*
- `uint8_t subaddressSize`  
*A size of the command buffer.*
- `uint8_t *volatile data`  
*A transfer buffer.*
- `volatile size_t dataSize`  
*A transfer size.*

### 19.2.3.3.0.53 Field Documentation

19.2.3.3.0.53.1 `uint32_t i2c_master_transfer_t::flags`

19.2.3.3.0.53.2 `uint8_t i2c_master_transfer_t::slaveAddress`

19.2.3.3.0.53.3 `i2c_direction_t i2c_master_transfer_t::direction`

19.2.3.3.0.53.4 `uint32_t i2c_master_transfer_t::subaddress`

Transferred MSB first.

## I2C Driver

**19.2.3.3.0.53.5 uint8\_t i2c\_master\_transfer\_t::subaddressSize**

**19.2.3.3.0.53.6 uint8\_t\* volatile i2c\_master\_transfer\_t::data**

**19.2.3.3.0.53.7 volatile size\_t i2c\_master\_transfer\_t::dataSize**

### 19.2.3.4 struct \_i2c\_master\_handle

I2C master handle typedef.

#### Data Fields

- [i2c\\_master\\_transfer\\_t transfer](#)  
*I2C master transfer copy.*
- [size\\_t transferSize](#)  
*Total bytes to be transferred.*
- [uint8\\_t state](#)  
*A transfer state maintained during transfer.*
- [i2c\\_master\\_transfer\\_callback\\_t completionCallback](#)  
*A callback function called when the transfer is finished.*
- [void \\* userData](#)  
*A callback parameter passed to the callback function.*

### 19.2.3.4.0.54 Field Documentation

**19.2.3.4.0.54.1 i2c\_master\_transfer\_t i2c\_master\_handle\_t::transfer**

**19.2.3.4.0.54.2 size\_t i2c\_master\_handle\_t::transferSize**

**19.2.3.4.0.54.3 uint8\_t i2c\_master\_handle\_t::state**

**19.2.3.4.0.54.4 i2c\_master\_transfer\_callback\_t i2c\_master\_handle\_t::completionCallback**

**19.2.3.4.0.54.5 void\* i2c\_master\_handle\_t::userData**

### 19.2.3.5 struct i2c\_slave\_transfer\_t

#### Data Fields

- [i2c\\_slave\\_transfer\\_event\\_t event](#)  
*A reason that the callback is invoked.*
- [uint8\\_t \\*volatile data](#)  
*A transfer buffer.*
- [volatile size\\_t dataSize](#)  
*A transfer size.*
- [status\\_t completionStatus](#)  
*Success or error code describing how the transfer completed.*
- [size\\_t transferredCount](#)  
*A number of bytes actually transferred since the start or since the last repeated start.*

### 19.2.3.5.0.55 Field Documentation

19.2.3.5.0.55.1 `i2c_slave_transfer_event_t i2c_slave_transfer_t::event`

19.2.3.5.0.55.2 `uint8_t* volatile i2c_slave_transfer_t::data`

19.2.3.5.0.55.3 `volatile size_t i2c_slave_transfer_t::dataSize`

19.2.3.5.0.55.4 `status_t i2c_slave_transfer_t::completionStatus`

Only applies for [kI2C\\_SlaveCompletionEvent](#).

19.2.3.5.0.55.5 `size_t i2c_slave_transfer_t::transferredCount`

### 19.2.3.6 struct \_i2c\_slave\_handle

I2C slave handle typedef.

#### Data Fields

- volatile bool [isBusy](#)  
*Indicates whether a transfer is busy.*
- [i2c\\_slave\\_transfer\\_t transfer](#)  
*I2C slave transfer copy.*
- uint32\_t [eventMask](#)  
*A mask of enabled events.*
- [i2c\\_slave\\_transfer\\_callback\\_t callback](#)  
*A callback function called at the transfer event.*
- void \* [userData](#)  
*A callback parameter passed to the callback.*

## I2C Driver

### 19.2.3.6.0.56 Field Documentation

19.2.3.6.0.56.1 `volatile bool i2c_slave_handle_t::isBusy`

19.2.3.6.0.56.2 `i2c_slave_transfer_t i2c_slave_handle_t::transfer`

19.2.3.6.0.56.3 `uint32_t i2c_slave_handle_t::eventMask`

19.2.3.6.0.56.4 `i2c_slave_transfer_callback_t i2c_slave_handle_t::callback`

19.2.3.6.0.56.5 `void* i2c_slave_handle_t::userData`

### 19.2.4 Macro Definition Documentation

19.2.4.1 `#define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))`

### 19.2.5 Typedef Documentation

19.2.5.1 `typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *userData)`

19.2.5.2 `typedef void(* i2c_slave_transfer_callback_t)(I2C_Type *base, i2c_slave_transfer_t *xfer, void *userData)`

### 19.2.6 Enumeration Type Documentation

#### 19.2.6.1 enum \_i2c\_status

Enumerator

*kStatus\_I2C\_Busy* I2C is busy with current transfer.

*kStatus\_I2C\_Idle* Bus is Idle.

*kStatus\_I2C\_Nak* NAK received during transfer.

*kStatus\_I2C\_ArbitrationLost* Arbitration lost during transfer.

*kStatus\_I2C\_Timeout* Wait event timeout.

*kStatus\_I2C\_Addr\_Nak* NAK received during the address probe.

#### 19.2.6.2 enum \_i2c\_flags

The following status register flags can be cleared:

- [kI2C\\_ArbitrationLostFlag](#)
- [kI2C\\_IntPendingFlag](#)
- [kI2C\\_StartDetectFlag](#)
- [kI2C\\_StopDetectFlag](#)

## Note

These enumerations are meant to be OR'd together to form a bit mask.

## Enumerator

- kI2C\_ReceiveNakFlag* I2C receive NAK flag.
- kI2C\_IntPendingFlag* I2C interrupt pending flag.
- kI2C\_TransferDirectionFlag* I2C transfer direction flag.
- kI2C\_RangeAddressMatchFlag* I2C range address match flag.
- kI2C\_ArbitrationLostFlag* I2C arbitration lost flag.
- kI2C\_BusBusyFlag* I2C bus busy flag.
- kI2C\_AddressMatchFlag* I2C address match flag.
- kI2C\_TransferCompleteFlag* I2C transfer complete flag.
- kI2C\_StopDetectFlag* I2C stop detect flag.
- kI2C\_StartDetectFlag* I2C start detect flag.

**19.2.6.3 enum \_i2c\_interrupt\_enable**

## Enumerator

- kI2C\_GlobalInterruptEnable* I2C global interrupt.
- kI2C\_StartStopDetectInterruptEnable* I2C start&stop detect interrupt.

**19.2.6.4 enum i2c\_direction\_t**

## Enumerator

- kI2C\_Write* Master transmits to the slave.
- kI2C\_Read* Master receives from the slave.

**19.2.6.5 enum i2c\_slave\_address\_mode\_t**

## Enumerator

- kI2C\_Address7bit* 7-bit addressing mode.
- kI2C\_RangeMatch* Range address match addressing mode.

**19.2.6.6 enum \_i2c\_master\_transfer\_flags**

## Enumerator

- kI2C\_TransferDefaultFlag* A transfer starts with a start signal, stops with a stop signal.

## I2C Driver

***kI2C\_TransferNoStartFlag*** A transfer starts without a start signal.

***kI2C\_TransferRepeatedStartFlag*** A transfer starts with a repeated start signal.

***kI2C\_TransferNoStopFlag*** A transfer ends without a stop signal.

### 19.2.6.7 enum i2c\_slave\_transfer\_event\_t

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to [I2C\\_SlaveTransferNonBlocking\(\)](#) to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note

These enumerations are meant to be OR'd together to form a bit mask of events.

Enumerator

***kI2C\_SlaveAddressMatchEvent*** Received the slave address after a start or repeated start.

***kI2C\_SlaveTransmitEvent*** A callback is requested to provide data to transmit (slave-transmitter role).

***kI2C\_SlaveReceiveEvent*** A callback is requested to provide a buffer in which to place received data (slave-receiver role).

***kI2C\_SlaveTransmitAckEvent*** A callback needs to either transmit an ACK or NACK.

***kI2C\_SlaveStartEvent*** A start/repeated start was detected.

***kI2C\_SlaveCompletionEvent*** A stop was detected or finished transfer, completing the transfer.

***kI2C\_SlaveGeneralCallEvent*** Received the general call address after a start or repeated start.

***kI2C\_SlaveAllEvents*** A bit mask of all available events.

### 19.2.7 Function Documentation

#### 19.2.7.1 void I2C\_MasterInit ( *I2C\_Type* \* *base*, *const i2c\_master\_config\_t* \* *masterConfig*, *uint32\_t srcClock\_Hz* )

Call this API to ungate the I2C clock and configure the I2C with master configuration.

Note

This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can be custom filled or it can be set with default values by using the [I2C\\_MasterGetDefaultConfig\(\)](#). After calling this API, the master is ready to transfer. This is an example.

```
* i2c_master_config_t config = {  
* .enableMaster = true,  
* .enableStopHold = false,  
* .highDrive = false,  
* .baudRate_Bps = 100000,
```

```

* .glitchFilterWidth = 0
* };
* I2C_MasterInit(I2C0, &config, 12000000U);
*

```

## Parameters

|                     |                                                 |
|---------------------|-------------------------------------------------|
| <i>base</i>         | I2C base pointer                                |
| <i>masterConfig</i> | A pointer to the master configuration structure |
| <i>srcClock_Hz</i>  | I2C peripheral clock frequency in Hz            |

### 19.2.7.2 void I2C\_SlaveInit ( I2C\_Type \* *base*, const i2c\_slave\_config\_t \* *slaveConfig*, uint32\_t *srcClock\_Hz* )

Call this API to ungate the I2C clock and initialize the I2C with the slave configuration.

## Note

This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can partly be set with default values by [I2C\\_SlaveGetDefaultConfig\(\)](#) or it can be custom filled by the user. This is an example.

```

* i2c_slave_config_t config = {
*   .enableSlave = true,
*   .enableGeneralCall = false,
*   .addressingMode = kI2C_Address7bit,
*   .slaveAddress = 0x1DU,
*   .enableWakeUp = false,
*   .enablehighDrive = false,
*   .enableBaudRateCtl = false,
*   .sclStopHoldTime_ns = 4000
* };
* I2C_SlaveInit(I2C0, &config, 12000000U);
*

```

## Parameters

|                    |                                                |
|--------------------|------------------------------------------------|
| <i>base</i>        | I2C base pointer                               |
| <i>slaveConfig</i> | A pointer to the slave configuration structure |
| <i>srcClock_Hz</i> | I2C peripheral clock frequency in Hz           |

### 19.2.7.3 void I2C\_MasterDeinit ( I2C\_Type \* *base* )

Call this API to gate the I2C clock. The I2C master module can't work unless the I2C\_MasterInit is called.

## I2C Driver

Parameters

|             |                  |
|-------------|------------------|
| <i>base</i> | I2C base pointer |
|-------------|------------------|

### 19.2.7.4 void I2C\_SlaveDeinit ( I2C\_Type \* *base* )

Calling this API gates the I2C clock. The I2C slave module can't work unless the I2C\_SlaveInit is called to enable the clock.

Parameters

|             |                  |
|-------------|------------------|
| <i>base</i> | I2C base pointer |
|-------------|------------------|

### 19.2.7.5 void I2C\_MasterGetDefaultConfig ( i2c\_master\_config\_t \* *masterConfig* )

The purpose of this API is to get the configuration structure initialized for use in the I2C\_MasterConfigure(). Use the initialized structure unchanged in the I2C\_MasterConfigure() or modify the structure before calling the I2C\_MasterConfigure(). This is an example.

```
* i2c_master_config_t config;
* I2C_MasterGetDefaultConfig(&config);
*
```

Parameters

|                     |                                                  |
|---------------------|--------------------------------------------------|
| <i>masterConfig</i> | A pointer to the master configuration structure. |
|---------------------|--------------------------------------------------|

### 19.2.7.6 void I2C\_SlaveGetDefaultConfig ( i2c\_slave\_config\_t \* *slaveConfig* )

The purpose of this API is to get the configuration structure initialized for use in the I2C\_SlaveConfigure(). Modify fields of the structure before calling the I2C\_SlaveConfigure(). This is an example.

```
* i2c_slave_config_t config;
* I2C_SlaveGetDefaultConfig(&config);
*
```

Parameters

|                    |                                                 |
|--------------------|-------------------------------------------------|
| <i>slaveConfig</i> | A pointer to the slave configuration structure. |
|--------------------|-------------------------------------------------|

### 19.2.7.7 static void I2C\_Enable ( I2C\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

|               |                                                      |
|---------------|------------------------------------------------------|
| <i>base</i>   | I2C base pointer                                     |
| <i>enable</i> | Pass true to enable and false to disable the module. |

### 19.2.7.8 uint32\_t I2C\_MasterGetStatusFlags ( I2C\_Type \* *base* )

Parameters

|             |                  |
|-------------|------------------|
| <i>base</i> | I2C base pointer |
|-------------|------------------|

Returns

status flag, use status flag to AND [\\_i2c\\_flags](#) to get the related status.

### 19.2.7.9 static uint32\_t I2C\_SlaveGetStatusFlags ( I2C\_Type \* *base* ) [inline], [static]

Parameters

|             |                  |
|-------------|------------------|
| <i>base</i> | I2C base pointer |
|-------------|------------------|

Returns

status flag, use status flag to AND [\\_i2c\\_flags](#) to get the related status.

### 19.2.7.10 static void I2C\_MasterClearStatusFlags ( I2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared kI2C\_ArbitrationLostFlag and kI2C\_IntPendingFlag.

## I2C Driver

Parameters

|                   |                                                                                                                                                                                                                                                                                                                                |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | I2C base pointer                                                                                                                                                                                                                                                                                                               |
| <i>statusMask</i> | The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values: <ul style="list-style-type: none"><li>• kI2C_StartDetectFlag (if available)</li><li>• kI2C_StopDetectFlag (if available)</li><li>• kI2C_ArbitrationLostFlag</li><li>• kI2C_IntPendingFlagFlag</li></ul> |

### 19.2.7.11 static void I2C\_SlaveClearStatusFlags ( I2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared kI2C\_ArbitrationLostFlag and kI2C\_IntPendingFlag

Parameters

|                   |                                                                                                                                                                                                                                                                                                                                |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | I2C base pointer                                                                                                                                                                                                                                                                                                               |
| <i>statusMask</i> | The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values: <ul style="list-style-type: none"><li>• kI2C_StartDetectFlag (if available)</li><li>• kI2C_StopDetectFlag (if available)</li><li>• kI2C_ArbitrationLostFlag</li><li>• kI2C_IntPendingFlagFlag</li></ul> |

### 19.2.7.12 void I2C\_EnableInterrupts ( I2C\_Type \* *base*, uint32\_t *mask* )

Parameters

|             |                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | I2C base pointer                                                                                                                                                                                                                                                                     |
| <i>mask</i> | interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"><li>• kI2C_GlobalInterruptEnable</li><li>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</li><li>• kI2C_SdaTimeoutInterruptEnable</li></ul> |

### 19.2.7.13 void I2C\_DisableInterrupts ( I2C\_Type \* *base*, uint32\_t *mask* )

Parameters

|             |                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | I2C base pointer                                                                                                                                                                                                                                                                     |
| <i>mask</i> | interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"><li>• kI2C_GlobalInterruptEnable</li><li>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</li><li>• kI2C_SdaTimeoutInterruptEnable</li></ul> |

#### 19.2.7.14 static void I2C\_EnableDMA ( I2C\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

|               |                                  |
|---------------|----------------------------------|
| <i>base</i>   | I2C base pointer                 |
| <i>enable</i> | true to enable, false to disable |

#### 19.2.7.15 static uint32\_t I2C\_GetDataRegAddr ( I2C\_Type \* *base* ) [inline], [static]

This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

|             |                  |
|-------------|------------------|
| <i>base</i> | I2C base pointer |
|-------------|------------------|

Returns

data register address

#### 19.2.7.16 void I2C\_MasterSetBaudRate ( I2C\_Type \* *base*, uint32\_t *baudRate\_Bps*, uint32\_t *srcClock\_Hz* )

Parameters

## I2C Driver

|                     |                            |
|---------------------|----------------------------|
| <i>base</i>         | I2C base pointer           |
| <i>baudRate_Bps</i> | the baud rate value in bps |
| <i>srcClock_Hz</i>  | Source clock               |

### 19.2.7.17 status\_t I2C\_MasterStart ( I2C\_Type \* *base*, uint8\_t *address*, i2c\_direction\_t *direction* )

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

|                  |                                               |
|------------------|-----------------------------------------------|
| <i>base</i>      | I2C peripheral base pointer                   |
| <i>address</i>   | 7-bit slave device address.                   |
| <i>direction</i> | Master transfer directions(transmit/receive). |

Return values

|                         |                                     |
|-------------------------|-------------------------------------|
| <i>kStatus_Success</i>  | Successfully send the start signal. |
| <i>kStatus_I2C_Busy</i> | Current bus is busy.                |

### 19.2.7.18 status\_t I2C\_MasterStop ( I2C\_Type \* *base* )

Return values

|                            |                                    |
|----------------------------|------------------------------------|
| <i>kStatus_Success</i>     | Successfully send the stop signal. |
| <i>kStatus_I2C_Timeout</i> | Send stop signal failed, timeout.  |

### 19.2.7.19 status\_t I2C\_MasterRepeatedStart ( I2C\_Type \* *base*, uint8\_t *address*, i2c\_direction\_t *direction* )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | I2C peripheral base pointer |
|-------------|-----------------------------|

|                  |                                               |
|------------------|-----------------------------------------------|
| <i>address</i>   | 7-bit slave device address.                   |
| <i>direction</i> | Master transfer directions(transmit/receive). |

Return values

|                         |                                                             |
|-------------------------|-------------------------------------------------------------|
| <i>kStatus_Success</i>  | Successfully send the start signal.                         |
| <i>kStatus_I2C_Busy</i> | Current bus is busy but not occupied by current I2C master. |

### 19.2.7.20 **status\_t I2C\_MasterWriteBlocking ( I2C\_Type \* *base*, const uint8\_t \* *txBuff*, size\_t *txSize*, uint32\_t *flags* )**

Parameters

|               |                                                                                                                                                       |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | The I2C peripheral base pointer.                                                                                                                      |
| <i>txBuff</i> | The pointer to the data to be transferred.                                                                                                            |
| <i>txSize</i> | The length in bytes of the data to be transferred.                                                                                                    |
| <i>flags</i>  | Transfer control flag to decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop. |

Return values

|                                     |                                              |
|-------------------------------------|----------------------------------------------|
| <i>kStatus_Success</i>              | Successfully complete the data transmission. |
| <i>kStatus_I2C_Arbitration-Lost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>              | Transfer error, receive NAK during transfer. |

### 19.2.7.21 **status\_t I2C\_MasterReadBlocking ( I2C\_Type \* *base*, uint8\_t \* *rxBuff*, size\_t *rxSize*, uint32\_t *flags* )**

Note

The I2C\_MasterReadBlocking function stops the bus before reading the final byte. Without stopping the bus prior for the final read, the bus issues another read, resulting in garbage data being read into the data register.

## I2C Driver

Parameters

|               |                                                                                                                                                       |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base pointer.                                                                                                                          |
| <i>rxBuff</i> | The pointer to the data to store the received data.                                                                                                   |
| <i>rxSize</i> | The length in bytes of the data to be received.                                                                                                       |
| <i>flags</i>  | Transfer control flag to decide whether need to send a stop, use kI2C_TransferDefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop. |

Return values

|                            |                                              |
|----------------------------|----------------------------------------------|
| <i>kStatus_Success</i>     | Successfully complete the data transmission. |
| <i>kStatus_I2C_Timeout</i> | Send stop signal failed, timeout.            |

### 19.2.7.22 **status\_t I2C\_SlaveWriteBlocking ( I2C\_Type \* *base*, const uint8\_t \* *txBuff*, size\_t *txSize* )**

Parameters

|               |                                                    |
|---------------|----------------------------------------------------|
| <i>base</i>   | The I2C peripheral base pointer.                   |
| <i>txBuff</i> | The pointer to the data to be transferred.         |
| <i>txSize</i> | The length in bytes of the data to be transferred. |

Return values

|                                     |                                              |
|-------------------------------------|----------------------------------------------|
| <i>kStatus_Success</i>              | Successfully complete the data transmission. |
| <i>kStatus_I2C_Arbitration-Lost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>              | Transfer error, receive NAK during transfer. |

### 19.2.7.23 **void I2C\_SlaveReadBlocking ( I2C\_Type \* *base*, uint8\_t \* *rxBuff*, size\_t *rxSize* )**

Parameters

|               |                                                     |
|---------------|-----------------------------------------------------|
| <i>base</i>   | I2C peripheral base pointer.                        |
| <i>rxBuff</i> | The pointer to the data to store the received data. |
| <i>rxSize</i> | The length in bytes of the data to be received.     |

#### 19.2.7.24 **status\_t I2C\_MasterTransferBlocking ( I2C\_Type \* *base*, i2c\_master\_transfer\_t \* *xfer* )**

Note

The API does not return until the transfer succeeds or fails due to arbitration lost or receiving a NAK.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | I2C peripheral base address.       |
| <i>xfer</i> | Pointer to the transfer structure. |

Return values

|                                     |                                              |
|-------------------------------------|----------------------------------------------|
| <i>kStatus_Success</i>              | Successfully complete the data transmission. |
| <i>kStatus_I2C_Busy</i>             | Previous transmission still not finished.    |
| <i>kStatus_I2C_Timeout</i>          | Transfer error, wait signal timeout.         |
| <i>kStatus_I2C_Arbitration-Lost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>              | Transfer error, receive NAK during transfer. |

#### 19.2.7.25 **void I2C\_MasterTransferCreateHandle ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle*, i2c\_master\_transfer\_callback\_t *callback*, void \* *userData* )**

Parameters

|                 |                                                                       |
|-----------------|-----------------------------------------------------------------------|
| <i>base</i>     | I2C base pointer.                                                     |
| <i>handle</i>   | pointer to i2c_master_handle_t structure to store the transfer state. |
| <i>callback</i> | pointer to user callback function.                                    |

## I2C Driver

|                 |                                                 |
|-----------------|-------------------------------------------------|
| <i>userData</i> | user parameter passed to the callback function. |
|-----------------|-------------------------------------------------|

### 19.2.7.26 status\_t I2C\_MasterTransferNonBlocking ( *I2C\_Type \* base*, *i2c\_master\_handle\_t \* handle*, *i2c\_master\_transfer\_t \* xfer* )

Note

Calling the API returns immediately after transfer initiates. The user needs to call I2C\_MasterGetTransferCount to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus\_I2C\_Busy, the transfer is finished.

Parameters

|               |                                                                           |
|---------------|---------------------------------------------------------------------------|
| <i>base</i>   | I2C base pointer.                                                         |
| <i>handle</i> | pointer to i2c_master_handle_t structure which stores the transfer state. |
| <i>xfer</i>   | pointer to <a href="#">i2c_master_transfer_t</a> structure.               |

Return values

|                            |                                           |
|----------------------------|-------------------------------------------|
| <i>kStatus_Success</i>     | Successfully start the data transmission. |
| <i>kStatus_I2C_Busy</i>    | Previous transmission still not finished. |
| <i>kStatus_I2C_Timeout</i> | Transfer error, wait signal timeout.      |

### 19.2.7.27 status\_t I2C\_MasterTransferGetCount ( *I2C\_Type \* base*, *i2c\_master\_handle\_t \* handle*, *size\_t \* count* )

Parameters

|               |                                                                           |
|---------------|---------------------------------------------------------------------------|
| <i>base</i>   | I2C base pointer.                                                         |
| <i>handle</i> | pointer to i2c_master_handle_t structure which stores the transfer state. |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction.       |

Return values

|                                |                                |
|--------------------------------|--------------------------------|
| <i>kStatus_InvalidArgument</i> | count is Invalid.              |
| <i>kStatus_Success</i>         | Successfully return the count. |

### **19.2.7.28 void I2C\_MasterTransferAbort ( I2C\_Type \* *base*, i2c\_master\_handle\_t \* *handle* )**

Note

This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

|               |                                                                          |
|---------------|--------------------------------------------------------------------------|
| <i>base</i>   | I2C base pointer.                                                        |
| <i>handle</i> | pointer to i2c_master_handle_t structure which stores the transfer state |

### **19.2.7.29 void I2C\_MasterTransferHandleIRQ ( I2C\_Type \* *base*, void \* *i2cHandle* )**

Parameters

|                  |                                           |
|------------------|-------------------------------------------|
| <i>base</i>      | I2C base pointer.                         |
| <i>i2cHandle</i> | pointer to i2c_master_handle_t structure. |

### **19.2.7.30 void I2C\_SlaveTransferCreateHandle ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, i2c\_slave\_transfer\_callback\_t *callback*, void \* *userData* )**

Parameters

|                 |                                                                      |
|-----------------|----------------------------------------------------------------------|
| <i>base</i>     | I2C base pointer.                                                    |
| <i>handle</i>   | pointer to i2c_slave_handle_t structure to store the transfer state. |
| <i>callback</i> | pointer to user callback function.                                   |
| <i>userData</i> | user parameter passed to the callback function.                      |

### **19.2.7.31 status\_t I2C\_SlaveTransferNonBlocking ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, uint32\_t *eventMask* )**

Call this API after calling the [I2C\\_SlaveInit\(\)](#) and [I2C\\_SlaveTransferCreateHandle\(\)](#) to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to [I2C\\_SlaveTransferCreateHandle\(\)](#). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of [i2c\\_slave\\_transfer\\_event\\_t](#) enumerators for the events you wish to receive. The [k\\_I2C\\_SlaveTransmitEvent](#) and [#kLPI2C\\_SlaveReceiveEvent](#) events are always enabled and do not need to

## I2C Driver

be included in the mask. Alternatively, pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the [kI2C\\_SlaveAllEvents](#) constant is provided as a convenient way to enable all events.

Parameters

|                  |                                                                                                                                                                                                                                                                                                    |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>      | The I2C peripheral base address.                                                                                                                                                                                                                                                                   |
| <i>handle</i>    | Pointer to #i2c_slave_handle_t structure which stores the transfer state.                                                                                                                                                                                                                          |
| <i>eventMask</i> | Bit mask formed by OR'ing together <a href="#">i2c_slave_transfer_event_t</a> enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and <a href="#">kI2C_SlaveAllEvents</a> to enable all events. |

Return values

|                                  |                                                           |
|----------------------------------|-----------------------------------------------------------|
| # <i>kStatus_Success</i>         | Slave transfers were successfully started.                |
| <a href="#">kStatus_I2C_Busy</a> | Slave transfers have already been started on this handle. |

### 19.2.7.32 void I2C\_SlaveTransferAbort ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle* )

Note

This API can be called at any time to stop slave for handling the bus events.

Parameters

|               |                                                                          |
|---------------|--------------------------------------------------------------------------|
| <i>base</i>   | I2C base pointer.                                                        |
| <i>handle</i> | pointer to i2c_slave_handle_t structure which stores the transfer state. |

### 19.2.7.33 status\_t I2C\_SlaveTransferGetCount ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

|               |                                                                     |
|---------------|---------------------------------------------------------------------|
| <i>base</i>   | I2C base pointer.                                                   |
| <i>handle</i> | pointer to i2c_slave_handle_t structure.                            |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction. |

Return values

|                                |                                |
|--------------------------------|--------------------------------|
| <i>kStatus_InvalidArgument</i> | count is Invalid.              |
| <i>kStatus_Success</i>         | Successfully return the count. |

#### 19.2.7.34 void I2C\_SlaveTransferHandleIRQ ( **I2C\_Type \* base**, **void \* i2cHandle** )

Parameters

|                  |                                                                         |
|------------------|-------------------------------------------------------------------------|
| <i>base</i>      | I2C base pointer.                                                       |
| <i>i2cHandle</i> | pointer to i2c_slave_handle_t structure which stores the transfer state |

## I2C eDMA Driver

### 19.3 I2C eDMA Driver

#### 19.3.1 Overview

#### Data Structures

- struct [i2c\\_master\\_edma\\_handle\\_t](#)  
*I2C master eDMA transfer structure.* [More...](#)

#### Typedefs

- [typedef void\(\\* i2c\\_master\\_edma\\_transfer\\_callback\\_t \)\(I2C\\_Type \\*base, i2c\\_master\\_edma\\_handle\\_t \\*handle, status\\_t status, void \\*userData\)](#)  
*I2C master eDMA transfer callback typedef.*

#### I2C Block eDMA Transfer Operation

- [void I2C\\_MasterCreateEDMAHandle \(I2C\\_Type \\*base, i2c\\_master\\_edma\\_handle\\_t \\*handle, \[i2c\\\_master\\\_edma\\\_transfer\\\_callback\\\_t\]\(#\) callback, void \\*userData, \[edma\\\_handle\\\_t\]\(#\) \\*edmaHandle\)](#)  
*Initializes the I2C handle which is used in transactional functions.*
- [status\\_t I2C\\_MasterTransferEDMA \(I2C\\_Type \\*base, i2c\\_master\\_edma\\_handle\\_t \\*handle, \[i2c\\\_master\\\_transfer\\\_t\]\(#\) \\*xfer\)](#)  
*Performs a master eDMA non-blocking transfer on the I2C bus.*
- [status\\_t I2C\\_MasterTransferGetCountEDMA \(I2C\\_Type \\*base, i2c\\_master\\_edma\\_handle\\_t \\*handle, size\\_t \\*count\)](#)  
*Gets a master transfer status during the eDMA non-blocking transfer.*
- [void I2C\\_MasterTransferAbortEDMA \(I2C\\_Type \\*base, i2c\\_master\\_edma\\_handle\\_t \\*handle\)](#)  
*Aborts a master eDMA non-blocking transfer early.*

#### 19.3.2 Data Structure Documentation

##### 19.3.2.1 struct \_i2c\_master\_edma\_handle

I2C master eDMA handle typedef.

#### Data Fields

- [i2c\\_master\\_transfer\\_t transfer](#)  
*I2C master transfer structure.*
- [size\\_t transferSize](#)  
*Total bytes to be transferred.*
- [uint8\\_t nbytes](#)  
*eDMA minor byte transfer count initially configured.*
- [uint8\\_t state](#)

*I2C master transfer status.*

- `edma_handle_t * dmaHandle`

*The eDMA handler used.*

- `i2c_master_edma_transfer_callback_t completionCallback`

*A callback function called after the eDMA transfer is finished.*

- `void * userData`

*A callback parameter passed to the callback function.*

### 19.3.2.1.0.57 Field Documentation

19.3.2.1.0.57.1 `i2c_master_transfer_t i2c_master_edma_handle_t::transfer`

19.3.2.1.0.57.2 `size_t i2c_master_edma_handle_t::transferSize`

19.3.2.1.0.57.3 `uint8_t i2c_master_edma_handle_t::nbytes`

19.3.2.1.0.57.4 `uint8_t i2c_master_edma_handle_t::state`

19.3.2.1.0.57.5 `edma_handle_t* i2c_master_edma_handle_t::dmaHandle`

19.3.2.1.0.57.6 `i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completionCallback`

19.3.2.1.0.57.7 `void* i2c_master_edma_handle_t::userData`

### 19.3.3 Typedef Documentation

19.3.3.1 `typedef void(* i2c_master_edma_transfer_callback_t)(I2C_Type *base, i2c_master_edma_handle_t *handle, status_t status, void *userData)`

### 19.3.4 Function Documentation

19.3.4.1 `void I2C_MasterCreateEDMAHandle ( I2C_Type * base, i2c_master_edma_handle_t * handle, i2c_master_edma_transfer_callback_t callback, void * userData, edma_handle_t * edmaHandle )`

Parameters

|                       |                                                                   |
|-----------------------|-------------------------------------------------------------------|
| <code>base</code>     | I2C peripheral base address.                                      |
| <code>handle</code>   | A pointer to the <code>i2c_master_edma_handle_t</code> structure. |
| <code>callback</code> | A pointer to the user callback function.                          |

## I2C eDMA Driver

|                   |                                                   |
|-------------------|---------------------------------------------------|
| <i>userData</i>   | A user parameter passed to the callback function. |
| <i>edmaHandle</i> | eDMA handle pointer.                              |

**19.3.4.2 status\_t I2C\_MasterTransferEDMA ( *I2C\_Type* \* *base*, *i2c\_master\_edma\_handle\_t* \* *handle*, *i2c\_master\_transfer\_t* \* *xfer* )**

Parameters

|               |                                                                       |
|---------------|-----------------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address.                                          |
| <i>handle</i> | A pointer to the <i>i2c_master_edma_handle_t</i> structure.           |
| <i>xfer</i>   | A pointer to the transfer structure of <i>i2c_master_transfer_t</i> . |

Return values

|                                     |                                                |
|-------------------------------------|------------------------------------------------|
| <i>kStatus_Success</i>              | Sucessfully completed the data transmission.   |
| <i>kStatus_I2C_Busy</i>             | A previous transmission is still not finished. |
| <i>kStatus_I2C_Timeout</i>          | Transfer error, waits for a signal timeout.    |
| <i>kStatus_I2C_Arbitration_Lost</i> | Transfer error, arbitration lost.              |
| <i>kStatus_I2C_Nak</i>              | Transfer error, receive NAK during transfer.   |

**19.3.4.3 status\_t I2C\_MasterTransferGetCountEDMA ( *I2C\_Type* \* *base*, *i2c\_master\_edma\_handle\_t* \* *handle*, *size\_t* \* *count* )**

Parameters

|               |                                                                |
|---------------|----------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address.                                   |
| <i>handle</i> | A pointer to the <i>i2c_master_edma_handle_t</i> structure.    |
| <i>count</i>  | A number of bytes transferred by the non-blocking transaction. |

**19.3.4.4 void I2C\_MasterTransferAbortEDMA ( *I2C\_Type* \* *base*, *i2c\_master\_edma\_handle\_t* \* *handle* )**

## Parameters

|               |                                                      |
|---------------|------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address.                         |
| <i>handle</i> | A pointer to the i2c_master_edma_handle_t structure. |

## I2C DMA Driver

### 19.4 I2C DMA Driver

#### 19.4.1 Overview

#### Data Structures

- struct [i2c\\_master\\_dma\\_handle\\_t](#)  
*I2C master DMA transfer structure. [More...](#)*

#### TypeDefs

- typedef void(\* [i2c\\_master\\_dma\\_transfer\\_callback\\_t](#))(I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, status\_t status, void \*userData)  
*I2C master DMA transfer callback typedef.*

#### I2C Block DMA Transfer Operation

- void [I2C\\_MasterTransferCreateHandleDMA](#) (I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, [i2c\\_master\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData, dma\_handle\_t \*dmaHandle)  
*Initializes the I2C handle which is used in transactional functions.*
- status\_t [I2C\\_MasterTransferDMA](#) (I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, [i2c\\_master\\_transfer\\_t](#) \*xfer)  
*Performs a master DMA non-blocking transfer on the I2C bus.*
- status\_t [I2C\\_MasterTransferGetCountDMA](#) (I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets a master transfer status during a DMA non-blocking transfer.*
- void [I2C\\_MasterTransferAbortDMA](#) (I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle)  
*Aborts a master DMA non-blocking transfer early.*

#### 19.4.2 Data Structure Documentation

##### 19.4.2.1 struct \_i2c\_master\_dma\_handle

I2C master DMA handle typedef.

#### Data Fields

- [i2c\\_master\\_transfer\\_t](#) [transfer](#)  
*I2C master transfer struct.*
- size\_t [transferSize](#)  
*Total bytes to be transferred.*
- uint8\_t [state](#)  
*I2C master transfer status.*
- dma\_handle\_t \* [dmaHandle](#)

*The DMA handler used.*

- **i2c\_master\_dma\_transfer\_callback\_t completionCallback**  
*A callback function called after the DMA transfer finished.*
- **void \*userData**  
*A callback parameter passed to the callback function.*

#### 19.4.2.1.0.58 Field Documentation

19.4.2.1.0.58.1 **i2c\_master\_transfer\_t i2c\_master\_dma\_handle\_t::transfer**

19.4.2.1.0.58.2 **size\_t i2c\_master\_dma\_handle\_t::transferSize**

19.4.2.1.0.58.3 **uint8\_t i2c\_master\_dma\_handle\_t::state**

19.4.2.1.0.58.4 **dma\_handle\_t\* i2c\_master\_dma\_handle\_t::dmaHandle**

19.4.2.1.0.58.5 **i2c\_master\_dma\_transfer\_callback\_t i2c\_master\_dma\_handle\_t::completionCallback**

19.4.2.1.0.58.6 **void\* i2c\_master\_dma\_handle\_t::userData**

#### 19.4.3 Typedef Documentation

19.4.3.1 **typedef void(\* i2c\_master\_dma\_transfer\_callback\_t)(I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, status\_t status, void \*userData)**

#### 19.4.4 Function Documentation

19.4.4.1 **void I2C\_MasterTransferCreateHandleDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle*, i2c\_master\_dma\_transfer\_callback\_t *callback*, void \* *userData*, dma\_handle\_t \* *dmaHandle* )**

Parameters

|                  |                                                  |
|------------------|--------------------------------------------------|
| <i>base</i>      | I2C peripheral base address                      |
| <i>handle</i>    | Pointer to the i2c_master_dma_handle_t structure |
| <i>callback</i>  | Pointer to the user callback function            |
| <i>userData</i>  | A user parameter passed to the callback function |
| <i>dmaHandle</i> | DMA handle pointer                               |

19.4.4.2 **status\_t I2C\_MasterTransferDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle*, i2c\_master\_transfer\_t \* *xfer* )**

## I2C DMA Driver

Parameters

|               |                                                                  |
|---------------|------------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address                                      |
| <i>handle</i> | A pointer to the i2c_master_dma_handle_t structure               |
| <i>xfer</i>   | A pointer to the transfer structure of the i2c_master_transfer_t |

Return values

|                                     |                                                 |
|-------------------------------------|-------------------------------------------------|
| <i>kStatus_Success</i>              | Sucessfully completes the data transmission.    |
| <i>kStatus_I2C_Busy</i>             | A previous transmission is still not finished.  |
| <i>kStatus_I2C_Timeout</i>          | A transfer error, waits for the signal timeout. |
| <i>kStatus_I2C_Arbitration-Lost</i> | A transfer error, arbitration lost.             |
| <i>kStatus_I2C_Nak</i>              | A transfer error, receives NAK during transfer. |

### 19.4.4.3 status\_t I2C\_MasterTransferGetCountDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

|               |                                                                       |
|---------------|-----------------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address                                           |
| <i>handle</i> | A pointer to the i2c_master_dma_handle_t structure                    |
| <i>count</i>  | A number of bytes transferred so far by the non-blocking transaction. |

### 19.4.4.4 void I2C\_MasterTransferAbortDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle* )

Parameters

|               |                                                     |
|---------------|-----------------------------------------------------|
| <i>base</i>   | I2C peripheral base address                         |
| <i>handle</i> | A pointer to the i2c_master_dma_handle_t structure. |

## 19.5 I2C FreeRTOS Driver

### 19.5.1 Overview

#### I2C RTOS Operation

- status\_t [I2C\\_RTOS\\_Init](#) (i2c\_rtos\_handle\_t \*handle, I2C\_Type \*base, const i2c\_master\_config\_t \*masterConfig, uint32\_t srcClock\_Hz)  
*Initializes I2C.*
- status\_t [I2C\\_RTOS\\_Deinit](#) (i2c\_rtos\_handle\_t \*handle)  
*Deinitializes the I2C.*
- status\_t [I2C\\_RTOS\\_Transfer](#) (i2c\_rtos\_handle\_t \*handle, i2c\_master\_transfer\_t \*transfer)  
*Performs the I2C transfer.*

### 19.5.2 Function Documentation

#### 19.5.2.1 status\_t I2C\_RTOS\_Init ( *i2c\_rtos\_handle\_t \* handle, I2C\_Type \* base, const i2c\_master\_config\_t \* masterConfig, uint32\_t srcClock\_Hz* )

This function initializes the I2C module and the related RTOS context.

Parameters

|                     |                                                                          |
|---------------------|--------------------------------------------------------------------------|
| <i>handle</i>       | The RTOS I2C handle, the pointer to an allocated space for RTOS context. |
| <i>base</i>         | The pointer base address of the I2C instance to initialize.              |
| <i>masterConfig</i> | The configuration structure to set-up I2C in master mode.                |
| <i>srcClock_Hz</i>  | The frequency of an input clock of the I2C module.                       |

Returns

status of the operation.

#### 19.5.2.2 status\_t I2C\_RTOS\_Deinit ( *i2c\_rtos\_handle\_t \* handle* )

This function deinitializes the I2C module and the related RTOS context.

Parameters

## I2C FreeRTOS Driver

|               |                      |
|---------------|----------------------|
| <i>handle</i> | The RTOS I2C handle. |
|---------------|----------------------|

### 19.5.2.3 **status\_t I2C\_RTOS\_Transfer( i2c\_rtos\_handle\_t \* *handle*, i2c\_master\_transfer\_t \* *transfer* )**

This function performs the I2C transfer according to the data given in the transfer structure.

#### Parameters

|                 |                                                 |
|-----------------|-------------------------------------------------|
| <i>handle</i>   | The RTOS I2C handle.                            |
| <i>transfer</i> | A structure specifying the transfer parameters. |

#### Returns

status of the operation.

# Chapter 20

## LLWU: Low-Leakage Wakeup Unit Driver

### 20.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Low-Leakage Wakeup Unit (LLWU) module of MCUXpresso SDK devices. The LLWU module allows the user to select external pin sources and internal modules as a wake-up source from low-leakage power modes.

### 20.2 External wakeup pins configurations

Configures the external wakeup pins' working modes, gets, and clears the wake pin flags. External wakeup pins are accessed by the `pinIndex`, which is started from 1. Numbers of the external pins depend on the SoC configuration.

### 20.3 Internal wakeup modules configurations

Enables/disables the internal wakeup modules and gets the module flags. Internal modules are accessed by `moduleIndex`, which is started from 1. Numbers of external pins depend the on SoC configuration.

### 20.4 Digital pin filter for external wakeup pin configurations

Configures the digital pin filter of the external wakeup pins' working modes, gets, and clears the pin filter flags. Digital pin filters are accessed by the `filterIndex`, which is started from 1. Numbers of external pins depend on the SoC configuration.

## Data Structures

- struct `llwu_external_pin_filter_mode_t`  
*An external input pin filter control structure. [More...](#)*

## Enumerations

- enum `llwu_external_pin_mode_t` {  
  `kLLWU_ExternalPinDisable` = 0U,  
  `kLLWU_ExternalPinRisingEdge` = 1U,  
  `kLLWU_ExternalPinFallingEdge` = 2U,  
  `kLLWU_ExternalPinAnyEdge` = 3U }  
*External input pin control modes.*
- enum `llwu_pin_filter_mode_t` {  
  `kLLWU_PinFilterDisable` = 0U,  
  `kLLWU_PinFilterRisingEdge` = 1U,  
  `kLLWU_PinFilterFallingEdge` = 2U,  
  `kLLWU_PinFilterAnyEdge` = 3U }  
*Digital filter control modes.*

## Macro Definition Documentation

### Driver version

- #define **FSL\_LLWU\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 1))  
*LLWU driver version 2.0.1.*

## Low-Leakage Wakeup Unit Control APIs

- void **LLWU\_SetExternalWakeUpPinMode** (LLWU\_Type \*base, uint32\_t pinIndex, **llwu\_external\_pin\_mode\_t** pinMode)  
*Sets the external input pin source mode.*
- bool **LLWU\_GetExternalWakeUpPinFlag** (LLWU\_Type \*base, uint32\_t pinIndex)  
*Gets the external wakeup source flag.*
- void **LLWU\_ClearExternalWakeUpPinFlag** (LLWU\_Type \*base, uint32\_t pinIndex)  
*Clears the external wakeup source flag.*
- static void **LLWU\_EnableInternalModuleInterruptWakeup** (LLWU\_Type \*base, uint32\_t moduleIndex, bool enable)  
*Enables/disables the internal module source.*
- static bool **LLWU\_GetInternalWakeUpModuleFlag** (LLWU\_Type \*base, uint32\_t moduleIndex)  
*Gets the external wakeup source flag.*
- void **LLWU\_SetPinFilterMode** (LLWU\_Type \*base, uint32\_t filterIndex, **llwu\_external\_pin\_filter\_mode\_t** filterMode)  
*Sets the pin filter configuration.*
- bool **LLWU\_GetPinFilterFlag** (LLWU\_Type \*base, uint32\_t filterIndex)  
*Gets the pin filter configuration.*
- void **LLWU\_ClearPinFilterFlag** (LLWU\_Type \*base, uint32\_t filterIndex)  
*Clears the pin filter configuration.*
- void **LLWU\_SetResetPinMode** (LLWU\_Type \*base, bool pinEnable, bool enableInLowLeakageMode)  
*Sets the reset pin mode.*

## 20.5 Data Structure Documentation

### 20.5.1 struct **llwu\_external\_pin\_filter\_mode\_t**

#### Data Fields

- uint32\_t **pinIndex**  
*A pin number.*
- **llwu\_pin\_filter\_mode\_t** **filterMode**  
*Filter mode.*

## 20.6 Macro Definition Documentation

### 20.6.1 #define **FSL\_LLWU\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 1))

## 20.7 Enumeration Type Documentation

### 20.7.1 enum llwu\_external\_pin\_mode\_t

Enumerator

*kLLWU\_ExternalPinDisable* Pin disabled as a wakeup input.

*kLLWU\_ExternalPinRisingEdge* Pin enabled with the rising edge detection.

*kLLWU\_ExternalPinFallingEdge* Pin enabled with the falling edge detection.

*kLLWU\_ExternalPinAnyEdge* Pin enabled with any change detection.

### 20.7.2 enum llwu\_pin\_filter\_mode\_t

Enumerator

*kLLWU\_PinFilterDisable* Filter disabled.

*kLLWU\_PinFilterRisingEdge* Filter positive edge detection.

*kLLWU\_PinFilterFallingEdge* Filter negative edge detection.

*kLLWU\_PinFilterAnyEdge* Filter any edge detection.

## 20.8 Function Documentation

### 20.8.1 void LLWU\_SetExternalWakeupsPinMode ( LLWU\_Type \* *base*, uint32\_t *pinIndex*, llwu\_external\_pin\_mode\_t *pinMode* )

This function sets the external input pin source mode that is used as a wake up source.

Parameters

|                 |                                                                         |
|-----------------|-------------------------------------------------------------------------|
| <i>base</i>     | LLWU peripheral base address.                                           |
| <i>pinIndex</i> | A pin index to be enabled as an external wakeup source starting from 1. |
| <i>pinMode</i>  | A pin configuration mode defined in the llwu_external_pin_modes_t.      |

### 20.8.2 bool LLWU\_GetExternalWakeupsPinFlag ( LLWU\_Type \* *base*, uint32\_t *pinIndex* )

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

## Function Documentation

Parameters

|                 |                                   |
|-----------------|-----------------------------------|
| <i>base</i>     | LLWU peripheral base address.     |
| <i>pinIndex</i> | A pin index, which starts from 1. |

Returns

True if the specific pin is a wakeup source.

### 20.8.3 void LLWU\_ClearExternalWakeupPinFlag ( **LLWU\_Type** \* *base*, **uint32\_t** *pinIndex* )

This function clears the external wakeup source flag for a specific pin.

Parameters

|                 |                                   |
|-----------------|-----------------------------------|
| <i>base</i>     | LLWU peripheral base address.     |
| <i>pinIndex</i> | A pin index, which starts from 1. |

### 20.8.4 static void LLWU\_EnableInternalModuleInterruptWakup ( **LLWU\_Type** \* *base*, **uint32\_t** *moduleIndex*, **bool** *enable* ) [inline], [static]

This function enables/disables the internal module source mode that is used as a wake up source.

Parameters

|                    |                                                                            |
|--------------------|----------------------------------------------------------------------------|
| <i>base</i>        | LLWU peripheral base address.                                              |
| <i>moduleIndex</i> | A module index to be enabled as an internal wakeup source starting from 1. |
| <i>enable</i>      | An enable or a disable setting                                             |

### 20.8.5 static bool LLWU\_GetInternalWakeupModuleFlag ( **LLWU\_Type** \* *base*, **uint32\_t** *moduleIndex* ) [inline], [static]

This function checks the external pin flag to detect whether the system is woken up by the specific pin.

Parameters

|                    |                                      |
|--------------------|--------------------------------------|
| <i>base</i>        | LLWU peripheral base address.        |
| <i>moduleIndex</i> | A module index, which starts from 1. |

Returns

True if the specific pin is a wake up source.

#### **20.8.6 void LLWU\_SetPinFilterMode ( LLWU\_Type \* *base*, uint32\_t *filterIndex*, llwu\_external\_pin\_filter\_mode\_t *filterMode* )**

This function sets the pin filter configuration.

Parameters

|                    |                                                                                |
|--------------------|--------------------------------------------------------------------------------|
| <i>base</i>        | LLWU peripheral base address.                                                  |
| <i>filterIndex</i> | A pin filter index used to enable/disable the digital filter, starting from 1. |
| <i>filterMode</i>  | A filter mode configuration                                                    |

#### **20.8.7 bool LLWU\_GetPinFilterFlag ( LLWU\_Type \* *base*, uint32\_t *filterIndex* )**

This function gets the pin filter flag.

Parameters

|                    |                                          |
|--------------------|------------------------------------------|
| <i>base</i>        | LLWU peripheral base address.            |
| <i>filterIndex</i> | A pin filter index, which starts from 1. |

Returns

True if the flag is a source of the existing low-leakage power mode.

#### **20.8.8 void LLWU\_ClearPinFilterFlag ( LLWU\_Type \* *base*, uint32\_t *filterIndex* )**

This function clears the pin filter flag.

## Function Documentation

Parameters

|                    |                                                        |
|--------------------|--------------------------------------------------------|
| <i>base</i>        | LLWU peripheral base address.                          |
| <i>filterIndex</i> | A pin filter index to clear the flag, starting from 1. |

### 20.8.9 void LLWU\_SetResetPinMode ( LLWU\_Type \* *base*, bool *pinEnable*, bool *enableInLowLeakageMode* )

This function determines how the reset pin is used as a low leakage mode exit source.

Parameters

|                         |                                                                      |
|-------------------------|----------------------------------------------------------------------|
| <i>pinEnable</i>        | Enable reset the pin filter                                          |
| <i>pinFilter-Enable</i> | Specify whether the pin filter is enabled in Low-Leakage power mode. |

# Chapter 21

## LPTMR: Low-Power Timer

### 21.1 Overview

The MCUXpresso SDK provides a driver for the Low-Power Timer (LPTMR) of MCUXpresso SDK devices.

### 21.2 Function groups

The LPTMR driver supports operating the module as a time counter or as a pulse counter.

#### 21.2.1 Initialization and deinitialization

The function [LPTMR\\_Init\(\)](#) initializes the LPTMR with specified configurations. The function [LPTMR\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the LPTMR for a timer or a pulse counter mode mode. It also sets up the LPTMR's free running mode operation and a clock source.

The function [LPTMR\\_DeInit\(\)](#) disables the LPTMR module and gates the module clock.

#### 21.2.2 Timer period Operations

The function [LPTMR\\_SetTimerPeriod\(\)](#) sets the timer period in units of count. Timers counts from 0 to the count value set here.

The function [LPTMR\\_GetCurrentTimerCount\(\)](#) reads the current timer counting value. This function returns the real-time timer counting value ranging from 0 to a timer period.

The timer period operation function takes the count value in ticks. Call the utility macros provided in the `fsl_common.h` file to convert to microseconds or milliseconds.

#### 21.2.3 Start and Stop timer operations

The function [LPTMR\\_StartTimer\(\)](#) starts the timer counting. After calling this function, the timer counts up to the counter value set earlier by using the [LPTMR\\_SetPeriod\(\)](#) function. Each time the timer reaches the count value and increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

The function [LPTMR\\_StopTimer\(\)](#) stops the timer counting and resets the timer's counter register.

## Typical use case

### 21.2.4 Status

Provides functions to get and clear the LPTMR status.

### 21.2.5 Interrupt

Provides functions to enable/disable LPTMR interrupts and get the currently enabled interrupts.

## 21.3 Typical use case

### 21.3.1 LPTMR tick example

Updates the LPTMR period and toggles an LED periodically.

```
int main(void)
{
    uint32_t currentCounter = 0U;
    lptmr_config_t lptmrConfig;

    LED_INIT();

    /* Board pin, clock, debug console initialization */
    BOARD_InitHardware();

    /* Configures the LPTMR */
    LPTMR_GetDefaultConfig(&lptmrConfig);

    /* Initializes the LPTMR */
    LPTMR_Init(LPTMR0, &lptmrConfig);

    /* Sets the timer period */
    LPTMR_SetTimerPeriod(LPTMR0, USEC_TO_COUNT(1000000U, LPTMR_SOURCE_CLOCK));

    /* Enables a timer interrupt */
    LPTMR_EnableInterrupts(LPTMR0,
                           kLPTMR_TimerInterruptEnable);

    /* Enables the NVIC */
    EnableIRQ(LPTMR0_IRQn);

    PRINTF("Low Power Timer Example\r\n");

    /* Starts counting */
    LPTMR_StartTimer(LPTMR0);
    while (1)
    {
        if (currentCounter != lptmrCounter)
        {
            currentCounter = lptmrCounter;
            PRINTF("LPTMR interrupt No.%d \r\n", currentCounter);
        }
    }
}
```

## Data Structures

- struct [lptmr\\_config\\_t](#)  
*LPTMR config structure.* [More...](#)

## Enumerations

- enum `lptmr_pin_select_t` {
   
  `kLPTMR_PinSelectInput_0` = 0x0U,
   
  `kLPTMR_PinSelectInput_1` = 0x1U,
   
  `kLPTMR_PinSelectInput_2` = 0x2U,
   
  `kLPTMR_PinSelectInput_3` = 0x3U }
   
*LPTMR pin selection used in pulse counter mode.*
- enum `lptmr_pin_polarity_t` {
   
  `kLPTMR_PinPolarityActiveHigh` = 0x0U,
   
  `kLPTMR_PinPolarityActiveLow` = 0x1U }
   
*LPTMR pin polarity used in pulse counter mode.*
- enum `lptmr_timer_mode_t` {
   
  `kLPTMR_TimerModeTimeCounter` = 0x0U,
   
  `kLPTMR_TimerModePulseCounter` = 0x1U }
   
*LPTMR timer mode selection.*
- enum `lptmr_prescaler_glitch_value_t` {
   
  `kLPTMR_Prescale_Glitch_0` = 0x0U,
   
  `kLPTMR_Prescale_Glitch_1` = 0x1U,
   
  `kLPTMR_Prescale_Glitch_2` = 0x2U,
   
  `kLPTMR_Prescale_Glitch_3` = 0x3U,
   
  `kLPTMR_Prescale_Glitch_4` = 0x4U,
   
  `kLPTMR_Prescale_Glitch_5` = 0x5U,
   
  `kLPTMR_Prescale_Glitch_6` = 0x6U,
   
  `kLPTMR_Prescale_Glitch_7` = 0x7U,
   
  `kLPTMR_Prescale_Glitch_8` = 0x8U,
   
  `kLPTMR_Prescale_Glitch_9` = 0x9U,
   
  `kLPTMR_Prescale_Glitch_10` = 0xAU,
   
  `kLPTMR_Prescale_Glitch_11` = 0xBU,
   
  `kLPTMR_Prescale_Glitch_12` = 0xCU,
   
  `kLPTMR_Prescale_Glitch_13` = 0xDU,
   
  `kLPTMR_Prescale_Glitch_14` = 0xEU,
   
  `kLPTMR_Prescale_Glitch_15` = 0xFU }
   
*LPTMR prescaler/glitch filter values.*
- enum `lptmr_prescaler_clock_select_t` {
   
  `kLPTMR_PrescalerClock_0` = 0x0U,
   
  `kLPTMR_PrescalerClock_1` = 0x1U,
   
  `kLPTMR_PrescalerClock_2` = 0x2U,
   
  `kLPTMR_PrescalerClock_3` = 0x3U }
   
*LPTMR prescaler/glitch filter clock select.*
- enum `lptmr_interrupt_enable_t` { `kLPTMR_TimerInterruptEnable` = LPTMR\_CSR\_TIE\_MASK }
   
*List of the LPTMR interrupts.*
- enum `lptmr_status_flags_t` { `kLPTMR_TimerCompareFlag` = LPTMR\_CSR\_TCF\_MASK }
   
*List of the LPTMR status flags.*

## Driver version

- #define `FSL_LPTMR_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 1))

Version 2.0.1.

### Initialization and deinitialization

- void [LPTMR\\_Init](#) (LPTMR\_Type \*base, const [lptmr\\_config\\_t](#) \*config)  
*Ungates the LPTMR clock and configures the peripheral for a basic operation.*
- void [LPTMR\\_Deinit](#) (LPTMR\_Type \*base)  
*Gates the LPTMR clock.*
- void [LPTMR\\_GetDefaultConfig](#) ([lptmr\\_config\\_t](#) \*config)  
*Fills in the LPTMR configuration structure with default settings.*

### Interrupt Interface

- static void [LPTMR\\_EnableInterrupts](#) (LPTMR\_Type \*base, uint32\_t mask)  
*Enables the selected LPTMR interrupts.*
- static void [LPTMR\\_DisableInterrupts](#) (LPTMR\_Type \*base, uint32\_t mask)  
*Disables the selected LPTMR interrupts.*
- static uint32\_t [LPTMR\\_GetEnabledInterrupts](#) (LPTMR\_Type \*base)  
*Gets the enabled LPTMR interrupts.*

### Status Interface

- static uint32\_t [LPTMR\\_GetStatusFlags](#) (LPTMR\_Type \*base)  
*Gets the LPTMR status flags.*
- static void [LPTMR\\_ClearStatusFlags](#) (LPTMR\_Type \*base, uint32\_t mask)  
*Clears the LPTMR status flags.*

### Read and write the timer period

- static void [LPTMR\\_SetTimerPeriod](#) (LPTMR\_Type \*base, uint32\_t ticks)  
*Sets the timer period in units of count.*
- static uint32\_t [LPTMR\\_GetCurrentTimerCount](#) (LPTMR\_Type \*base)  
*Reads the current timer counting value.*

### Timer Start and Stop

- static void [LPTMR\\_StartTimer](#) (LPTMR\_Type \*base)  
*Starts the timer.*
- static void [LPTMR\\_StopTimer](#) (LPTMR\_Type \*base)  
*Stops the timer.*

## 21.4 Data Structure Documentation

### 21.4.1 struct [lptmr\\_config\\_t](#)

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the [LPTMR\\_GetDefaultConfig\(\)](#) function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

## Data Fields

- [lptmr\\_timer\\_mode\\_t timerMode](#)  
*Time counter mode or pulse counter mode.*
- [lptmr\\_pin\\_select\\_t pinSelect](#)  
*LPTMR pulse input pin select; used only in pulse counter mode.*
- [lptmr\\_pin\\_polarity\\_t pinPolarity](#)  
*LPTMR pulse input pin polarity; used only in pulse counter mode.*
- [bool enableFreeRunning](#)  
*True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set.*
- [bool bypassPrescaler](#)  
*True: bypass prescaler; false: use clock from prescaler.*
- [lptmr\\_prescaler\\_clock\\_select\\_t prescalerClockSource](#)  
*LPTMR clock source.*
- [lptmr\\_prescaler\\_glitch\\_value\\_t value](#)  
*Prescaler or glitch filter value.*

## 21.5 Enumeration Type Documentation

### 21.5.1 enum lptmr\_pin\_select\_t

Enumerator

- kLPTMR\_PinSelectInput\_0*** Pulse counter input 0 is selected.
- kLPTMR\_PinSelectInput\_1*** Pulse counter input 1 is selected.
- kLPTMR\_PinSelectInput\_2*** Pulse counter input 2 is selected.
- kLPTMR\_PinSelectInput\_3*** Pulse counter input 3 is selected.

### 21.5.2 enum lptmr\_pin\_polarity\_t

Enumerator

- kLPTMR\_PinPolarityActiveHigh*** Pulse Counter input source is active-high.
- kLPTMR\_PinPolarityActiveLow*** Pulse Counter input source is active-low.

### 21.5.3 enum lptmr\_timer\_mode\_t

Enumerator

- kLPTMR\_TimerModeTimeCounter*** Time Counter mode.
- kLPTMR\_TimerModePulseCounter*** Pulse Counter mode.

## Enumeration Type Documentation

### 21.5.4 enum lptmr\_prescaler\_glitch\_value\_t

Enumerator

- kLPTMR\_Prescale\_Glitch\_0*** Prescaler divide 2, glitch filter does not support this setting.
- kLPTMR\_Prescale\_Glitch\_1*** Prescaler divide 4, glitch filter 2.
- kLPTMR\_Prescale\_Glitch\_2*** Prescaler divide 8, glitch filter 4.
- kLPTMR\_Prescale\_Glitch\_3*** Prescaler divide 16, glitch filter 8.
- kLPTMR\_Prescale\_Glitch\_4*** Prescaler divide 32, glitch filter 16.
- kLPTMR\_Prescale\_Glitch\_5*** Prescaler divide 64, glitch filter 32.
- kLPTMR\_Prescale\_Glitch\_6*** Prescaler divide 128, glitch filter 64.
- kLPTMR\_Prescale\_Glitch\_7*** Prescaler divide 256, glitch filter 128.
- kLPTMR\_Prescale\_Glitch\_8*** Prescaler divide 512, glitch filter 256.
- kLPTMR\_Prescale\_Glitch\_9*** Prescaler divide 1024, glitch filter 512.
- kLPTMR\_Prescale\_Glitch\_10*** Prescaler divide 2048 glitch filter 1024.
- kLPTMR\_Prescale\_Glitch\_11*** Prescaler divide 4096, glitch filter 2048.
- kLPTMR\_Prescale\_Glitch\_12*** Prescaler divide 8192, glitch filter 4096.
- kLPTMR\_Prescale\_Glitch\_13*** Prescaler divide 16384, glitch filter 8192.
- kLPTMR\_Prescale\_Glitch\_14*** Prescaler divide 32768, glitch filter 16384.
- kLPTMR\_Prescale\_Glitch\_15*** Prescaler divide 65536, glitch filter 32768.

### 21.5.5 enum lptmr\_prescaler\_clock\_select\_t

Note

Clock connections are SoC-specific

Enumerator

- kLPTMR\_PrescalerClock\_0*** Prescaler/glitch filter clock 0 selected.
- kLPTMR\_PrescalerClock\_1*** Prescaler/glitch filter clock 1 selected.
- kLPTMR\_PrescalerClock\_2*** Prescaler/glitch filter clock 2 selected.
- kLPTMR\_PrescalerClock\_3*** Prescaler/glitch filter clock 3 selected.

### 21.5.6 enum lptmr\_interrupt\_enable\_t

Enumerator

- kLPTMR\_TimerInterruptEnable*** Timer interrupt enable.

## 21.5.7 enum lptmr\_status\_flags\_t

Enumerator

*kLPTMR\_TimerCompareFlag* Timer compare flag.

## 21.6 Function Documentation

### 21.6.1 void LPTMR\_Init ( LPTMR\_Type \* *base*, const lptmr\_config\_t \* *config* )

Note

This API should be called at the beginning of the application using the LPTMR driver.

Parameters

|               |                                                 |
|---------------|-------------------------------------------------|
| <i>base</i>   | LPTMR peripheral base address                   |
| <i>config</i> | A pointer to the LPTMR configuration structure. |

### 21.6.2 void LPTMR\_Deinit ( LPTMR\_Type \* *base* )

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | LPTMR peripheral base address |
|-------------|-------------------------------|

### 21.6.3 void LPTMR\_GetDefaultConfig ( lptmr\_config\_t \* *config* )

The default values are as follows.

```
* config->timerMode = kLPTMR_TimerModeTimeCounter;
* config->pinSelect = kLPTMR_PinSelectInput_0;
* config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
* config->enableFreeRunning = false;
* config->bypassPrescaler = true;
* config->prescalerClockSource = kLPTMR_PrescalerClock_1;
* config->value = kLPTMR_Prescale_Glitch_0;
*
```

Parameters

## Function Documentation

|               |                                                 |
|---------------|-------------------------------------------------|
| <i>config</i> | A pointer to the LPTMR configuration structure. |
|---------------|-------------------------------------------------|

### 21.6.4 static void LPTMR\_EnableInterrupts ( LPTMR\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                           |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">lptmr_interrupt_enable_t</a> . |

### 21.6.5 static void LPTMR\_DisableInterrupts ( LPTMR\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                            |
| <i>mask</i> | The interrupts to disable. This is a logical OR of members of the enumeration <a href="#">lptmr_interrupt_enable_t</a> . |

### 21.6.6 static uint32\_t LPTMR\_GetEnabledInterrupts ( LPTMR\_Type \* *base* ) [inline], [static]

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | LPTMR peripheral base address |
|-------------|-------------------------------|

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [lptmr\\_interrupt\\_enable\\_t](#)

### 21.6.7 static uint32\_t LPTMR\_GetStatusFlags ( LPTMR\_Type \* *base* ) [inline], [static]

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | LPTMR peripheral base address |
|-------------|-------------------------------|

Returns

The status flags. This is the logical OR of members of the enumeration [lptmr\\_status\\_flags\\_t](#)

### 21.6.8 static void LPTMR\_ClearStatusFlags ( LPTMR\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                        |
| <i>mask</i> | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">lptmr_status_flags_t</a> . |

### 21.6.9 static void LPTMR\_SetTimerPeriod ( LPTMR\_Type \* *base*, uint32\_t *ticks* ) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note

1. The TCF flag is set with the CNR equals the count provided here and then increments.
2. Call the utility macros provided in the `fsl_common.h` to convert to ticks.

Parameters

|              |                                                                            |
|--------------|----------------------------------------------------------------------------|
| <i>base</i>  | LPTMR peripheral base address                                              |
| <i>ticks</i> | A timer period in units of ticks, which should be equal or greater than 1. |

### 21.6.10 static uint32\_t LPTMR\_GetCurrentTimerCount ( LPTMR\_Type \* *base* ) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

## Function Documentation

Note

Call the utility macros provided in the fsl\_common.h to convert ticks to usec or msec.

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | LPTMR peripheral base address |
|-------------|-------------------------------|

Returns

The current counter value in ticks

### **21.6.11 static void LPTMR\_StartTimer ( LPTMR\_Type \* *base* ) [inline], [static]**

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | LPTMR peripheral base address |
|-------------|-------------------------------|

### **21.6.12 static void LPTMR\_StopTimer ( LPTMR\_Type \* *base* ) [inline], [static]**

This function stops the timer and resets the timer's counter register.

Parameters

|             |                               |
|-------------|-------------------------------|
| <i>base</i> | LPTMR peripheral base address |
|-------------|-------------------------------|

# Chapter 22

## PDB: Programmable Delay Block

### 22.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Programmable Delay Block (PDB) module of MCUXpresso SDK devices.

The PDB driver includes a basic PDB counter, trigger generators for ADC, DAC, and pulse-out.

The basic PDB counter can be used as a general programmable timer with an interrupt. The counter increases automatically with the divided clock signal after it is triggered to start by an external trigger input or the software trigger. There are "milestones" for the output trigger event. When the counter is equal to any of these "milestones", the corresponding trigger is generated and sent out to other modules. These "milestones" are for the following events.

- Counter delay interrupt, which is the interrupt for the PDB module
- ADC pre-trigger to trigger the ADC conversion
- DAC interval trigger to trigger the DAC buffer and move the buffer read pointer
- Pulse-out triggers to generate a single or rising and falling edges, which can be assembled to a window.

The "milestone" values have a flexible load mode. To call the APIs to set these value is equivalent to writing data to their buffer. The loading event occurs as the load mode describes. This design ensures that all "milestones" can be updated at the same time.

### 22.2 Typical use case

#### 22.2.1 Working as basic PDB counter with a PDB interrupt.

```
int main(void)
{
    // ...
    EnableIRQ(DEMO_PDB_IRQ_ID);

    // ...
    // Configures the PDB counter.
    PDB_SetDefaultConfig(& pdbConfigStruct);
    PDB_Init(DEMO_PDB_INSTANCE, & pdbConfigStruct);

    // Configures the delay interrupt.
    PDB_SetModulusValue(DEMO_PDB_INSTANCE, 1000U);
    PDB_SetCounterDelayValue(DEMO_PDB_INSTANCE, 1000U); // The available delay
    // value is less than or equal to the modulus value.
    PDB_EnableInterrupts(DEMO_PDB_INSTANCE,
        kPDB_DelayInterruptEnable);
    PDB_DoLoadValues(DEMO_PDB_INSTANCE);

    while (1)
    {
        // ...
        g_PdbDelayInterruptFlag = false;
```

## Typical use case

```
PDB_DoSoftwareTrigger(DEMO_PDB_INSTANCE);
while (!g_PdbDelayInterruptFlag)
{
}
}

void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
// ...
g_PdbDelayInterruptFlag = true;
PDB_ClearStatusFlags(DEMO_PDB_INSTANCE,
    kPDB_DelayEventFlag);
}
```

### 22.2.2 Working with an additional trigger. The ADC trigger is used as an example.

```
void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
    PDB_ClearStatusFlags(DEMO_PDB_INSTANCE,
        kPDB_DelayEventFlag);
    g_PdbDelayInterruptCounter++;
    g_PdbDelayInterruptFlag = true;
}

void DEMO_PDB_InitADC(void)
{
    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

    ADC16_GetDefaultConfig(&adc16ConfigStruct);
    ADC16_Init(DEMO_PDB_ADC_INSTANCE, &adc16ConfigStruct);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
    ADC16_EnableHardwareTrigger(DEMO_PDB_ADC_INSTANCE, false);
    ADC16_DoAutoCalibration(DEMO_PDB_ADC_INSTANCE);
#endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION */
    ADC16_EnableHardwareTrigger(DEMO_PDB_ADC_INSTANCE, true);

    adc16ChannelConfigStruct.channelNumber = DEMO_PDB_ADC_USER_CHANNEL;
    adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
        true; /* Enable the interrupt. */
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE */
    ADC16_SetChannelConfig(DEMO_PDB_ADC_INSTANCE, DEMO_PDB_ADC_CHANNEL_GROUP, &
        adc16ChannelConfigStruct);
}

void DEMO_PDB_ADC_IRQ_HANDLER_FUNCTION(void)
{
    uint32_t tmp32;

    tmp32 = ADC16_GetChannelConversionValue(DEMO_PDB_ADC_INSTANCE,
        DEMO_PDB_ADC_CHANNEL_GROUP); /* Read to clear COCO flag. */
    g_AdcInterruptCounter++;
    g_AdcInterruptFlag = true;
}

int main(void)
{
// ...

EnableIRQ(DEMO_PDB IRQ_ID);
EnableIRQ(DEMO_PDB_ADC IRQ_ID);
```

```

// ...

// Configures the PDB counter.
PDB_SetDefaultConfig(& pdbConfigStruct);
PDB_Init( DEMO_PDB_INSTANCE, & pdbConfigStruct);

// Configures the delay interrupt.
PDB_SetModulusValue( DEMO_PDB_INSTANCE, 1000U );
PDB_SetCounterDelayValue( DEMO_PDB_INSTANCE, 1000U ); // The available delay
// value is less than or equal to the modulus value.
PDB_EnableInterrupts( DEMO_PDB_INSTANCE,
    kPDB_DelayInterruptEnable );

// Configures the ADC pre-trigger.
pdbAdcPreTriggerConfigStruct.enablePreTriggerMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
pdbAdcPreTriggerConfigStruct.enableOutputMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
pdbAdcPreTriggerConfigStruct.enableBackToBackOperationMask = 0U;
PDB_SetADCPreTriggerConfig( DEMO_PDB_INSTANCE, DEMO_PDB_ADC_TRIGGER_CHANNEL, &
    pdbAdcPreTriggerConfigStruct );
PDB_SetADCPreTriggerDelayValue( DEMO_PDB_INSTANCE,
    DEMO_PDB_ADC_TRIGGER_CHANNEL, DEMO_PDB_ADC_PRETRIGGER_CHANNEL, 200U );
// The available pre-trigger delay value is less than or equal to the modulus
// value.

PDB_DoLoadValues( DEMO_PDB_INSTANCE );

// Configures the ADC.
DEMO_PDB_InitADC();

while (1)
{
    g_PdbDelayInterruptFlag = false;
    g_AdcInterruptFlag = false;
    PDB_DoSoftwareTrigger( DEMO_PDB_INSTANCE );
    while (( !g_PdbDelayInterruptFlag ) || ( !g_AdcInterruptFlag ))
    {
    }
    // ...
}
}

```

## Data Structures

- struct `pdb_config_t`  
*PDB module configuration.* [More...](#)
- struct `pdb_adc_pretrigger_config_t`  
*PDB ADC Pre-trigger configuration.* [More...](#)
- struct `pdb_dac_trigger_config_t`  
*PDB DAC trigger configuration.* [More...](#)

## Enumerations

- enum `_pdb_status_flags` {
 kPDB\_LoadOKFlag = PDB\_SC\_LDOK\_MASK,
 kPDB\_DelayEventFlag = PDB\_SC\_PDBIF\_MASK
 }  
*PDB flags.*
- enum `_pdb_adc_pretrigger_flags` {
 kPDB\_ADCPreTriggerChannel0Flag = PDB\_S\_CF(1U << 0),
 kPDB\_ADCPreTriggerChannel1Flag = PDB\_S\_CF(1U << 1),
 kPDB\_ADCPreTriggerChannel0ErrorFlag = PDB\_S\_ERR(1U << 0),
 }

## Typical use case

- ```
kPDB_ADCPreTriggerChannel1ErrorFlag = PDB_S_ERR(1U << 1) }  
    PDB ADC PreTrigger channel flags.  
• enum _pdb_interrupt_enable {  
    kPDB_SequenceErrorInterruptEnable = PDB_SC_PDDEIE_MASK,  
    kPDB_DelayInterruptEnable = PDB_SC_PDBIE_MASK }  
        PDB buffer interrupts.  
• enum pdb_load_value_mode_t {  
    kPDB_LoadValueImmediately = 0U,  
    kPDB_LoadValueOnCounterOverflow = 1U,  
    kPDB_LoadValueOnTriggerInput = 2U,  
    kPDB_LoadValueOnCounterOverflowOrTriggerInput = 3U }  
        PDB load value mode.  
• enum pdb_prescaler_divider_t {  
    kPDB_PrescalerDivider1 = 0U,  
    kPDB_PrescalerDivider2 = 1U,  
    kPDB_PrescalerDivider4 = 2U,  
    kPDB_PrescalerDivider8 = 3U,  
    kPDB_PrescalerDivider16 = 4U,  
    kPDB_PrescalerDivider32 = 5U,  
    kPDB_PrescalerDivider64 = 6U,  
    kPDB_PrescalerDivider128 = 7U }  
        Prescaler divider.  
• enum pdb_divider_multiplication_factor_t {  
    kPDB_DividerMultiplicationFactor1 = 0U,  
    kPDB_DividerMultiplicationFactor10 = 1U,  
    kPDB_DividerMultiplicationFactor20 = 2U,  
    kPDB_DividerMultiplicationFactor40 = 3U }  
        Multiplication factor select for prescaler.  
• enum pdb_trigger_input_source_t {  
    kPDB_TriggerInput0 = 0U,  
    kPDB_TriggerInput1 = 1U,  
    kPDB_TriggerInput2 = 2U,  
    kPDB_TriggerInput3 = 3U,  
    kPDB_TriggerInput4 = 4U,  
    kPDB_TriggerInput5 = 5U,  
    kPDB_TriggerInput6 = 6U,  
    kPDB_TriggerInput7 = 7U,  
    kPDB_TriggerInput8 = 8U,  
    kPDB_TriggerInput9 = 9U,  
    kPDB_TriggerInput10 = 10U,  
    kPDB_TriggerInput11 = 11U,  
    kPDB_TriggerInput12 = 12U,  
    kPDB_TriggerInput13 = 13U,  
    kPDB_TriggerInput14 = 14U,  
    kPDB_TriggerSoftware = 15U }  
        Trigger input source.
```

## Driver version

- #define **FSL\_PDB\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 1))  
*PDB driver version 2.0.1.*

## Initialization

- void **PDB\_Init** (PDB\_Type \*base, const **pdb\_config\_t** \*config)  
*Initializes the PDB module.*
- void **PDB\_Deinit** (PDB\_Type \*base)  
*De-initializes the PDB module.*
- void **PDB\_GetDefaultConfig** (**pdb\_config\_t** \*config)  
*Initializes the PDB user configuration structure.*
- static void **PDB\_Enable** (PDB\_Type \*base, bool enable)  
*Enables the PDB module.*

## Basic Counter

- static void **PDB\_DoSoftwareTrigger** (PDB\_Type \*base)  
*Triggers the PDB counter by software.*
- static void **PDB\_DoLoadValues** (PDB\_Type \*base)  
*Loads the counter values.*
- static void **PDB\_EnableDMA** (PDB\_Type \*base, bool enable)  
*Enables the DMA for the PDB module.*
- static void **PDB\_EnableInterrupts** (PDB\_Type \*base, uint32\_t mask)  
*Enables the interrupts for the PDB module.*
- static void **PDB\_DisableInterrupts** (PDB\_Type \*base, uint32\_t mask)  
*Disables the interrupts for the PDB module.*
- static uint32\_t **PDB\_GetStatusFlags** (PDB\_Type \*base)  
*Gets the status flags of the PDB module.*
- static void **PDB\_ClearStatusFlags** (PDB\_Type \*base, uint32\_t mask)  
*Clears the status flags of the PDB module.*
- static void **PDB\_SetModulusValue** (PDB\_Type \*base, uint32\_t value)  
*Specifies the counter period.*
- static uint32\_t **PDB\_GetCounterValue** (PDB\_Type \*base)  
*Gets the PDB counter's current value.*
- static void **PDB\_SetCounterDelayValue** (PDB\_Type \*base, uint32\_t value)  
*Sets the value for the PDB counter delay event.*

## ADC Pre-trigger

- static void **PDB\_SetADCPreTriggerConfig** (PDB\_Type \*base, uint32\_t channel, **pdb\_adc\_pretrigger\_config\_t** \*config)  
*Configures the ADC pre-trigger in the PDB module.*
- static void **PDB\_SetADCPreTriggerDelayValue** (PDB\_Type \*base, uint32\_t channel, uint32\_t pre-Channel, uint32\_t value)  
*Sets the value for the ADC pre-trigger delay event.*
- static uint32\_t **PDB\_GetADCPreTriggerStatusFlags** (PDB\_Type \*base, uint32\_t channel)  
*Gets the ADC pre-trigger's status flags.*
- static void **PDB\_ClearADCPreTriggerStatusFlags** (PDB\_Type \*base, uint32\_t channel, uint32\_t mask)

## Data Structure Documentation

*Clears the ADC pre-trigger status flags.*

### DAC Interval Trigger

- void [PDB\\_SetDACTriggerConfig](#) (PDB\_Type \*base, uint32\_t channel, [pdb\\_dac\\_trigger\\_config\\_t](#) \*config)  
*Configures the DAC trigger in the PDB module.*
- static void [PDB\\_SetDACTriggerIntervalValue](#) (PDB\_Type \*base, uint32\_t channel, uint32\_t value)  
*Sets the value for the DAC interval event.*

### Pulse-Out Trigger

- static void [PDB\\_EnablePulseOutTrigger](#) (PDB\_Type \*base, uint32\_t channelMask, bool enable)  
*Enables the pulse out trigger channels.*
- static void [PDB\\_SetPulseOutTriggerDelayValue](#) (PDB\_Type \*base, uint32\_t channel, uint32\_t value1, uint32\_t value2)  
*Sets event values for the pulse out trigger.*

## 22.3 Data Structure Documentation

### 22.3.1 struct [pdb\\_config\\_t](#)

#### Data Fields

- [pdb\\_load\\_value\\_mode\\_t](#) loadValueMode  
*Select the load value mode.*
- [pdb\\_prescaler\\_divider\\_t](#) prescalerDivide  
*Select the prescaler divider.*
- [pdb\\_divider\\_multiplication\\_factor\\_t](#) dividerMultiplicationFactor  
*Multiplication factor select for prescaler.*
- [pdb\\_trigger\\_input\\_source\\_t](#) triggerInputSource  
*Select the trigger input source.*
- bool [enableContinuousMode](#)  
*Enable the PDB operation in Continuous mode.*

**22.3.1.0.0.59 Field Documentation****22.3.1.0.0.59.1 `adb_load_value_mode_t pdb_config_t::loadValueMode`****22.3.1.0.0.59.2 `adb_prescaler_divider_t pdb_config_t::prescalerDivider`****22.3.1.0.0.59.3 `adb_divider_multiplication_factor_t pdb_config_t::dividerMultiplicationFactor`****22.3.1.0.0.59.4 `adb_trigger_input_source_t pdb_config_t::triggerInputSource`****22.3.1.0.0.59.5 `bool pdb_config_t::enableContinuousMode`****22.3.2 `struct adb_adc_pretrigger_config_t`****Data Fields**

- `uint32_t enablePreTriggerMask`  
*PDB Channel Pre-trigger Enable.*
- `uint32_t enableOutputMask`  
*PDB Channel Pre-trigger Output Select.*
- `uint32_t enableBackToBackOperationMask`  
*PDB Channel pre-trigger Back-to-Back Operation Enable.*

**22.3.2.0.0.60 Field Documentation****22.3.2.0.0.60.1 `uint32_t pdb_adc_pretrigger_config_t::enablePreTriggerMask`****22.3.2.0.0.60.2 `uint32_t pdb_adc_pretrigger_config_t::enableOutputMask`**

PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

**22.3.2.0.0.60.3 `uint32_t pdb_adc_pretrigger_config_t::enableBackToBackOperationMask`**

Back-to-back operation enables the ADC conversions complete to trigger the next PDB channel pre-trigger and trigger output, so that the ADC conversions can be triggered on next set of configuration and results registers.

**22.3.3 `struct adb_dac_trigger_config_t`****Data Fields**

- `bool enableExternalTriggerInput`  
*Enables the external trigger for DAC interval counter.*
- `bool enableIntervalTrigger`  
*Enables the DAC interval trigger.*

## Enumeration Type Documentation

### 22.3.3.0.0.61 Field Documentation

22.3.3.0.0.61.1 `bool pdb_dac_trigger_config_t::enableExternalTriggerInput`

22.3.3.0.0.61.2 `bool pdb_dac_trigger_config_t::enableIntervalTrigger`

## 22.4 Macro Definition Documentation

22.4.1 `#define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

## 22.5 Enumeration Type Documentation

### 22.5.1 enum \_pdb\_status\_flags

Enumerator

*kPDB\_LoadOKFlag* This flag is automatically cleared when the values in buffers are loaded into the internal registers after the LDOK bit is set or the PDBEN is cleared.

*kPDB\_DelayEventFlag* PDB timer delay event flag.

### 22.5.2 enum \_pdb\_adc\_pretrigger\_flags

Enumerator

*kPDB\_ADCPreTriggerChannel0Flag* Pre-trigger 0 flag.

*kPDB\_ADCPreTriggerChannel1Flag* Pre-trigger 1 flag.

*kPDB\_ADCPreTriggerChannel0ErrorFlag* Pre-trigger 0 Error.

*kPDB\_ADCPreTriggerChannel1ErrorFlag* Pre-trigger 1 Error.

### 22.5.3 enum \_pdb\_interrupt\_enable

Enumerator

*kPDB\_SequenceErrorInterruptEnable* PDB sequence error interrupt enable.

*kPDB\_DelayInterruptEnable* PDB delay interrupt enable.

### 22.5.4 enum pdb\_load\_value\_mode\_t

Selects the mode to load the internal values after doing the load operation (write 1 to PDBx\_SC[LDOK]). These values are for the following operations.

- PDB counter (PDBx\_MOD, PDBx\_IDLY)
- ADC trigger (PDBx\_CHnDLYm)

- DAC trigger (PDBx\_DACINTx)
- CMP trigger (PDBx\_POyDLY)

Enumerator

***kPDB\_LoadValueImmediately*** Load immediately after 1 is written to LDOK.

***kPDB\_LoadValueOnCounterOverflow*** Load when the PDB counter overflows (reaches the MOD register value).

***kPDB\_LoadValueOnTriggerInput*** Load a trigger input event is detected.

***kPDB\_LoadValueOnCounterOverflowOrTriggerInput*** Load either when the PDB counter overflows or a trigger input is detected.

## 22.5.5 enum pdb\_prescaler\_divider\_t

Counting uses the peripheral clock divided by multiplication factor selected by times of MULT.

Enumerator

***kPDB\_PrescalerDivider1*** Divider x1.

***kPDB\_PrescalerDivider2*** Divider x2.

***kPDB\_PrescalerDivider4*** Divider x4.

***kPDB\_PrescalerDivider8*** Divider x8.

***kPDB\_PrescalerDivider16*** Divider x16.

***kPDB\_PrescalerDivider32*** Divider x32.

***kPDB\_PrescalerDivider64*** Divider x64.

***kPDB\_PrescalerDivider128*** Divider x128.

## 22.5.6 enum pdb\_divider\_multiplication\_factor\_t

Selects the multiplication factor of the prescaler divider for the counter clock.

Enumerator

***kPDB\_DividerMultiplicationFactor1*** Multiplication factor is 1.

***kPDB\_DividerMultiplicationFactor10*** Multiplication factor is 10.

***kPDB\_DividerMultiplicationFactor20*** Multiplication factor is 20.

***kPDB\_DividerMultiplicationFactor40*** Multiplication factor is 40.

## 22.5.7 enum pdb\_trigger\_input\_source\_t

Selects the trigger input source for the PDB. The trigger input source can be internal or external (EXTRG pin), or the software trigger. See chip configuration details for the actual PDB input trigger connections.

## Function Documentation

Enumerator

<i>kPDB_TriggerInput0</i>	Trigger-In 0.
<i>kPDB_TriggerInput1</i>	Trigger-In 1.
<i>kPDB_TriggerInput2</i>	Trigger-In 2.
<i>kPDB_TriggerInput3</i>	Trigger-In 3.
<i>kPDB_TriggerInput4</i>	Trigger-In 4.
<i>kPDB_TriggerInput5</i>	Trigger-In 5.
<i>kPDB_TriggerInput6</i>	Trigger-In 6.
<i>kPDB_TriggerInput7</i>	Trigger-In 7.
<i>kPDB_TriggerInput8</i>	Trigger-In 8.
<i>kPDB_TriggerInput9</i>	Trigger-In 9.
<i>kPDB_TriggerInput10</i>	Trigger-In 10.
<i>kPDB_TriggerInput11</i>	Trigger-In 11.
<i>kPDB_TriggerInput12</i>	Trigger-In 12.
<i>kPDB_TriggerInput13</i>	Trigger-In 13.
<i>kPDB_TriggerInput14</i>	Trigger-In 14.
<i>kPDB_TriggerSoftware</i>	Trigger-In 15, software trigger.

## 22.6 Function Documentation

### 22.6.1 void PDB\_Init ( **PDB\_Type** \* *base*, const **pdb\_config\_t** \* *config* )

This function initializes the PDB module. The operations included are as follows.

- Enable the clock for PDB instance.
- Configure the PDB module.
- Enable the PDB module.

Parameters

<i>base</i>	PDB peripheral base address.
<i>config</i>	Pointer to the configuration structure. See "pdb_config_t".

### 22.6.2 void PDB\_Deinit ( **PDB\_Type** \* *base* )

Parameters

<i>base</i>	PDB peripheral base address.
-------------	------------------------------

### 22.6.3 void PDB\_GetDefaultConfig ( *pdb\_config\_t \* config* )

This function initializes the user configuration structure to a default value. The default values are as follows.

```
* config->loadValueMode = kPDB_LoadValueImmediately;
* config->prescalerDivider = kPDB_PrescalerDivider1;
* config->dividerMultiplicationFactor = kPDB_DividerMultiplicationFactor1
    ;
* config->triggerInputSource = kPDB_TriggerSoftware;
* config->enableContinuousMode = false;
*
```

Parameters

<i>config</i>	Pointer to configuration structure. See "pdb_config_t".
---------------	---

### 22.6.4 static void PDB\_Enable ( *PDB\_Type \* base, bool enable* ) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>enable</i>	Enable the module or not.

### 22.6.5 static void PDB\_DoSoftwareTrigger ( *PDB\_Type \* base* ) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
-------------	------------------------------

### 22.6.6 static void PDB\_DoLoadValues ( *PDB\_Type \* base* ) [inline], [static]

This function loads the counter values from the internal buffer. See "pdb\_load\_value\_mode\_t" about PD-B's load mode.

## Function Documentation

Parameters

<i>base</i>	PDB peripheral base address.
-------------	------------------------------

**22.6.7 static void PDB\_EnableDMA ( PDB\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

<i>base</i>	PDB peripheral base address.
<i>enable</i>	Enable the feature or not.

**22.6.8 static void PDB\_EnableInterrupts ( PDB\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

<i>base</i>	PDB peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_pdb_interrupt_enable".

**22.6.9 static void PDB\_DisableInterrupts ( PDB\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

<i>base</i>	PDB peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_pdb_interrupt_enable".

**22.6.10 static uint32\_t PDB\_GetStatusFlags ( PDB\_Type \* *base* ) [inline], [static]**

Parameters

<i>base</i>	PDB peripheral base address.
-------------	------------------------------

Returns

Mask value for asserted flags. See "\_pdb\_status\_flags".

#### 22.6.11 static void PDB\_ClearStatusFlags ( PDB\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>mask</i>	Mask value of flags. See "_pdb_status_flags".

#### 22.6.12 static void PDB\_SetModulusValue ( PDB\_Type \* *base*, uint32\_t *value* ) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>value</i>	Setting value for the modulus. 16-bit is available.

#### 22.6.13 static uint32\_t PDB\_GetCounterValue ( PDB\_Type \* *base* ) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
-------------	------------------------------

Returns

PDB counter's current value.

#### 22.6.14 static void PDB\_SetCounterDelayValue ( PDB\_Type \* *base*, uint32\_t *value* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	PDB peripheral base address.
<i>value</i>	Setting value for PDB counter delay event. 16-bit is available.

**22.6.15 static void PDB\_SetADCPreTriggerConfig ( PDB\_Type \* *base*, uint32\_t *channel*, pdb\_adc\_pretrigger\_config\_t \* *config* ) [inline], [static]**

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.
<i>config</i>	Pointer to the configuration structure. See "pdb_adc_pretrigger_config_t".

**22.6.16 static void PDB\_SetADCPreTriggerDelayValue ( PDB\_Type \* *base*, uint32\_t *channel*, uint32\_t *preChannel*, uint32\_t *value* ) [inline], [static]**

This function sets the value for ADC pre-trigger delay event. It specifies the delay value for the channel's corresponding pre-trigger. The pre-trigger asserts when the PDB counter is equal to the set value.

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.
<i>preChannel</i>	Channel group index for ADC instance.
<i>value</i>	Setting value for ADC pre-trigger delay event. 16-bit is available.

**22.6.17 static uint32\_t PDB\_GetADCPreTriggerStatusFlags ( PDB\_Type \* *base*, uint32\_t *channel* ) [inline], [static]**

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.

Returns

Mask value for asserted flags. See "\_pdb\_adc\_pretrigger\_flags".

### 22.6.18 static void PDB\_ClearADCPreTriggerStatusFlags ( PDB\_Type \* *base*, uint32\_t *channel*, uint32\_t *mask* ) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.
<i>mask</i>	Mask value for flags. See "_pdb_adc_pretrigger_flags".

### 22.6.19 void PDB\_SetDACTriggerConfig ( PDB\_Type \* *base*, uint32\_t *channel*, pdb\_dac\_trigger\_config\_t \* *config* )

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for DAC instance.
<i>config</i>	Pointer to the configuration structure. See "pdb_dac_trigger_config_t".

### 22.6.20 static void PDB\_SetDACTriggerIntervalValue ( PDB\_Type \* *base*, uint32\_t *channel*, uint32\_t *value* ) [inline], [static]

This function sets the value for DAC interval event. DAC interval trigger triggers the DAC module to update the buffer when the DAC interval counter is equal to the set value.

Parameters

## Function Documentation

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for DAC instance.
<i>value</i>	Setting value for the DAC interval event.

**22.6.21 static void PDB\_EnablePulseOutTrigger ( PDB\_Type \* *base*, uint32\_t *channelMask*, bool *enable* ) [inline], [static]**

Parameters

<i>base</i>	PDB peripheral base address.
<i>channelMask</i>	Channel mask value for multiple pulse out trigger channel.
<i>enable</i>	Whether the feature is enabled or not.

**22.6.22 static void PDB\_SetPulseOutTriggerDelayValue ( PDB\_Type \* *base*, uint32\_t *channel*, uint32\_t *value1*, uint32\_t *value2* ) [inline], [static]**

This function is used to set event values for the pulse output trigger. These pulse output trigger delay values specify the delay for the PDB Pulse-out. Pulse-out goes high when the PDB counter is equal to the pulse output high value (value1). Pulse-out goes low when the PDB counter is equal to the pulse output low value (value2).

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for pulse out trigger channel.
<i>value1</i>	Setting value for pulse out high.
<i>value2</i>	Setting value for pulse out low.

# Chapter 23

## PIT: Periodic Interrupt Timer

### 23.1 Overview

The MCUXpresso SDK provides a driver for the Periodic Interrupt Timer (PIT) of MCUXpresso SDK devices.

### 23.2 Function groups

The PIT driver supports operating the module as a time counter.

#### 23.2.1 Initialization and deinitialization

The function [PIT\\_Init\(\)](#) initializes the PIT with specified configurations. The function [PIT\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the PIT operation in debug mode.

The function [PIT\\_SetTimerChainMode\(\)](#) configures the chain mode operation of each PIT channel.

The function [PIT\\_Deinit\(\)](#) disables the PIT timers and disables the module clock.

#### 23.2.2 Timer period Operations

The function [PITR\\_SetTimerPeriod\(\)](#) sets the timer period in units of count. Timers begin counting down from the value set by this function until it reaches 0.

The function [PIT\\_GetCurrentTimerCount\(\)](#) reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

The timer period operation functions takes the count value in ticks. Users can call the utility macros provided in `fsl_common.h` to convert to microseconds or milliseconds.

#### 23.2.3 Start and Stop timer operations

The function [PIT\\_StartTimer\(\)](#) starts the timer counting. After calling this function, the timer loads the period value set earlier via the [PIT\\_SetPeriod\(\)](#) function and starts counting down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

The function [PIT\\_StopTimer\(\)](#) stops the timer counting.

## Typical use case

### 23.2.4 Status

Provides functions to get and clear the PIT status.

### 23.2.5 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

## 23.3 Typical use case

### 23.3.1 PIT tick example

Updates the PIT period and toggles an LED periodically.

```
int main(void)
{
    /* Structure of initialize PIT */
    pit_config_t pitConfig;

    /* Initialize and enable LED */
    LED_INIT();

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    PIT_GetDefaultConfig(&pitConfig);

    /* Init pit module */
    PIT_Init(PIT, &pitConfig);

    /* Set timer period for channel 0 */
    PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(1000000U,
        PIT_SOURCE_CLOCK));

    /* Enable timer interrupts for channel 0 */
    PIT_EnableInterrupts(PIT, kPIT_Chnl_0,
        kPIT_TimerInterruptEnable);

    /* Enable at the NVIC */
    EnableIRQ(PIT IRQ_ID);

    /* Start channel 0 */
    PRINTF("\r\nStarting channel No.0 ...");
    PIT_StartTimer(PIT, kPIT_Chnl_0);

    while (true)
    {
        /* Check whether occur interrupt and toggle LED */
        if (true == pitIsrFlag)
        {
            PRINTF("\r\n Channel No.0 interrupt is occurred !");
            LED_TOGGLE();
            pitIsrFlag = false;
        }
    }
}
```

## Data Structures

- struct `pit_config_t`  
*PIT configuration structure.* [More...](#)

## Enumerations

- enum `pit_chnl_t` {
   
   kPIT\_Chnl\_0 = 0U,
   
   kPIT\_Chnl\_1,
   
   kPIT\_Chnl\_2,
   
   kPIT\_Chnl\_3
 }
   
*List of PIT channels.*
  - enum `pit_interrupt_enable_t` { kPIT\_TimerInterruptEnable = PIT\_TCTRL\_TIE\_MASK }
  - enum `pit_status_flags_t` { kPIT\_TimerFlag = PIT\_TFLG\_TIF\_MASK }
- List of PIT interrupts.*
- List of PIT status flags.*

## Driver version

- #define `FSL_PIT_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 0))  
*Version 2.0.0.*

## Initialization and deinitialization

- void `PIT_Init` (PIT\_Type \*base, const `pit_config_t` \*config)  
*Ungates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.*
- void `PIT_Deinit` (PIT\_Type \*base)  
*Gates the PIT clock and disables the PIT module.*
- static void `PIT_GetDefaultConfig` (`pit_config_t` \*config)  
*Fills in the PIT configuration structure with the default settings.*
- static void `PIT_SetTimerChainMode` (PIT\_Type \*base, `pit_chnl_t` channel, bool enable)  
*Enables or disables chaining a timer with the previous timer.*

## Interrupt Interface

- static void `PIT_EnableInterrupts` (PIT\_Type \*base, `pit_chnl_t` channel, uint32\_t mask)  
*Enables the selected PIT interrupts.*
- static void `PIT_DisableInterrupts` (PIT\_Type \*base, `pit_chnl_t` channel, uint32\_t mask)  
*Disables the selected PIT interrupts.*
- static uint32\_t `PIT_GetEnabledInterrupts` (PIT\_Type \*base, `pit_chnl_t` channel)  
*Gets the enabled PIT interrupts.*

## Status Interface

- static uint32\_t `PIT_GetStatusFlags` (PIT\_Type \*base, `pit_chnl_t` channel)  
*Gets the PIT status flags.*
- static void `PIT_ClearStatusFlags` (PIT\_Type \*base, `pit_chnl_t` channel, uint32\_t mask)  
*Clears the PIT status flags.*

## Enumeration Type Documentation

### Read and Write the timer period

- static void [PIT\\_SetTimerPeriod](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel, uint32\_t count)  
*Sets the timer period in units of count.*
- static uint32\_t [PIT\\_GetCurrentTimerCount](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel)  
*Reads the current timer counting value.*

### Timer Start and Stop

- static void [PIT\\_StartTimer](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel)  
*Starts the timer counting.*
- static void [PIT\\_StopTimer](#) (PIT\_Type \*base, [pit\\_chnl\\_t](#) channel)  
*Stops the timer counting.*

## 23.4 Data Structure Documentation

### 23.4.1 struct pit\_config\_t

This structure holds the configuration settings for the PIT peripheral. To initialize this structure to reasonable defaults, call the [PIT\\_GetDefaultConfig\(\)](#) function and pass a pointer to your config structure instance.

The configuration structure can be made constant so it resides in flash.

### Data Fields

- bool [enableRunInDebug](#)  
*true: Timers run in debug mode; false: Timers stop in debug mode*

## 23.5 Enumeration Type Documentation

### 23.5.1 enum pit\_chnl\_t

Note

Actual number of available channels is SoC dependent

Enumerator

- kPIT\_Chnl\_0*** PIT channel number 0.
- kPIT\_Chnl\_1*** PIT channel number 1.
- kPIT\_Chnl\_2*** PIT channel number 2.
- kPIT\_Chnl\_3*** PIT channel number 3.

### 23.5.2 enum pit\_interrupt\_enable\_t

Enumerator

*kPIT\_TimerInterruptEnable* Timer interrupt enable.

### 23.5.3 enum pit\_status\_flags\_t

Enumerator

*kPIT\_TimerFlag* Timer flag.

## 23.6 Function Documentation

### 23.6.1 void PIT\_Init ( PIT\_Type \* *base*, const pit\_config\_t \* *config* )

Note

This API should be called at the beginning of the application using the PIT driver.

Parameters

<i>base</i>	PIT peripheral base address
<i>config</i>	Pointer to the user's PIT config structure

### 23.6.2 void PIT\_Deinit ( PIT\_Type \* *base* )

Parameters

<i>base</i>	PIT peripheral base address
-------------	-----------------------------

### 23.6.3 static void PIT\_GetDefaultConfig ( pit\_config\_t \* *config* ) [inline], [static]

The default values are as follows.

```
*     config->enableRunInDebug = false;
*
```

## Function Documentation

Parameters

<i>config</i>	Pointer to the configuration structure.
---------------	---

### 23.6.4 static void PIT\_SetTimerChainMode ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **bool** *enable* ) [inline], [static]

When a timer has a chain mode enabled, it only counts after the previous timer has expired. If the timer n-1 has counted down to 0, counter n decrements the value by one. Each timer is 32-bits, which allows the developers to chain timers together and form a longer timer (64-bits and larger). The first timer (timer 0) can't be chained to any other timer.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number which is chained with the previous timer
<i>enable</i>	Enable or disable chain. true: Current timer is chained with the previous timer. false: Timer doesn't chain with other timers.

### 23.6.5 static void PIT\_EnableInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">pit_interrupt_enable_t</a>

### 23.6.6 static void PIT\_DisableInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number
<i>mask</i>	The interrupts to disable. This is a logical OR of members of the enumeration <a href="#">pit_interrupt_enable_t</a>

### 23.6.7 static uint32\_t PIT\_GetEnabledInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [pit\\_interrupt\\_enable\\_t](#)

### 23.6.8 static uint32\_t PIT\_GetStatusFlags ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number

Returns

The status flags. This is the logical OR of members of the enumeration [pit\\_status\\_flags\\_t](#)

### 23.6.9 static void PIT\_ClearStatusFlags ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration <a href="#">pit_status_flags_t</a>

### 23.6.10 static void PIT\_SetTimerPeriod( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *count* ) [inline], [static]

Timers begin counting from the value set by this function until it reaches 0, then it generates an interrupt and load this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note

Users can call the utility macros provided in fsl\_common.h to convert to ticks.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number
<i>count</i>	Timer period in units of ticks

### 23.6.11 static uint32\_t PIT\_GetCurrentTimerCount( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [inline], [static]

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

Users can call the utility macros provided in fsl\_common.h to convert ticks to usec or msec.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number

Returns

Current timer counting value in ticks

### 23.6.12 static void PIT\_StartTimer ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [**inline**], [**static**]

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number.

### 23.6.13 static void PIT\_StopTimer ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel* ) [**inline**], [**static**]

This function stops every timer counting. Timers reload their periods respectively after the next time they call the PIT\_DRV\_StartTimer.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number.

## Function Documentation

# Chapter 24

## PMC: Power Management Controller

### 24.1 Overview

The MCUXpresso SDK provides a Peripheral driver for the Power Management Controller (PMC) module of MCUXpresso SDK devices. The PMC module contains internal voltage regulator, power on reset, low-voltage detect system, and high-voltage detect system.

### Data Structures

- struct [pmc\\_low\\_volt\\_detect\\_config\\_t](#)  
*Low-voltage Detect Configuration Structure. [More...](#)*
- struct [pmc\\_low\\_volt\\_warning\\_config\\_t](#)  
*Low-voltage Warning Configuration Structure. [More...](#)*
- struct [pmc\\_bandgap\\_buffer\\_config\\_t](#)  
*Bandgap Buffer configuration. [More...](#)*

### Enumerations

- enum [pmc\\_low\\_volt\\_detect\\_volt\\_select\\_t](#) {  
  kPMC\_LowVoltDetectLowTrip = 0U,  
  kPMC\_LowVoltDetectHighTrip = 1U }  
*Low-voltage Detect Voltage Select.*
- enum [pmc\\_low\\_volt\\_warning\\_volt\\_select\\_t](#) {  
  kPMC\_LowVoltWarningLowTrip = 0U,  
  kPMC\_LowVoltWarningMid1Trip = 1U,  
  kPMC\_LowVoltWarningMid2Trip = 2U,  
  kPMC\_LowVoltWarningHighTrip = 3U }  
*Low-voltage Warning Voltage Select.*

### Driver version

- #define [FSL\\_PMC\\_DRIVER\\_VERSION](#) (MAKE\_VERSION(2, 0, 0))  
*PMC driver version.*

### Power Management Controller Control APIs

- void [PMC\\_ConfigureLowVoltDetect](#) (PMC\_Type \*base, const [pmc\\_low\\_volt\\_detect\\_config\\_t](#) \*config)  
*Configures the low-voltage detect setting.*
- static bool [PMC\\_GetLowVoltDetectFlag](#) (PMC\_Type \*base)  
*Gets the Low-voltage Detect Flag status.*
- static void [PMC\\_ClearLowVoltDetectFlag](#) (PMC\_Type \*base)  
*Acknowledges clearing the Low-voltage Detect flag.*

## Data Structure Documentation

- void [PMC\\_ConfigureLowVoltWarning](#) (PMC\_Type \*base, const [pmc\\_low\\_volt\\_warning\\_config\\_t](#) \*config)  
*Configures the low-voltage warning setting.*
- static bool [PMC\\_GetLowVoltWarningFlag](#) (PMC\_Type \*base)  
*Gets the Low-voltage Warning Flag status.*
- static void [PMC\\_ClearLowVoltWarningFlag](#) (PMC\_Type \*base)  
*Acknowledges the Low-voltage Warning flag.*
- void [PMC\\_ConfigureBandgapBuffer](#) (PMC\_Type \*base, const [pmc\\_bandgap\\_buffer\\_config\\_t](#) \*config)  
*Configures the PMC bandgap.*
- static bool [PMC\\_GetPeriphIOIsolationFlag](#) (PMC\_Type \*base)  
*Gets the acknowledge Peripherals and I/O pads isolation flag.*
- static void [PMC\\_ClearPeriphIOIsolationFlag](#) (PMC\_Type \*base)  
*Acknowledges the isolation flag to Peripherals and I/O pads.*
- static bool [PMC\\_IsRegulatorInRunRegulation](#) (PMC\_Type \*base)  
*Gets the regulator regulation status.*

## 24.2 Data Structure Documentation

### 24.2.1 struct pmc\_low\_volt\_detect\_config\_t

#### Data Fields

- bool [enableInt](#)  
*Enable interrupt when Low-voltage detect.*
- bool [enableReset](#)  
*Enable system reset when Low-voltage detect.*
- [pmc\\_low\\_volt\\_detect\\_volt\\_select\\_t](#) [voltSelect](#)  
*Low-voltage detect trip point voltage selection.*

### 24.2.2 struct pmc\_low\_volt\_warning\_config\_t

#### Data Fields

- bool [enableInt](#)  
*Enable interrupt when low-voltage warning.*
- [pmc\\_low\\_volt\\_warning\\_volt\\_select\\_t](#) [voltSelect](#)  
*Low-voltage warning trip point voltage selection.*

### 24.2.3 struct pmc\_bandgap\_buffer\_config\_t

#### Data Fields

- bool [enable](#)  
*Enable bandgap buffer.*
- bool [enableInLowPowerMode](#)

*Enable bandgap buffer in low-power mode.*

#### 24.2.3.0.0.62 Field Documentation

24.2.3.0.0.62.1 `bool pmc_bandgap_buffer_config_t::enable`

24.2.3.0.0.62.2 `bool pmc_bandgap_buffer_config_t::enableInLowPowerMode`

### 24.3 Macro Definition Documentation

24.3.1 `#define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

Version 2.0.0.

### 24.4 Enumeration Type Documentation

24.4.1 `enum pmc_low_volt_detect_volt_select_t`

Enumerator

*kPMC\_LowVoltDetectLowTrip* Low-trip point selected (VLVD = VLVDL )

*kPMC\_LowVoltDetectHighTrip* High-trip point selected (VLVD = VLVDH )

24.4.2 `enum pmc_low_volt_warning_volt_select_t`

Enumerator

*kPMC\_LowVoltWarningLowTrip* Low-trip point selected (VLVW = VLVW1)

*kPMC\_LowVoltWarningMid1Trip* Mid 1 trip point selected (VLVW = VLVW2)

*kPMC\_LowVoltWarningMid2Trip* Mid 2 trip point selected (VLVW = VLVW3)

*kPMC\_LowVoltWarningHighTrip* High-trip point selected (VLVW = VLVW4)

### 24.5 Function Documentation

24.5.1 `void PMC_ConfigureLowVoltDetect ( PMC_Type * base, const pmc_low_volt_detect_config_t * config )`

This function configures the low-voltage detect setting, including the trip point voltage setting, enables or disables the interrupt, enables or disables the system reset.

Parameters

## Function Documentation

<i>base</i>	PMC peripheral base address.
<i>config</i>	Low-voltage detect configuration structure.

### 24.5.2 static bool PMC\_GetLowVoltDetectFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function reads the current LVDF status. If it returns 1, a low-voltage event is detected.

Parameters

<i>base</i>	PMC peripheral base address.
-------------	------------------------------

Returns

Current low-voltage detect flag

- true: Low-voltage detected
- false: Low-voltage not detected

### 24.5.3 static void PMC\_ClearLowVoltDetectFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function acknowledges the low-voltage detection errors (write 1 to clear LVDF).

Parameters

<i>base</i>	PMC peripheral base address.
-------------	------------------------------

### 24.5.4 void PMC\_ConfigureLowVoltWarning ( **PMC\_Type** \* *base*, const *pmc\_low\_volt\_warning\_config\_t* \* *config* )

This function configures the low-voltage warning setting, including the trip point voltage setting and enabling or disabling the interrupt.

Parameters

<i>base</i>	PMC peripheral base address.
<i>config</i>	Low-voltage warning configuration structure.

#### 24.5.5 static bool PMC\_GetLowVoltWarningFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Parameters

<i>base</i>	PMC peripheral base address.
-------------	------------------------------

Returns

Current LVWF status

- true: Low-voltage Warning Flag is set.
- false: the Low-voltage Warning does not happen.

#### 24.5.6 static void PMC\_ClearLowVoltWarningFlag ( **PMC\_Type** \* *base* ) [inline], [static]

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

<i>base</i>	PMC peripheral base address.
-------------	------------------------------

#### 24.5.7 void PMC\_ConfigureBandgapBuffer ( **PMC\_Type** \* *base*, const **pmc\_bandgap\_buffer\_config\_t** \* *config* )

This function configures the PMC bandgap, including the drive select and behavior in low-power mode.

Parameters

## Function Documentation

<i>base</i>	PMC peripheral base address.
<i>config</i>	Pointer to the configuration structure

### 24.5.8 static bool PMC\_GetPeriphIOIsolationFlag ( **PMC\_Type** \* *base* ) [**inline**], [**static**]

This function reads the Acknowledge Isolation setting that indicates whether certain peripherals and the I/O pads are in a latched state as a result of having been in the VLLS mode.

Parameters

<i>base</i>	PMC peripheral base address.
<i>base</i>	Base address for current PMC instance.

Returns

ACK isolation 0 - Peripherals and I/O pads are in a normal run state. 1 - Certain peripherals and I/O pads are in an isolated and latched state.

### 24.5.9 static void PMC\_ClearPeriphIOIsolationFlag ( **PMC\_Type** \* *base* ) [**inline**], [**static**]

This function clears the ACK Isolation flag. Writing one to this setting when it is set releases the I/O pads and certain peripherals to their normal run mode state.

Parameters

<i>base</i>	PMC peripheral base address.
-------------	------------------------------

### 24.5.10 static bool PMC\_IsRegulatorInRunRegulation ( **PMC\_Type** \* *base* ) [**inline**], [**static**]

This function returns the regulator to run a regulation status. It provides the current status of the internal voltage regulator.

## Parameters

<i>base</i>	PMC peripheral base address.
<i>base</i>	Base address for current PMC instance.

## Returns

Regulation status 0 - Regulator is in a stop regulation or in transition to/from the regulation. 1 - Regulator is in a run regulation.

## Function Documentation

# Chapter 25

## PORT: Port Control and Interrupts

### 25.1 Overview

The MCUXpresso SDK provides a driver for the Port Control and Interrupts (PORT) module of MCUXpresso SDK devices.

### 25.2 Typical configuration use case

#### 25.2.1 Input PORT configuration

```
/* Input pin PORT configuration */
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnLockRegister,
};

/* Sets the configuration */
PORT_SetPinConfig(PORTA, 4, &config);
```

#### 25.2.2 I2C PORT Configuration

```
/* I2C pin PORTconfiguration */
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainEnable,
    kPORT_LowDriveStrength,
    kPORT_MuxAlt5,
    kPORT_UnLockRegister,
};

PORT_SetPinConfig(PORTE, 24u, &config);
PORT_SetPinConfig(PORTE, 25u, &config);
```

## Data Structures

- struct [port\\_digital\\_filter\\_config\\_t](#)  
*PORT digital filter feature configuration definition.* [More...](#)
- struct [port\\_pin\\_config\\_t](#)  
*PORT pin configuration structure.* [More...](#)

## Typical configuration use case

### Enumerations

- enum `_port_pull` {  
    `kPORT_PullDisable` = 0U,  
    `kPORT_PullDown` = 2U,  
    `kPORT_PullUp` = 3U }  
    *Internal resistor pull feature selection.*
- enum `_port_slew_rate` {  
    `kPORT_FastSlewRate` = 0U,  
    `kPORT_SlowSlewRate` = 1U }  
    *Slew rate selection.*
- enum `_port_open_drain_enable` {  
    `kPORT_OpenDrainDisable` = 0U,  
    `kPORT_OpenDrainEnable` = 1U }  
    *Open Drain feature enable/disable.*
- enum `_port_passive_filter_enable` {  
    `kPORT_PassiveFilterDisable` = 0U,  
    `kPORT_PassiveFilterEnable` = 1U }  
    *Passive filter feature enable/disable.*
- enum `_port_drive_strength` {  
    `kPORT_LowDriveStrength` = 0U,  
    `kPORT_HighDriveStrength` = 1U }  
    *Configures the drive strength.*
- enum `_port_lock_register` {  
    `kPORT_UnlockRegister` = 0U,  
    `kPORT_LockRegister` = 1U }  
    *Unlock/lock the pin control register field[15:0].*
- enum `port_mux_t` {  
    `kPORT_PinDisabledOrAnalog` = 0U,  
    `kPORT_MuxAsGpio` = 1U,  
    `kPORT_MuxAlt2` = 2U,  
    `kPORT_MuxAlt3` = 3U,  
    `kPORT_MuxAlt4` = 4U,  
    `kPORT_MuxAlt5` = 5U,  
    `kPORT_MuxAlt6` = 6U,  
    `kPORT_MuxAlt7` = 7U,  
    `kPORT_MuxAlt8` = 8U,  
    `kPORT_MuxAlt9` = 9U,  
    `kPORT_MuxAlt10` = 10U,  
    `kPORT_MuxAlt11` = 11U,  
    `kPORT_MuxAlt12` = 12U,  
    `kPORT_MuxAlt13` = 13U,  
    `kPORT_MuxAlt14` = 14U,  
    `kPORT_MuxAlt15` = 15U }  
    *Pin mux selection.*
- enum `port_interrupt_t` {

```
kPORT_InterruptOrDMADisabled = 0x0U,
kPORT_DMARisingEdge = 0x1U,
kPORT_DMAFallingEdge = 0x2U,
kPORT_DMAEitherEdge = 0x3U,
kPORT_InterruptLogicZero = 0x8U,
kPORT_InterruptRisingEdge = 0x9U,
kPORT_InterruptFallingEdge = 0xAU,
kPORT_InterruptEitherEdge = 0xBU,
kPORT_InterruptLogicOne = 0xCU }
```

*Configures the interrupt generation condition.*

- enum **port\_digital\_filter\_clock\_source\_t** {
 kPORT\_BusClock = 0U,
 kPORT\_LpoClock = 1U }

*Digital filter clock source selection.*

## Driver version

- #define **FSL\_PORT\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 2))  
*Version 2.0.2.*

## Configuration

- static void **PORT\_SetPinConfig** (PORT\_Type \*base, uint32\_t pin, const **port\_pin\_config\_t** \*config)  
*Sets the port PCR register.*
- static void **PORT\_SetMultiplePinsConfig** (PORT\_Type \*base, uint32\_t mask, const **port\_pin\_config\_t** \*config)  
*Sets the port PCR register for multiple pins.*
- static void **PORT\_SetPinMux** (PORT\_Type \*base, uint32\_t pin, **port\_mux\_t** mux)  
*Configures the pin muxing.*
- static void **PORT\_EnablePinsDigitalFilter** (PORT\_Type \*base, uint32\_t mask, bool enable)  
*Enables the digital filter in one port, each bit of the 32-bit register represents one pin.*
- static void **PORT\_SetDigitalFilterConfig** (PORT\_Type \*base, const **port\_digital\_filter\_config\_t** \*config)  
*Sets the digital filter in one port, each bit of the 32-bit register represents one pin.*

## Interrupt

- static void **PORT\_SetPinInterruptConfig** (PORT\_Type \*base, uint32\_t pin, **port\_interrupt\_t** config)  
*Configures the port pin interrupt/DMA request.*
- static uint32\_t **PORT\_GetPinsInterruptFlags** (PORT\_Type \*base)  
*Reads the whole port status flag.*
- static void **PORT\_ClearPinsInterruptFlags** (PORT\_Type \*base, uint32\_t mask)  
*Clears the multiple pin interrupt status flag.*

## Enumeration Type Documentation

### 25.3 Data Structure Documentation

#### 25.3.1 struct port\_digital\_filter\_config\_t

##### Data Fields

- `uint32_t digitalFilterWidth`  
*Set digital filter width.*
- `port_digital_filter_clock_source_t clockSource`  
*Set digital filter clockSource.*

#### 25.3.2 struct port\_pin\_config\_t

##### Data Fields

- `uint16_t pullSelect: 2`  
*No-pull/pull-down/pull-up select.*
- `uint16_t slewRate: 1`  
*Fast/slow slew rate Configure.*
- `uint16_t passiveFilterEnable: 1`  
*Passive filter enable/disable.*
- `uint16_t openDrainEnable: 1`  
*Open drain enable/disable.*
- `uint16_t driveStrength: 1`  
*Fast/slow drive strength configure.*
- `uint16_t mux: 3`  
*Pin mux Configure.*
- `uint16_t lockRegister: 1`  
*Lock/unlock the PCR field[15:0].*

### 25.4 Macro Definition Documentation

#### 25.4.1 #define FSL\_PORT\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 2))

### 25.5 Enumeration Type Documentation

#### 25.5.1 enum \_port\_pull

Enumerator

`kPORT_PullDisable` Internal pull-up/down resistor is disabled.

`kPORT_PullDown` Internal pull-down resistor is enabled.

`kPORT_PullUp` Internal pull-up resistor is enabled.

### 25.5.2 enum \_port\_slew\_rate

Enumerator

***kPORT\_FastSlewRate*** Fast slew rate is configured.

***kPORT\_SlowSlewRate*** Slow slew rate is configured.

### 25.5.3 enum \_port\_open\_drain\_enable

Enumerator

***kPORT\_OpenDrainDisable*** Open drain output is disabled.

***kPORT\_OpenDrainEnable*** Open drain output is enabled.

### 25.5.4 enum \_port\_passive\_filter\_enable

Enumerator

***kPORT\_PassiveFilterDisable*** Passive input filter is disabled.

***kPORT\_PassiveFilterEnable*** Passive input filter is enabled.

### 25.5.5 enum \_port\_drive\_strength

Enumerator

***kPORT\_LowDriveStrength*** Low-drive strength is configured.

***kPORT\_HighDriveStrength*** High-drive strength is configured.

### 25.5.6 enum \_port\_lock\_register

Enumerator

***kPORT\_UnlockRegister*** Pin Control Register fields [15:0] are not locked.

***kPORT\_LockRegister*** Pin Control Register fields [15:0] are locked.

### 25.5.7 enum port\_mux\_t

Enumerator

***kPORT\_PinDisabledOrAnalog*** Corresponding pin is disabled, but is used as an analog pin.

## Function Documentation

***kPORT\_MuxAsGpio*** Corresponding pin is configured as GPIO.  
***kPORT\_MuxAlt2*** Chip-specific.  
***kPORT\_MuxAlt3*** Chip-specific.  
***kPORT\_MuxAlt4*** Chip-specific.  
***kPORT\_MuxAlt5*** Chip-specific.  
***kPORT\_MuxAlt6*** Chip-specific.  
***kPORT\_MuxAlt7*** Chip-specific.  
***kPORT\_MuxAlt8*** Chip-specific.  
***kPORT\_MuxAlt9*** Chip-specific.  
***kPORT\_MuxAlt10*** Chip-specific.  
***kPORT\_MuxAlt11*** Chip-specific.  
***kPORT\_MuxAlt12*** Chip-specific.  
***kPORT\_MuxAlt13*** Chip-specific.  
***kPORT\_MuxAlt14*** Chip-specific.  
***kPORT\_MuxAlt15*** Chip-specific.

### 25.5.8 enum port\_interrupt\_t

Enumerator

***kPORT\_InterruptOrDMADisabled*** Interrupt/DMA request is disabled.  
***kPORT\_DMARisingEdge*** DMA request on rising edge.  
***kPORT\_DMAFallingEdge*** DMA request on falling edge.  
***kPORT\_DMAEitherEdge*** DMA request on either edge.  
***kPORT\_InterruptLogicZero*** Interrupt when logic zero.  
***kPORT\_InterruptRisingEdge*** Interrupt on rising edge.  
***kPORT\_InterruptFallingEdge*** Interrupt on falling edge.  
***kPORT\_InterruptEitherEdge*** Interrupt on either edge.  
***kPORT\_InterruptLogicOne*** Interrupt when logic one.

### 25.5.9 enum port\_digital\_filter\_clock\_source\_t

Enumerator

***kPORT\_BusClock*** Digital filters are clocked by the bus clock.  
***kPORT\_LpoClock*** Digital filters are clocked by the 1 kHz LPO clock.

## 25.6 Function Documentation

### 25.6.1 static void PORT\_SetPinConfig ( PORT\_Type \* *base*, uint32\_t *pin*, const port\_pin\_config\_t \* *config* ) [inline], [static]

This is an example to define an input pin or output pin PCR configuration.

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
*     kPORT_PullUp,
*     kPORT_FastSlewRate,
*     kPORT_PassiveFilterDisable,
*     kPORT_OpenDrainDisable,
*     kPORT_LowDriveStrength,
*     kPORT_MuxAsGpio,
*     kPORT_UnLockRegister,
* };
*
```

## Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>config</i>	PORT PCR register configuration structure.

### 25.6.2 static void PORT\_SetMultiplePinsConfig ( PORT\_Type \* *base*, uint32\_t *mask*, const port\_pin\_config\_t \* *config* ) [inline], [static]

This is an example to define input pins or output pins PCR configuration.

```
* // Define a digital input pin PCR configuration
* port_pin_config_t config = {
*     kPORT_PullUp ,
*     kPORT_PullEnable,
*     kPORT_FastSlewRate,
*     kPORT_PassiveFilterDisable,
*     kPORT_OpenDrainDisable,
*     kPORT_LowDriveStrength,
*     kPORT_MuxAsGpio,
*     kPORT_UnlockRegister,
* };
*
```

## Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.
<i>config</i>	PORT PCR register configuration structure.

### 25.6.3 static void PORT\_SetPinMux ( PORT\_Type \* *base*, uint32\_t *pin*, port\_mux\_t *mux* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>mux</i>	pin muxing slot selection. <ul style="list-style-type: none"><li>• <a href="#">kPORT_PinDisabledOrAnalog</a>: Pin disabled or work in analog function.</li><li>• <a href="#">kPORT_MuxAsGpio</a> : Set as GPIO.</li><li>• <a href="#">kPORT_MuxAlt2</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt3</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt4</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt5</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt6</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt7</a> : chip-specific. : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : <a href="#">kPORT_PinDisabledOrAnalog</a>). This function is recommended to use to reset the pin mux</li></ul>

**25.6.4 static void PORT\_EnablePinsDigitalFilter ( PORT\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]**

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.

**25.6.5 static void PORT\_SetDigitalFilterConfig ( PORT\_Type \* *base*, const port\_digital\_filter\_config\_t \* *config* ) [inline], [static]**

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>config</i>	PORT digital filter configuration structure.

**25.6.6 static void PORT\_SetPinInterruptConfig ( PORT\_Type \* *base*, uint32\_t *pin*, port\_interrupt\_t *config* ) [inline], [static]**

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>config</i>	<p>PORT pin interrupt configuration.</p> <ul style="list-style-type: none"> <li>• <a href="#">kPORT_InterruptOrDMADisabled</a>: Interrupt/DMA request disabled.</li> <li>• <a href="#">kPORT_DMARisingEdge</a> : DMA request on rising edge(if the DMA requests exit).</li> <li>• <a href="#">kPORT_DMAFallingEdge</a>: DMA request on falling edge(if the DMA requests exit).</li> <li>• <a href="#">kPORT_DMAEitherEdge</a> : DMA request on either edge(if the DMA requests exit).</li> <li>• #<a href="#">kPORT_FlagRisingEdge</a> : Flag sets on rising edge(if the Flag states exit).</li> <li>• #<a href="#">kPORT_FlagFallingEdge</a> : Flag sets on falling edge(if the Flag states exit).</li> <li>• #<a href="#">kPORT_FlagEitherEdge</a> : Flag sets on either edge(if the Flag states exit).</li> <li>• <a href="#">kPORT_InterruptLogicZero</a> : Interrupt when logic zero.</li> <li>• <a href="#">kPORT_InterruptRisingEdge</a> : Interrupt on rising edge.</li> <li>• <a href="#">kPORT_InterruptFallingEdge</a>: Interrupt on falling edge.</li> <li>• <a href="#">kPORT_InterruptEitherEdge</a> : Interrupt on either edge.</li> <li>• <a href="#">kPORT_InterruptLogicOne</a> : Interrupt when logic one.</li> <li>• #<a href="#">kPORT_ActiveHighTriggerOutputEnable</a> : Enable active high-trigger output (if the trigger states exit).</li> <li>• #<a href="#">kPORT_ActiveLowTriggerOutputEnable</a> : Enable active low-trigger output (if the trigger states exit).</li> </ul>

### 25.6.7 static uint32\_t PORT\_GetPinsInterruptFlags ( PORT\_Type \* *base* ) [inline], [static]

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

<i>base</i>	PORT peripheral base pointer.
-------------	-------------------------------

Returns

Current port interrupt status flags, for example, 0x00010001 means the pin 0 and 16 have the interrupt.

## Function Documentation

25.6.8 **static void PORT\_ClearPinsInterruptFlags ( PORT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

## Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.

## Function Documentation

# Chapter 26

## RCM: Reset Control Module Driver

### 26.1 Overview

The MCUXpresso SDK provides a Peripheral driver for the Reset Control Module (RCM) module of MCUXpresso SDK devices.

### Data Structures

- struct `rcm_reset_pin_filter_config_t`  
*Reset pin filter configuration. [More...](#)*

### Enumerations

- enum `rcm_reset_source_t`{  
  `kRCM_SourceWakeup` = RCM\_SRS0\_WAKEUP\_MASK,  
  `kRCM_SourceLvd` = RCM\_SRS0\_LVD\_MASK,  
  `kRCM_SourceLoc` = RCM\_SRS0\_LOC\_MASK,  
  `kRCM_SourceLol` = RCM\_SRS0\_LOL\_MASK,  
  `kRCM_SourceWdog` = RCM\_SRS0\_WDOG\_MASK,  
  `kRCM_SourcePin` = RCM\_SRS0\_PIN\_MASK,  
  `kRCM_SourcePor` = RCM\_SRS0\_POR\_MASK,  
  `kRCM_SourceJtag` = RCM\_SRS1\_JTAG\_MASK << 8U,  
  `kRCM_SourceLockup` = RCM\_SRS1\_LOCKUP\_MASK << 8U,  
  `kRCM_SourceSw` = RCM\_SRS1\_SW\_MASK << 8U,  
  `kRCM_SourceMdmap` = RCM\_SRS1\_MDM\_AP\_MASK << 8U,  
  `kRCM_SourceEzpt` = RCM\_SRS1\_EZPT\_MASK << 8U,  
  `kRCM_SourceSackerr` = RCM\_SRS1\_SACKERR\_MASK << 8U }  
  *System Reset Source Name definitions.*  
}
- enum `rcm_run_wait_filter_mode_t`{  
  `kRCM_FilterDisable` = 0U,  
  `kRCM_FilterBusClock` = 1U,  
  `kRCM_FilterLpoClock` = 2U }  
  *Reset pin filter select in Run and Wait modes.*

### Driver version

- #define `FSL_RCM_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 1))  
*RCM driver version 2.0.1.*

### Reset Control Module APIs

- static uint32\_t `RCM_GetPreviousResetSources` (RCM\_Type \*base)

## Enumeration Type Documentation

- Gets the reset source status which caused a previous reset.  
void [RCM\\_ConfigureResetPinFilter](#) (RCM\_Type \*base, const [rcm\\_reset\\_pin\\_filter\\_config\\_t](#) \*config)  
Configures the reset pin filter.
- static bool [RCM\\_GetEasyPortModePinStatus](#) (RCM\_Type \*base)  
Gets the EZP\_MS\_B pin assert status.

## 26.2 Data Structure Documentation

### 26.2.1 struct rcm\_reset\_pin\_filter\_config\_t

#### Data Fields

- bool [enableFilterInStop](#)  
Reset pin filter select in stop mode.
- [rcm\\_run\\_wait\\_filter\\_mode\\_t](#) [filterInRunWait](#)  
Reset pin filter in run/wait mode.
- uint8\_t [busClockFilterCount](#)  
Reset pin bus clock filter width.

#### 26.2.1.0.0.63 Field Documentation

##### 26.2.1.0.0.63.1 bool rcm\_reset\_pin\_filter\_config\_t::enableFilterInStop

##### 26.2.1.0.0.63.2 rcm\_run\_wait\_filter\_mode\_t rcm\_reset\_pin\_filter\_config\_t::filterInRunWait

##### 26.2.1.0.0.63.3 uint8\_t rcm\_reset\_pin\_filter\_config\_t::busClockFilterCount

## 26.3 Macro Definition Documentation

### 26.3.1 #define FSL\_RCM\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))

## 26.4 Enumeration Type Documentation

### 26.4.1 enum rcm\_reset\_source\_t

Enumerator

*kRCM\_SourceWakeup* Low-leakage wakeup reset.

*kRCM\_SourceLvd* Low-voltage detect reset.

*kRCM\_SourceLoc* Loss of clock reset.

*kRCM\_SourceLol* Loss of lock reset.

*kRCM\_SourceWdog* Watchdog reset.

*kRCM\_SourcePin* External pin reset.

*kRCM\_SourcePor* Power on reset.

*kRCM\_SourceJtag* JTAG generated reset.

*kRCM\_SourceLockup* Core lock up reset.

*kRCM\_SourceSw* Software reset.

*kRCM\_SourceMdmap* MDM-AP system reset.

*kRCM\_SourceEzpt* EzPort reset.

*kRCM\_SourceSackerr* Parameter could get all reset flags.

## 26.4.2 enum rcm\_run\_wait\_filter\_mode\_t

Enumerator

*kRCM\_FilterDisable* All filtering disabled.

*kRCM\_FilterBusClock* Bus clock filter enabled.

*kRCM\_FilterLpoClock* LPO clock filter enabled.

## 26.5 Function Documentation

### 26.5.1 static uint32\_t RCM\_GetPreviousResetSources ( RCM\_Type \* *base* ) [inline], [static]

This function gets the current reset source status. Use source masks defined in the rcm\_reset\_source\_t to get the desired source status.

This is an example.

```
uint32_t resetStatus;

// To get all reset source statuses.
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;

// To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetPreviousResetSources(RCM) &
kRCM_SourceWdog;

// To test multiple reset sources.
resetStatus = RCM_GetPreviousResetSources(RCM) & (
kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

<i>base</i>	RCM peripheral base address.
-------------	------------------------------

Returns

All reset source status bit map.

### 26.5.2 void RCM\_ConfigureResetPinFilter ( RCM\_Type \* *base*, const rcm\_reset\_pin\_filter\_config\_t \* *config* )

This function sets the reset pin filter including the filter source, filter width, and so on.

## Function Documentation

Parameters

<i>base</i>	RCM peripheral base address.
<i>config</i>	Pointer to the configuration structure.

### 26.5.3 **static bool RCM\_GetEasyPortModePinStatus ( RCM\_Type \* *base* ) [inline], [static]**

This function gets the easy port mode status (EZP\_MS\_B) pin assert status.

Parameters

<i>base</i>	RCM peripheral base address.
-------------	------------------------------

Returns

status true - asserted, false - reasserted

# Chapter 27

## RNGA: Random Number Generator Accelerator Driver

### 27.1 Overview

The MCUXpresso SDK provides Peripheral driver for the Random Number Generator Accelerator (RNGA) block of MCUXpresso SDK devices.

### 27.2 RNGA Initialization

1. To initialize the RNGA module, call the [RNGA\\_Init\(\)](#) function. This function automatically enables the RNGA module and its clock.
2. After calling the [RNGA\\_Init\(\)](#) function, the RNGA is enabled and the counter starts working.
3. To disable the RNGA module, call the [RNGA\\_Deinit\(\)](#) function.

### 27.3 Get random data from RNGA

1. [RNGA\\_GetRandomData\(\)](#) function gets random data from the RNGA module.

### 27.4 RNGA Set/Get Working Mode

The RNGA works either in sleep mode or normal mode

1. [RNGA\\_SetMode\(\)](#) function sets the RNGA mode.
2. [RNGA\\_GetMode\(\)](#) function gets the RNGA working mode.

### 27.5 Seed RNGA

1. [RNGA\\_Seed\(\)](#) function inputs an entropy value that the RNGA can use to seed the pseudo random algorithm.

This example code shows how to initialize and get random data from the RNGA driver:

```
{\n    status_t          status;\n    uint32_t         data;\n\n    /* Initialize RNGA */\n    status = RNGA_Init(RNG);\n\n    /* Read Random data*/\n    status = RNGA_GetRandomData(RNG, data, sizeof(data));\n\n    if(status == kStatus_Success)\n    {\n        /* Print data*/\n        PRINTF("Random = 0x%X\r\n", i, data);\n        PRINTF("Succeed.\r\n");\n    }\n    else\n    {\n\n    }\n}
```

## Seed RNGA

```
    PRINTF("RNGA failed! (0x%lx)\r\n", status);
}

/* Deinitialize RNGA */
RNGA_Deinit(RNG);
}
```

### Note

It is important to note that there is no known cryptographic proof showing this is a secure method for generating random data. In fact, there may be an attack against this random number generator if its output is used directly in a cryptographic application. The attack is based on the linearity of the internal shift registers. Therefore, it is highly recommended that the random data produced by this module be used as an entropy source to provide an input seed to a NIST-approved pseudo-random-number generator based on DES or SHA-1 and defined in NIST FIPS PUB 186-2 Appendix 3 and NIST FIPS PUB SP 800-90. The requirement is needed to maximize the entropy of this input seed. To do this, when data is extracted from RNGA as quickly as the hardware allows, there are one to two bits of added entropy per 32-bit word. Any single bit of that word contains that entropy. Therefore, when used as an entropy source, a random number should be generated for each bit of entropy required and the least significant bit (any bit would be equivalent) of each word retained. The remainder of each random number should then be discarded. Used this way, even with full knowledge of the internal state of RNGA and all prior random numbers, an attacker is not able to predict the values of the extracted bits. Other sources of entropy can be used along with RNGA to generate the seed to the pseudorandom algorithm. The more random sources combined to create the seed, the better. The following is a list of sources that can be easily combined with the output of this module.

- Current time using highest precision possible
- Real-time system inputs that can be characterized as "random"
- Other entropy supplied directly by the user

## Enumerations

- enum `rnga_mode_t` {  
  `kRNGA_ModeNormal` = 0U,  
  `kRNGA_ModeSleep` = 1U }  
*RNGA working mode.*

## Functions

- void `RNGA_Init` (RNG\_Type \*base)  
*Initializes the RNGA.*
- void `RNGA_Deinit` (RNG\_Type \*base)  
*Shuts down the RNGA.*
- status\_t `RNGA_GetRandomData` (RNG\_Type \*base, void \*data, size\_t data\_size)  
*Gets random data.*
- void `RNGA_Seed` (RNG\_Type \*base, uint32\_t seed)  
*Feeds the RNGA module.*
- void `RNGA_SetMode` (RNG\_Type \*base, `rnga_mode_t` mode)

- Sets the RNGA in normal mode or sleep mode.
- **rnga\_mode\_t RNGA\_GetMode (RNG\_Type \*base)**  
Gets the RNGA working mode.

## Driver version

- #define **FSL\_RNGA\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))**  
RNGA driver version 2.0.1.

## 27.6 Macro Definition Documentation

### 27.6.1 #define FSL\_RNGA\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))

## 27.7 Enumeration Type Documentation

### 27.7.1 enum rnga\_mode\_t

Enumerator

**kRNGA\_ModeNormal** Normal Mode. The ring-oscillator clocks are active; RNGA generates entropy (randomness) from the clocks and stores it in shift registers.

**kRNGA\_ModeSleep** Sleep Mode. The ring-oscillator clocks are inactive; RNGA does not generate entropy.

## 27.8 Function Documentation

### 27.8.1 void RNGA\_Init ( RNG\_Type \* *base* )

This function initializes the RNGA. When called, the RNGA entropy generation starts immediately.

Parameters

<i>base</i>	RNGA base address
-------------	-------------------

### 27.8.2 void RNGA\_Deinit ( RNG\_Type \* *base* )

This function shuts down the RNGA.

Parameters

## Function Documentation

<i>base</i>	RNGA base address
-------------	-------------------

### 27.8.3 **status\_t RNGA\_GetRandomData ( RNG\_Type \* *base*, void \* *data*, size\_t *data\_size* )**

This function gets random data from the RNGA.

Parameters

<i>base</i>	RNGA base address
<i>data</i>	pointer to user buffer to be filled by random data
<i>data_size</i>	size of data in bytes

Returns

RNGA status

### 27.8.4 **void RNGA\_Seed ( RNG\_Type \* *base*, uint32\_t *seed* )**

This function inputs an entropy value that the RNGA uses to seed its pseudo-random algorithm.

Parameters

<i>base</i>	RNGA base address
<i>seed</i>	input seed value

### 27.8.5 **void RNGA\_SetMode ( RNG\_Type \* *base*, rnga\_mode\_t *mode* )**

This function sets the RNGA in sleep mode or normal mode.

Parameters

<i>base</i>	RNGA base address
-------------	-------------------

<i>mode</i>	normal mode or sleep mode
-------------	---------------------------

## 27.8.6 **rnga\_mode\_t RNGA\_GetMode ( RNG\_Type \* *base* )**

This function gets the RNGA working mode.

Parameters

<i>base</i>	RNGA base address
-------------	-------------------

Returns

normal mode or sleep mode

## Function Documentation

# Chapter 28

## RTC: Real Time Clock

### 28.1 Overview

The MCUXpresso SDK provides a driver for the Real Time Clock (RTC) of MCUXpresso SDK devices.

### 28.2 Function groups

The RTC driver supports operating the module as a time counter.

#### 28.2.1 Initialization and deinitialization

The function [RTC\\_Init\(\)](#) initializes the RTC with specified configurations. The function [RTC\\_GetDefaultConfig\(\)](#) gets the default configurations.

The function [RTC\\_Deinit\(\)](#) disables the RTC timer and disables the module clock.

#### 28.2.2 Set & Get Datetime

The function [RTC\\_SetDatetime\(\)](#) sets the timer period in seconds. Users pass in the details in date & time format by using the below data structure.

```
typedef struct _rtc_datetime
{
    uint16_t year;
    uint8_t month;
    uint8_t day;
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
} rtc_datetime_t;
```

The function [RTC\\_GetDatetime\(\)](#) reads the current timer value in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

#### 28.2.3 Set & Get Alarm

The function [RTC\\_SetAlarm\(\)](#) sets the alarm time period in seconds. Users pass in the details in date & time format by using the datetime data structure.

The function [RTC\\_GetAlarm\(\)](#) reads the alarm time in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

## Typical use case

### 28.2.4 Start & Stop timer

The function [RTC\\_StartTimer\(\)](#) starts the RTC time counter.

The function [RTC\\_StopTimer\(\)](#) stops the RTC time counter.

### 28.2.5 Status

Provides functions to get and clear the RTC status.

### 28.2.6 Interrupt

Provides functions to enable/disable RTC interrupts and get current enabled interrupts.

### 28.2.7 RTC Oscillator

Some SoC's allow control of the RTC oscillator through the RTC module.

The function [RTC\\_SetOscCapLoad\(\)](#) allows the user to modify the capacitor load configuration of the RTC oscillator.

### 28.2.8 Monotonic Counter

Some SoC's have a 64-bit Monotonic counter available in the RTC module.

The function [RTC\\_SetMonotonicCounter\(\)](#) writes a 64-bit to the counter.

The function [RTC\\_GetMonotonicCounter\(\)](#) reads the monotonic counter and returns the 64-bit counter value to the user.

The function [RTC\\_IncrementMonotonicCounter\(\)](#) increments the Monotonic Counter by one.

## 28.3 Typical use case

### 28.3.1 RTC tick example

Example to set the RTC current time and trigger an alarm.

```
int main(void)
{
    uint32_t sec;
    uint32_t currSeconds;
    rtc_datetime_t date;
    rtc_config_t rtcConfig;

    /* Board pin, clock, debug console init */

    /* Set the RTC current time and trigger an alarm */

    /* ... */

    /* Main loop */
    while(1)
    {
        /* ... */
    }
}
```

```

BOARD_InitHardware();
/* Init RTC */
RTC_GetDefaultConfig(&rtcConfig);
RTC_Init(RTC, &rtcConfig);
/* Select RTC clock source */
BOARD_SetRtcClockSource();

PRINTF("RTC example: set up time to wake up an alarm\r\n");

/* Set a start date time and start RT */
date.year = 2014U;
date.month = 12U;
date.day = 25U;
date.hour = 19U;
date.minute = 0;
date.second = 0;

/* RTC time counter has to be stopped before setting the date & time in the TSR register */
RTC_StopTimer(RTC);

/* Set RTC time to default */
RTC_SetDatetime(RTC, &date);

/* Enable RTC alarm interrupt */
RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);

/* Enable at the NVIC */
EnableIRQ(RTC IRQn);

/* Start the RTC time counter */
RTC_StartTimer(RTC);

/* This loop will set the RTC alarm */
while (1)
{
    busyWait = true;
    /* Get date time */
    RTC_GetDatetime(RTC, &date);

    /* print default time */
    PRINTF("Current datetime: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day, date.hour,
        date.minute, date.second);

    /* Get alarm time from the user */
    sec = 0;
    PRINTF("Input the number of second to wait for alarm \r\n");
    PRINTF("The second must be positive value\r\n");
    while (sec < 1)
    {
        SCANF("%d", &sec);
    }

    /* Read the RTC seconds register to get current time in seconds */
    currSeconds = RTC->TSR;

    /* Add alarm seconds to current time */
    currSeconds += sec;

    /* Set alarm time in seconds */
    RTC->TAR = currSeconds;

    /* Get alarm time */
    RTC_GetAlarm(RTC, &date);

    /* Print alarm time */
    PRINTF("Alarm will occur at: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day,

```

## Typical use case

```
    date.hour, date.minute, date.second);

    /* Wait until alarm occurs */
    while (busyWait)
    {
    }

    PRINTF("\r\n Alarm occurs !!!! ");
}
}
```

## Data Structures

- struct `rtc_datetime_t`  
*Structure is used to hold the date and time.* [More...](#)
- struct `rtc_config_t`  
*RTC config structure.* [More...](#)

## Enumerations

- enum `rtc_interrupt_enable_t` {  
    kRTC\_TimeInvalidInterruptEnable = RTC\_IER\_TIE\_MASK,  
    kRTC\_TimeOverflowInterruptEnable = RTC\_IER\_TOIE\_MASK,  
    kRTC\_AlarmInterruptEnable = RTC\_IER\_TAIE\_MASK,  
    kRTC\_SecondsInterruptEnable = RTC\_IER\_TSIE\_MASK }  
*List of RTC interrupts.*
- enum `rtc_status_flags_t` {  
    kRTC\_TimeInvalidFlag = RTC\_SR\_TIF\_MASK,  
    kRTC\_TimeOverflowFlag = RTC\_SR\_TOF\_MASK,  
    kRTC\_AlarmFlag = RTC\_SR\_TAF\_MASK }  
*List of RTC flags.*
- enum `rtc_osc_cap_load_t` {  
    kRTC\_Capacitor\_2p = RTC\_CR\_SC2P\_MASK,  
    kRTC\_Capacitor\_4p = RTC\_CR\_SC4P\_MASK,  
    kRTC\_Capacitor\_8p = RTC\_CR\_SC8P\_MASK,  
    kRTC\_Capacitor\_16p = RTC\_CR\_SC16P\_MASK }  
*List of RTC Oscillator capacitor load settings.*

## Functions

- static void `RTC_SetOscCapLoad` (RTC\_Type \*base, uint32\_t capLoad)  
*This function sets the specified capacitor configuration for the RTC oscillator.*
- static void `RTC_Reset` (RTC\_Type \*base)  
*Performs a software reset on the RTC module.*

## Driver version

- #define `FSL_RTC_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 0))  
*Version 2.0.0.*

## Initialization and deinitialization

- void **RTC\_Init** (RTC\_Type \*base, const **rtc\_config\_t** \*config)  
*Ungates the RTC clock and configures the peripheral for basic operation.*
- static void **RTC\_Deinit** (RTC\_Type \*base)  
*Stops the timer and gate the RTC clock.*
- void **RTC\_GetDefaultConfig** (**rtc\_config\_t** \*config)  
*Fills in the RTC config struct with the default settings.*

## Current Time & Alarm

- status\_t **RTC\_SetDatetime** (RTC\_Type \*base, const **rtc\_datetime\_t** \*datetime)  
*Sets the RTC date and time according to the given time structure.*
- void **RTC\_GetDatetime** (RTC\_Type \*base, **rtc\_datetime\_t** \*datetime)  
*Gets the RTC time and stores it in the given time structure.*
- status\_t **RTC\_SetAlarm** (RTC\_Type \*base, const **rtc\_datetime\_t** \*alarmTime)  
*Sets the RTC alarm time.*
- void **RTC\_GetAlarm** (RTC\_Type \*base, **rtc\_datetime\_t** \*datetime)  
*Returns the RTC alarm time.*

## Interrupt Interface

- static void **RTC\_EnableInterrupts** (RTC\_Type \*base, uint32\_t mask)  
*Enables the selected RTC interrupts.*
- static void **RTC\_DisableInterrupts** (RTC\_Type \*base, uint32\_t mask)  
*Disables the selected RTC interrupts.*
- static uint32\_t **RTC\_GetEnabledInterrupts** (RTC\_Type \*base)  
*Gets the enabled RTC interrupts.*

## Status Interface

- static uint32\_t **RTC\_GetStatusFlags** (RTC\_Type \*base)  
*Gets the RTC status flags.*
- void **RTC\_ClearStatusFlags** (RTC\_Type \*base, uint32\_t mask)  
*Clears the RTC status flags.*

## Timer Start and Stop

- static void **RTC\_StartTimer** (RTC\_Type \*base)  
*Starts the RTC time counter.*
- static void **RTC\_StopTimer** (RTC\_Type \*base)  
*Stops the RTC time counter.*

## Monotonic counter functions

- void **RTC\_GetMonotonicCounter** (RTC\_Type \*base, uint64\_t \*counter)  
*Reads the values of the Monotonic Counter High and Monotonic Counter Low and returns them as a single value.*
- void **RTC\_SetMonotonicCounter** (RTC\_Type \*base, uint64\_t counter)  
*Writes values Monotonic Counter High and Monotonic Counter Low by decomposing the given single value.*

## Data Structure Documentation

- status\_t [RTC\\_IncrementMonotonicCounter](#) (RTC\_Type \*base)  
*Increments the Monotonic Counter by one.*

### 28.4 Data Structure Documentation

#### 28.4.1 struct rtc\_datetime\_t

##### Data Fields

- uint16\_t [year](#)  
*Range from 1970 to 2099.*
- uint8\_t [month](#)  
*Range from 1 to 12.*
- uint8\_t [day](#)  
*Range from 1 to 31 (depending on month).*
- uint8\_t [hour](#)  
*Range from 0 to 23.*
- uint8\_t [minute](#)  
*Range from 0 to 59.*
- uint8\_t [second](#)  
*Range from 0 to 59.*

##### 28.4.1.0.0.64 Field Documentation

###### 28.4.1.0.0.64.1 uint16\_t rtc\_datetime\_t::year

###### 28.4.1.0.0.64.2 uint8\_t rtc\_datetime\_t::month

###### 28.4.1.0.0.64.3 uint8\_t rtc\_datetime\_t::day

###### 28.4.1.0.0.64.4 uint8\_t rtc\_datetime\_t::hour

###### 28.4.1.0.0.64.5 uint8\_t rtc\_datetime\_t::minute

###### 28.4.1.0.0.64.6 uint8\_t rtc\_datetime\_t::second

#### 28.4.2 struct rtc\_config\_t

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the [RTC\\_GetDefaultConfig\(\)](#) function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

##### Data Fields

- bool [wakeupSelect](#)  
*true: Wakeup pin outputs the 32 KHz clock; false:Wakeup pin used to wakeup the chip*

- bool `updateMode`  
*true: Registers can be written even when locked under certain conditions, false: No writes allowed when registers are locked*
- bool `supervisorAccess`  
*true: Non-supervisor accesses are allowed; false: Non-supervisor accesses are not supported*
- uint32\_t `compensationInterval`  
*Compensation interval that is written to the CIR field in RTC TCR Register.*
- uint32\_t `compensationTime`  
*Compensation time that is written to the TCR field in RTC TCR Register.*

## 28.5 Enumeration Type Documentation

### 28.5.1 enum rtc\_interrupt\_enable\_t

Enumerator

- `kRTC_TimeInvalidInterruptEnable`** Time invalid interrupt.
- `kRTC_TimeOverflowInterruptEnable`** Time overflow interrupt.
- `kRTC_AlarmInterruptEnable`** Alarm interrupt.
- `kRTC_SecondsInterruptEnable`** Seconds interrupt.

### 28.5.2 enum rtc\_status\_flags\_t

Enumerator

- `kRTC_TimeInvalidFlag`** Time invalid flag.
- `kRTC_TimeOverflowFlag`** Time overflow flag.
- `kRTC_AlarmFlag`** Alarm flag.

### 28.5.3 enum rtc\_osc\_cap\_load\_t

Enumerator

- `kRTC_Capacitor_2p`** 2 pF capacitor load
- `kRTC_Capacitor_4p`** 4 pF capacitor load
- `kRTC_Capacitor_8p`** 8 pF capacitor load
- `kRTC_Capacitor_16p`** 16 pF capacitor load

## 28.6 Function Documentation

### 28.6.1 void RTC\_Init ( RTC\_Type \* *base*, const rtc\_config\_t \* *config* )

This function issues a software reset if the timer invalid flag is set.

## Function Documentation

### Note

This API should be called at the beginning of the application using the RTC driver.

### Parameters

<i>base</i>	RTC peripheral base address
<i>config</i>	Pointer to the user's RTC configuration structure.

### 28.6.2 static void RTC\_Deinit ( RTC\_Type \* *base* ) [inline], [static]

### Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

### 28.6.3 void RTC\_GetDefaultConfig ( rtc\_config\_t \* *config* )

The default values are as follows.

```
*     config->wakeupSelect = false;
*     config->updateMode = false;
*     config->supervisorAccess = false;
*     config->compensationInterval = 0;
*     config->compensationTime = 0;
*
```

### Parameters

<i>config</i>	Pointer to the user's RTC configuration structure.
---------------	--

### 28.6.4 status\_t RTC\_SetDatetime ( RTC\_Type \* *base*, const rtc\_datetime\_t \* *datetime* )

The RTC counter must be stopped prior to calling this function because writes to the RTC seconds register fail if the RTC counter is running.

Parameters

<i>base</i>	RTC peripheral base address
<i>datetime</i>	Pointer to the structure where the date and time details are stored.

Returns

kStatus\_Success: Success in setting the time and starting the RTC  
kStatus\_InvalidArgument: Error because the datetime format is incorrect

### 28.6.5 void RTC\_GetDatetime ( RTC\_Type \* *base*, rtc\_datetime\_t \* *datetime* )

Parameters

<i>base</i>	RTC peripheral base address
<i>datetime</i>	Pointer to the structure where the date and time details are stored.

### 28.6.6 status\_t RTC\_SetAlarm ( RTC\_Type \* *base*, const rtc\_datetime\_t \* *alarmTime* )

The function checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

<i>base</i>	RTC peripheral base address
<i>alarmTime</i>	Pointer to the structure where the alarm time is stored.

Returns

kStatus\_Success: success in setting the RTC alarm  
kStatus\_InvalidArgument: Error because the alarm datetime format is incorrect  
kStatus\_Fail: Error because the alarm time has already passed

### 28.6.7 void RTC\_GetAlarm ( RTC\_Type \* *base*, rtc\_datetime\_t \* *datetime* )

## Function Documentation

Parameters

<i>base</i>	RTC peripheral base address
<i>datetime</i>	Pointer to the structure where the alarm date and time details are stored.

**28.6.8 static void RTC\_EnableInterrupts ( RTC\_Type \* *base*, uint32\_t *mask* )  
[inline], [static]**

Parameters

<i>base</i>	RTC peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">rtc_interrupt_enable_t</a>

**28.6.9 static void RTC\_DisableInterrupts ( RTC\_Type \* *base*, uint32\_t *mask* )  
[inline], [static]**

Parameters

<i>base</i>	RTC peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">rtc_interrupt_enable_t</a>

**28.6.10 static uint32\_t RTC\_GetEnabledInterrupts ( RTC\_Type \* *base* )  
[inline], [static]**

Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [rtc\\_interrupt\\_enable\\_t](#)

**28.6.11 static uint32\_t RTC\_GetStatusFlags ( RTC\_Type \* *base* ) [inline],  
[static]**

Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

Returns

The status flags. This is the logical OR of members of the enumeration [rtc\\_status\\_flags\\_t](#)

### 28.6.12 void RTC\_ClearStatusFlags ( RTC\_Type \* *base*, uint32\_t *mask* )

Parameters

<i>base</i>	RTC peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration <a href="#">rtc_status_flags_t</a>

### 28.6.13 static void RTC\_StartTimer ( RTC\_Type \* *base* ) [inline], [static]

After calling this function, the timer counter increments once a second provided SR[TOF] or SR[TIF] are not set.

Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

### 28.6.14 static void RTC\_StopTimer ( RTC\_Type \* *base* ) [inline], [static]

RTC's seconds register can be written to only when the timer is stopped.

Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

### 28.6.15 static void RTC\_SetOscCapLoad ( RTC\_Type \* *base*, uint32\_t *capLoad* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	RTC peripheral base address
<i>capLoad</i>	Oscillator loads to enable. This is a logical OR of members of the enumeration <a href="#">rtc_osc_cap_load_t</a>

### 28.6.16 static void RTC\_Reset( RTC\_Type \* *base* ) [inline], [static]

This resets all RTC registers except for the SWR bit and the RTC\_WAR and RTC\_RAR registers. The SWR bit is cleared by software explicitly clearing it.

Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

### 28.6.17 void RTC\_GetMonotonicCounter( RTC\_Type \* *base*, uint64\_t \* *counter* )

Parameters

<i>base</i>	RTC peripheral base address
<i>counter</i>	Pointer to variable where the value is stored.

### 28.6.18 void RTC\_SetMonotonicCounter( RTC\_Type \* *base*, uint64\_t *counter* )

Parameters

<i>base</i>	RTC peripheral base address
<i>counter</i>	Counter value

### 28.6.19 status\_t RTC\_IncrementMonotonicCounter( RTC\_Type \* *base* )

Increments the Monotonic Counter (registers RTC\_MCLR and RTC\_MCHR accordingly) by setting the monotonic counter enable (MER[MCE]) and then writing to the RTC\_MCLR register. A write to the monotonic counter low that causes it to overflow also increments the monotonic counter high.

### Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

### Returns

kStatus\_Success: success  
kStatus\_Fail: error occurred, either time invalid or monotonic overflow flag was found

## Function Documentation

# Chapter 29

## SAI: Serial Audio Interface

### 29.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Serial Audio Interface (SAI) module of MCUXpresso SDK devices.

SAI driver includes functional APIs and transactional APIs.

Functional APIs target low-level APIs. Functional APIs can be used for SAI initialization, configuration and operation, and for optimization and customization purposes. Using the functional API requires the knowledge of the SAI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. SAI functional operation groups provide the functional API set.

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the sai\_handle\_t as the first parameter. Initialize the handle by calling the [SAI\\_TransferTxCreateHandle\(\)](#) or [SAI\\_TransferRxCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer. This means that the functions [SAI\\_TransferSendNonBlocking\(\)](#) and [SAI\\_TransferReceiveNonBlocking\(\)](#) set up the interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus\_SAI\_TxIdle and kStatus\_SAI\_RxIdle status.

### 29.2 Typical use case

#### 29.2.1 SAI Send/receive using an interrupt method

```
sai_handle_t g_saiTxHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
volatile bool rxFinished;
const uint8_t sendData[] = [.....];

void SAI_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_SAI_TxIdle == status)
    {
        txFinished = true;
    }
}

void main(void)
{
    //...
```

## Typical use case

```
SAI_TxGetDefaultConfig(&user_config);

SAI_TxInit(SAI0, &user_config);
SAI_TransferTxCreateHandle(SAI0, &g_saiHandle, SAI_UserCallback, NULL);

//Configure sai format
SAI_TransferTxSetTransferFormat(SAI0, &g_saiHandle, mclkSource, mclk);

// Prepare to send.
sendXfer.data = sendData
sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
txFinished = false;

// Send out.
SAI_TransferSendNonBlocking(SAI0, &g_saiHandle, &sendXfer);

// Wait send finished.
while (!txFinished)
{
}

// ...
}
```

### 29.2.2 SAI Send/receive using a DMA method

```
sai_handle_t g_saiHandle;
dma_handle_t g_saiTxDmaHandle;
dma_handle_t g_saiRxDmaHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
uint8_t sendData[] = ...;

void SAI_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_SAI_TxIdle == status)
    {
        txFinished = true;
    }
}

void main(void)
{
    //...

    SAI_TxGetDefaultConfig(&user_config);
    SAI_TxInit(SAI0, &user_config);

    // Sets up the DMA.
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, SAI_TX_DMA_CHANNEL, SAI_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SAI_TX_DMA_CHANNEL);

    DMA_Init(DMA0);

    /* Creates the DMA handle. */
    DMA_CreateHandle(&g_saiTxDmaHandle, DMA0, SAI_TX_DMA_CHANNEL);

    SAI_TransferTxCreateHandleDMA(SAI0, &g_saiTxDmaHandle, SAI_UserCallback,
        NULL);

    // Prepares to send.
```

```

sendXfer.data = sendData
sendXfer.dataSize = sizeof(sendData) / sizeof(sendData[0]);
txFinished = false;

// Sends out.
SAI_TransferSendDMA(&g_saiHandle, &sendXfer);

// Waits for send to complete.
while (!txFinished)
{
}

// ...
}

```

## Modules

- SAI DMA Driver
- SAI eDMA Driver

## Data Structures

- struct `sai_config_t`  
*SAI user configuration structure.* [More...](#)
- struct `sai_transfer_format_t`  
*sai transfer format* [More...](#)
- struct `sai_transfer_t`  
*SAI transfer structure.* [More...](#)
- struct `sai_handle_t`  
*SAI handle structure.* [More...](#)

## Macros

- #define `SAI_XFER_QUEUE_SIZE` (4)  
*SAI transfer queue size, user can refine it according to use case.*

## Typedefs

- typedef void(\* `sai_transfer_callback_t` )(I2S\_Type \*base, sai\_handle\_t \*handle, status\_t status, void \*userData)  
*SAI transfer callback prototype.*

## Enumerations

- enum `_sai_status_t` {
 `kStatus_SAI_TxBusy` = MAKE\_STATUS(kStatusGroup\_SAI, 0),
 `kStatus_SAI_RxBusy` = MAKE\_STATUS(kStatusGroup\_SAI, 1),
 `kStatus_SAI_TxError` = MAKE\_STATUS(kStatusGroup\_SAI, 2),
 `kStatus_SAI_RxError` = MAKE\_STATUS(kStatusGroup\_SAI, 3),
 `kStatus_SAI_QueueFull` = MAKE\_STATUS(kStatusGroup\_SAI, 4),
 `kStatus_SAI_TxIdle` = MAKE\_STATUS(kStatusGroup\_SAI, 5),
 `kStatus_SAI_RxIdle` = MAKE\_STATUS(kStatusGroup\_SAI, 6) }

## Typical use case

- SAI return status.*
- enum `sai_protocol_t` {  
    `kSAI_BusLeftJustified` = 0x0U,  
    `kSAI_BusRightJustified`,  
    `kSAI_BusI2S`,  
    `kSAI_BusPCMA`,  
    `kSAI_BusPCMB` }  
*Define the SAI bus type.*
  - enum `sai_master_slave_t` {  
    `kSAI_Master` = 0x0U,  
    `kSAI_Slave` = 0x1U }  
*Master or slave mode.*
  - enum `sai_mono_stereo_t` {  
    `kSAI_Stereo` = 0x0U,  
    `kSAI_MonoLeft`,  
    `kSAI_MonoRight` }  
*Mono or stereo audio format.*
  - enum `sai_sync_mode_t` {  
    `kSAI_ModeAsync` = 0x0U,  
    `kSAI_ModeSync`,  
    `kSAI_ModeSyncWithOtherTx`,  
    `kSAI_ModeSyncWithOtherRx` }  
*Synchronous or asynchronous mode.*
  - enum `sai_mclk_source_t` {  
    `kSAI_MclkSourceSysclk` = 0x0U,  
    `kSAI_MclkSourceSelect1`,  
    `kSAI_MclkSourceSelect2`,  
    `kSAI_MclkSourceSelect3` }  
*Mater clock source.*
  - enum `sai_bclk_source_t` {  
    `kSAI_BclkSourceBusclk` = 0x0U,  
    `kSAI_BclkSourceMclkDiv`,  
    `kSAI_BclkSourceOtherSai0`,  
    `kSAI_BclkSourceOtherSai1` }  
*Bit clock source.*
  - enum `_sai_interrupt_enable_t` {  
    `kSAI_WordStartInterruptEnable`,  
    `kSAI_SyncErrorInterruptEnable` = I2S\_TCSR\_SEIE\_MASK,  
    `kSAI_FIFOWarningInterruptEnable` = I2S\_TCSR\_FWIE\_MASK,  
    `kSAI_FIFOErrorInterruptEnable` = I2S\_TCSR\_FEIE\_MASK,  
    `kSAI_FIFORequestInterruptEnable` = I2S\_TCSR\_FRIE\_MASK }  
*The SAI interrupt enable flag.*
  - enum `_sai_dma_enable_t` {  
    `kSAI_FIFOWarningDMAEnable` = I2S\_TCSR\_FWDE\_MASK,  
    `kSAI_FIFORequestDMAEnable` = I2S\_TCSR\_FRDE\_MASK }  
*The DMA request sources.*
  - enum `_sai_flags` {

```
kSAI_WordStartFlag = I2S_TCSR_WSF_MASK,
kSAI_SyncErrorFlag = I2S_TCSR_SEF_MASK,
kSAI_FIFOErrorFlag = I2S_TCSR_FEF_MASK,
kSAI_FIFORequestFlag = I2S_TCSR_FRF_MASK,
kSAI_FIFOWarningFlag = I2S_TCSR_FWF_MASK }
```

*The SAI status flag.*

- enum `sai_reset_type_t` {
   
kSAI\_ResetTypeSoftware = I2S\_TCSR\_SR\_MASK,
   
kSAI\_ResetTypeFIFO = I2S\_TCSR\_FR\_MASK,
   
kSAI\_ResetAll = I2S\_TCSR\_SR\_MASK | I2S\_TCSR\_FR\_MASK }

*The reset type.*

- enum `sai_sample_rate_t` {
   
kSAI\_SampleRate8KHz = 8000U,
   
kSAI\_SampleRate11025Hz = 11025U,
   
kSAI\_SampleRate12KHz = 12000U,
   
kSAI\_SampleRate16KHz = 16000U,
   
kSAI\_SampleRate22050Hz = 22050U,
   
kSAI\_SampleRate24KHz = 24000U,
   
kSAI\_SampleRate32KHz = 32000U,
   
kSAI\_SampleRate44100Hz = 44100U,
   
kSAI\_SampleRate48KHz = 48000U,
   
kSAI\_SampleRate96KHz = 96000U }

*Audio sample rate.*

- enum `sai_word_width_t` {
   
kSAI\_WordWidth8bits = 8U,
   
kSAI\_WordWidth16bits = 16U,
   
kSAI\_WordWidth24bits = 24U,
   
kSAI\_WordWidth32bits = 32U }

*Audio word width.*

## Driver version

- #define `FSL_SAI_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 2)`)

*Version 2.1.2.*

## Initialization and deinitialization

- void `SAI_TxInit` (I2S\_Type \*base, const `sai_config_t` \*config)
   
*Initializes the SAI Tx peripheral.*
- void `SAI_RxInit` (I2S\_Type \*base, const `sai_config_t` \*config)
   
*Initializes the the SAI Rx peripheral.*
- void `SAI_TxGetDefaultConfig` (`sai_config_t` \*config)
   
*Sets the SAI Tx configuration structure to default values.*
- void `SAI_RxGetDefaultConfig` (`sai_config_t` \*config)
   
*Sets the SAI Rx configuration structure to default values.*
- void `SAI_Deinit` (I2S\_Type \*base)
   
*De-initializes the SAI peripheral.*
- void `SAI_TxReset` (I2S\_Type \*base)

## Typical use case

- Resets the SAI Tx.  
void **SAI\_RxReset** (I2S\_Type \*base)  
Resets the SAI Rx.
- void **SAI\_TxEnable** (I2S\_Type \*base, bool enable)  
Enables/disables the SAI Tx.  
• void **SAI\_RxEnable** (I2S\_Type \*base, bool enable)  
Enables/disables the SAI Rx.

## Status

- static uint32\_t **SAI\_TxGetStatusFlag** (I2S\_Type \*base)  
Gets the SAI Tx status flag state.
- static void **SAI\_TxClearStatusFlags** (I2S\_Type \*base, uint32\_t mask)  
Clears the SAI Tx status flag state.
- static uint32\_t **SAI\_RxGetStatusFlag** (I2S\_Type \*base)  
Gets the SAI Rx status flag state.
- static void **SAI\_RxClearStatusFlags** (I2S\_Type \*base, uint32\_t mask)  
Clears the SAI Rx status flag state.

## Interrupts

- static void **SAI\_TxEnableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
Enables the SAI Tx interrupt requests.
- static void **SAI\_RxEnableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
Enables the SAI Rx interrupt requests.
- static void **SAI\_TxDisableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
Disables the SAI Tx interrupt requests.
- static void **SAI\_RxDisableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
Disables the SAI Rx interrupt requests.

## DMA Control

- static void **SAI\_TxEnableDMA** (I2S\_Type \*base, uint32\_t mask, bool enable)  
Enables/disables the SAI Tx DMA requests.
- static void **SAI\_RxEnableDMA** (I2S\_Type \*base, uint32\_t mask, bool enable)  
Enables/disables the SAI Rx DMA requests.
- static uint32\_t **SAI\_TxGetDataRegisterAddress** (I2S\_Type \*base, uint32\_t channel)  
Gets the SAI Tx data register address.
- static uint32\_t **SAI\_RxGetDataRegisterAddress** (I2S\_Type \*base, uint32\_t channel)  
Gets the SAI Rx data register address.

## Bus Operations

- void **SAI\_TxSetFormat** (I2S\_Type \*base, sai\_transfer\_format\_t \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
Configures the SAI Tx audio format.
- void **SAI\_RxSetFormat** (I2S\_Type \*base, sai\_transfer\_format\_t \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
Configures the SAI Rx audio format.
- void **SAI\_WriteBlocking** (I2S\_Type \*base, uint32\_t channel, uint32\_t bitWidth, uint8\_t \*buffer, uint32\_t size)

- static void [SAI\\_WriteData](#) (I2S\_Type \*base, uint32\_t channel, uint32\_t data)
 

*Sends data using a blocking method.*

*Writes data into SAI FIFO.*
- void [SAI\\_ReadBlocking](#) (I2S\_Type \*base, uint32\_t channel, uint32\_t bitWidth, uint8\_t \*buffer, uint32\_t size)
 

*Receives data using a blocking method.*

*Reads data from the SAI FIFO.*

## Transactional

- void [SAI\\_TransferTxCreateHandle](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_callback\\_t](#) callback, void \*userData)
 

*Initializes the SAI Tx handle.*
- void [SAI\\_TransferRxCreateHandle](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_callback\\_t](#) callback, void \*userData)
 

*Initializes the SAI Rx handle.*
- status\_t [SAI\\_TransferTxSetFormat](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_format\\_t](#) \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)
 

*Configures the SAI Tx audio format.*
- status\_t [SAI\\_TransferRxSetFormat](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_format\\_t](#) \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)
 

*Configures the SAI Rx audio format.*
- status\_t [SAI\\_TransferSendNonBlocking](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_t](#) \*xfer)
 

*Performs an interrupt non-blocking send transfer on SAI.*
- status\_t [SAI\\_TransferReceiveNonBlocking](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_t](#) \*xfer)
 

*Performs an interrupt non-blocking receive transfer on SAI.*
- status\_t [SAI\\_TransferGetSendCount](#) (I2S\_Type \*base, sai\_handle\_t \*handle, size\_t \*count)
 

*Gets a set byte count.*
- status\_t [SAI\\_TransferGetReceiveCount](#) (I2S\_Type \*base, sai\_handle\_t \*handle, size\_t \*count)
 

*Gets a received byte count.*
- void [SAI\\_TransferAbortSend](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Aborts the current send.*
- void [SAI\\_TransferAbortReceive](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Aborts the the current IRQ receive.*
- void [SAI\\_TransferTxHandleIRQ](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Tx interrupt handler.*
- void [SAI\\_TransferRxHandleIRQ](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Tx interrupt handler.*

## 29.3 Data Structure Documentation

### 29.3.1 struct sai\_config\_t

#### Data Fields

- [sai\\_protocol\\_t](#) protocol

## Data Structure Documentation

- *Audio bus protocol in SAI.*
  - **sai\_sync\_mode\_t syncMode**  
*SAI sync mode, control Tx/Rx clock sync.*
  - **bool mclkOutputEnable**  
*Master clock output enable, true means master clock divider enabled.*
- **sai\_mclk\_source\_t mclkSource**  
*Master Clock source.*
- **sai\_bclk\_source\_t bclkSource**  
*Bit Clock source.*
- **sai\_master\_slave\_t masterSlave**  
*Master or slave.*

### 29.3.2 struct sai\_transfer\_format\_t

#### Data Fields

- **uint32\_t sampleRate\_Hz**  
*Sample rate of audio data.*
- **uint32\_t bitWidth**  
*Data length of audio data, usually 8/16/24/32 bits.*
- **sai\_mono\_stereo\_t stereo**  
*Mono or stereo.*
- **uint32\_t masterClockHz**  
*Master clock frequency in Hz.*
- **uint8\_t watermark**  
*Watermark value.*
- **uint8\_t channel**  
*Data channel used in transfer.*
- **sai\_protocol\_t protocol**  
*Which audio protocol used.*

#### 29.3.2.0.0.65 Field Documentation

##### 29.3.2.0.0.65.1 uint8\_t sai\_transfer\_format\_t::channel

### 29.3.3 struct sai\_transfer\_t

#### Data Fields

- **uint8\_t \* data**  
*Data start address to transfer.*
- **size\_t dataSize**  
*Transfer size.*

### 29.3.3.0.0.66 Field Documentation

29.3.3.0.0.66.1 `uint8_t* sai_transfer_t::data`

29.3.3.0.0.66.2 `size_t sai_transfer_t::dataSize`

### 29.3.4 struct \_sai\_handle

#### Data Fields

- `uint32_t state`  
*Transfer status.*
- `sai_transfer_callback_t callback`  
*Callback function called at transfer event.*
- `void *userData`  
*Callback parameter passed to callback function.*
- `uint8_t bitWidth`  
*Bit width for transfer, 8/16/24/32 bits.*
- `uint8_t channel`  
*Transfer channel.*
- `sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]`  
*Transfer queue storing queued transfer.*
- `size_t transferSize [SAI_XFER_QUEUE_SIZE]`  
*Data bytes need to transfer.*
- `volatile uint8_t queueUser`  
*Index for user to queue transfer.*
- `volatile uint8_t queueDriver`  
*Index for driver to get the transfer data and size.*
- `uint8_t watermark`  
*Watermark value.*

### 29.4 Macro Definition Documentation

29.4.1 `#define SAI_XFER_QUEUE_SIZE (4)`

### 29.5 Enumeration Type Documentation

#### 29.5.1 enum \_sai\_status\_t

Enumerator

- `kStatus_SAI_TxBusy` SAI Tx is busy.
- `kStatus_SAI_RxBusy` SAI Rx is busy.
- `kStatus_SAI_TxError` SAI Tx FIFO error.
- `kStatus_SAI_RxError` SAI Rx FIFO error.
- `kStatus_SAI_QueueFull` SAI transfer queue is full.
- `kStatus_SAI_TxIdle` SAI Tx is idle.
- `kStatus_SAI_RxIdle` SAI Rx is idle.

## Enumeration Type Documentation

### 29.5.2 enum sai\_protocol\_t

Enumerator

- kSAI\_BusLeftJustified*** Uses left justified format.
- kSAI\_BusRightJustified*** Uses right justified format.
- kSAI\_BusI2S*** Uses I2S format.
- kSAI\_BusPCMA*** Uses I2S PCM A format.
- kSAI\_BusPCMB*** Uses I2S PCM B format.

### 29.5.3 enum sai\_master\_slave\_t

Enumerator

- kSAI\_Master*** Master mode.
- kSAI\_Slave*** Slave mode.

### 29.5.4 enum sai\_mono\_stereo\_t

Enumerator

- kSAI\_Stereo*** Stereo sound.
- kSAI\_MonoLeft*** Only left channel have sound.
- kSAI\_MonoRight*** Only Right channel have sound.

### 29.5.5 enum sai\_sync\_mode\_t

Enumerator

- kSAI\_ModeAsync*** Asynchronous mode.
- kSAI\_ModeSync*** Synchronous mode (with receiver or transmit)
- kSAI\_ModeSyncWithOtherTx*** Synchronous with another SAI transmit.
- kSAI\_ModeSyncWithOtherRx*** Synchronous with another SAI receiver.

### 29.5.6 enum sai\_mclk\_source\_t

Enumerator

- kSAI\_MclkSourceSysclk*** Master clock from the system clock.
- kSAI\_MclkSourceSelect1*** Master clock from source 1.
- kSAI\_MclkSourceSelect2*** Master clock from source 2.
- kSAI\_MclkSourceSelect3*** Master clock from source 3.

### 29.5.7 enum sai\_bclk\_source\_t

Enumerator

- kSAI\_BclkSourceBusclk* Bit clock using bus clock.
- kSAI\_BclkSourceMclkDiv* Bit clock using master clock divider.
- kSAI\_BclkSourceOtherSai0* Bit clock from other SAI device.
- kSAI\_BclkSourceOtherSai1* Bit clock from other SAI device.

### 29.5.8 enum \_sai\_interrupt\_enable\_t

Enumerator

- kSAI\_WordStartInterruptEnable* Word start flag, means the first word in a frame detected.
- kSAI\_SyncErrorInterruptEnable* Sync error flag, means the sync error is detected.
- kSAI\_FIFOWarningInterruptEnable* FIFO warning flag, means the FIFO is empty.
- kSAI\_FIFOErrorInterruptEnable* FIFO error flag.
- kSAI\_FIFORequestInterruptEnable* FIFO request, means reached watermark.

### 29.5.9 enum \_sai\_dma\_enable\_t

Enumerator

- kSAI\_FIFOWarningDMAEnable* FIFO warning caused by the DMA request.
- kSAI\_FIFORequestDMAEnable* FIFO request caused by the DMA request.

### 29.5.10 enum \_sai\_flags

Enumerator

- kSAI\_WordStartFlag* Word start flag, means the first word in a frame detected.
- kSAI\_SyncErrorFlag* Sync error flag, means the sync error is detected.
- kSAI\_FIFOErrorFlag* FIFO error flag.
- kSAI\_FIFORequestFlag* FIFO request flag.
- kSAI\_FIFOWarningFlag* FIFO warning flag.

### 29.5.11 enum sai\_reset\_type\_t

Enumerator

- kSAI\_ResetTypeSoftware* Software reset, reset the logic state.

## Function Documentation

***kSAI\_ResetTypeFIFO*** FIFO reset, reset the FIFO read and write pointer.

***kSAI\_ResetAll*** All reset.

### 29.5.12 enum sai\_sample\_rate\_t

Enumerator

***kSAI\_SampleRate8KHz*** Sample rate 8000 Hz.  
***kSAI\_SampleRate11025Hz*** Sample rate 11025 Hz.  
***kSAI\_SampleRate12KHz*** Sample rate 12000 Hz.  
***kSAI\_SampleRate16KHz*** Sample rate 16000 Hz.  
***kSAI\_SampleRate22050Hz*** Sample rate 22050 Hz.  
***kSAI\_SampleRate24KHz*** Sample rate 24000 Hz.  
***kSAI\_SampleRate32KHz*** Sample rate 32000 Hz.  
***kSAI\_SampleRate44100Hz*** Sample rate 44100 Hz.  
***kSAI\_SampleRate48KHz*** Sample rate 48000 Hz.  
***kSAI\_SampleRate96KHz*** Sample rate 96000 Hz.

### 29.5.13 enum sai\_word\_width\_t

Enumerator

***kSAI\_WordWidth8bits*** Audio data width 8 bits.  
***kSAI\_WordWidth16bits*** Audio data width 16 bits.  
***kSAI\_WordWidth24bits*** Audio data width 24 bits.  
***kSAI\_WordWidth32bits*** Audio data width 32 bits.

## 29.6 Function Documentation

### 29.6.1 void SAI\_TxInit ( I2S\_Type \* *base*, const sai\_config\_t \* *config* )

Ungates the SAI clock, resets the module, and configures SAI Tx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI\\_TxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAIM module can cause a hard fault because the clock is not enabled.

Parameters

<i>base</i>	SAI base pointer
<i>config</i>	SAI configuration structure.

## 29.6.2 void SAI\_RxInit ( I2S\_Type \* *base*, const sai\_config\_t \* *config* )

Ungates the SAI clock, resets the module, and configures the SAI Rx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI\\_RxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAI module can cause a hard fault because the clock is not enabled.

Parameters

<i>base</i>	SAI base pointer
<i>config</i>	SAI configuration structure.

## 29.6.3 void SAI\_TxGetDefaultConfig ( sai\_config\_t \* *config* )

This API initializes the configuration structure for use in SAI\_TxConfig(). The initialized structure can remain unchanged in SAI\_TxConfig(), or it can be modified before calling SAI\_TxConfig(). This is an example.

```
sai_config_t config;
SAI_TxGetDefaultConfig(&config);
```

Parameters

<i>config</i>	pointer to master configuration structure
---------------	---

## 29.6.4 void SAI\_RxGetDefaultConfig ( sai\_config\_t \* *config* )

This API initializes the configuration structure for use in SAI\_RxConfig(). The initialized structure can remain unchanged in SAI\_RxConfig() or it can be modified before calling SAI\_RxConfig(). This is an example.

```
sai_config_t config;
SAI_RxGetDefaultConfig(&config);
```

## Function Documentation

Parameters

<i>config</i>	pointer to master configuration structure
---------------	---

### 29.6.5 void SAI\_Deinit ( I2S\_Type \* *base* )

This API gates the SAI clock. The SAI module can't operate unless SAI\_TxInit or SAI\_RxInit is called to enable the clock.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

### 29.6.6 void SAI\_TxReset ( I2S\_Type \* *base* )

This function enables the software reset and FIFO reset of SAI Tx. After reset, clear the reset bit.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

### 29.6.7 void SAI\_RxReset ( I2S\_Type \* *base* )

This function enables the software reset and FIFO reset of SAI Rx. After reset, clear the reset bit.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

### 29.6.8 void SAI\_TxEnable ( I2S\_Type \* *base*, bool *enable* )

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

<i>enable</i>	True means enable SAI Tx, false means disable.
---------------	--

**29.6.9 void SAI\_RxEnable ( I2S\_Type \* *base*, bool *enable* )**

Parameters

<i>base</i>	SAI base pointer
<i>enable</i>	True means enable SAI Rx, false means disable.

**29.6.10 static uint32\_t SAI\_TxGetStatusFlag ( I2S\_Type \* *base* ) [inline], [static]**

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Returns

SAI Tx status flag value. Use the Status Mask to get the status value needed.

**29.6.11 static void SAI\_TxClearStatusFlags ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	State mask. It can be a combination of the following source if defined: <ul style="list-style-type: none"><li>• kSAI_WordStartFlag</li><li>• kSAI_SyncErrorFlag</li><li>• kSAI_FIFOErrorFlag</li></ul>

**29.6.12 static uint32\_t SAI\_RxGetStatusFlag ( I2S\_Type \* *base* ) [inline], [static]**

## Function Documentation

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Returns

SAI Rx status flag value. Use the Status Mask to get the status value needed.

### 29.6.13 static void SAI\_RxClearStatusFlags ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	State mask. It can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_WordStartFlag</li><li>• kSAI_SyncErrorFlag</li><li>• kSAI_FIFOErrorFlag</li></ul>

### 29.6.14 static void SAI\_TxEnableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_WordStartInterruptEnable</li><li>• kSAI_SyncErrorInterruptEnable</li><li>• kSAI_FIFOWarningInterruptEnable</li><li>• kSAI_FIFORequestInterruptEnable</li><li>• kSAI_FIFOErrorInterruptEnable</li></ul>

### 29.6.15 static void SAI\_RxEnableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> <li>• kSAI_WordStartInterruptEnable</li> <li>• kSAI_SyncErrorInterruptEnable</li> <li>• kSAI_FIFOWarningInterruptEnable</li> <li>• kSAI_FIFORequestInterruptEnable</li> <li>• kSAI_FIFOErrorInterruptEnable</li> </ul>

**29.6.16 static void SAI\_TxDisableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* )  
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> <li>• kSAI_WordStartInterruptEnable</li> <li>• kSAI_SyncErrorInterruptEnable</li> <li>• kSAI_FIFOWarningInterruptEnable</li> <li>• kSAI_FIFORequestInterruptEnable</li> <li>• kSAI_FIFOErrorInterruptEnable</li> </ul>

## Function Documentation

**29.6.17 static void SAI\_RxDisableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_WordStartInterruptEnable</li><li>• kSAI_SyncErrorInterruptEnable</li><li>• kSAI_FIFOWarningInterruptEnable</li><li>• kSAI_FIFOResponseInterruptEnable</li><li>• kSAI_FIFOErrorInterruptEnable</li></ul>

**29.6.18 static void SAI\_TxEnableDMA ( I2S\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	DMA source The parameter can be combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_FIFOWarningDMAEnable</li><li>• kSAI_FIFOResponseDMAEnable</li></ul>
<i>enable</i>	True means enable DMA, false means disable DMA.

**29.6.19 static void SAI\_RxEnableDMA ( I2S\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	DMA source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_FIFOWarningDMAEnable</li><li>• kSAI_FIFOResponseDMAEnable</li></ul>
<i>enable</i>	True means enable DMA, false means disable DMA.

**29.6.20 static uint32\_t SAI\_TxGetDataRegisterAddress ( I2S\_Type \* *base*, uint32\_t *channel* ) [inline], [static]**

This API is used to provide a transfer address for the SAI DMA transfer configuration.

## Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

### **29.6.21 static uint32\_t SAI\_RxGetDataRegisterAddress ( I2S\_Type \* *base*, uint32\_t *channel* ) [inline], [static]**

This API is used to provide a transfer address for the SAI DMA transfer configuration.

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

### **29.6.22 void SAI\_TxSetFormat ( I2S\_Type \* *base*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )**

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

<i>base</i>	SAI base pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.

<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.
---------------------------	---

### 29.6.23 void SAI\_RxSetFormat ( I2S\_Type \* *base*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

<i>base</i>	SAI base pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.

### 29.6.24 void SAI\_WriteBlocking ( I2S\_Type \* *base*, uint32\_t *channel*, uint32\_t *bitWidth*, uint8\_t \* *buffer*, uint32\_t *size* )

Note

This function blocks by polling until data is ready to be sent.

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be written.
<i>size</i>	Bytes to be written.

### 29.6.25 static void SAI\_WriteData ( I2S\_Type \* *base*, uint32\_t *channel*, uint32\_t *data* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>data</i>	Data needs to be written.

**29.6.26 void SAI\_ReadBlocking ( I2S\_Type \* *base*, uint32\_t *channel*, uint32\_t *bitWidth*, uint8\_t \* *buffer*, uint32\_t *size* )**

Note

This function blocks by polling until data is ready to be sent.

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be read.
<i>size</i>	Bytes to be read.

**29.6.27 static uint32\_t SAI\_ReadData ( I2S\_Type \* *base*, uint32\_t *channel* )  
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.

Returns

Data in SAI FIFO.

**29.6.28 void SAI\_TransferTxCreateHandle ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, sai\_transfer\_callback\_t *callback*, void \* *userData* )**

This function initializes the Tx handle for the SAI Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI handle pointer.
<i>callback</i>	Pointer to the user callback function.
<i>userData</i>	User parameter passed to the callback function

### 29.6.29 void SAI\_TransferRxCreateHandle ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, sai\_transfer\_callback\_t *callback*, void \* *userData* )

This function initializes the Rx handle for the SAI Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>callback</i>	Pointer to the user callback function.
<i>userData</i>	User parameter passed to the callback function.

### 29.6.30 status\_t SAI\_TransferTxSetFormat ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.

## Function Documentation

<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.
---------------------------	---

Returns

Status of this function. Return value is the status\_t.

### 29.6.31 **status\_t SAI\_TransferRxSetFormat ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )**

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is one of status\_t.

### 29.6.32 **status\_t SAI\_TransferSendNonBlocking ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, sai\_transfer\_t \* *xfer* )**

Note

This API returns immediately after the transfer initiates. Call the SAI\_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus\_SAI\_Busy, the transfer is finished.

## Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the <code>sai_handle_t</code> structure which stores the transfer state.
<i>xfer</i>	Pointer to the <code>sai_transfer_t</code> structure.

## Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_TxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

**29.6.33 status\_t SAI\_TransferReceiveNonBlocking ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, sai\_transfer\_t \* *xfer* )**

## Note

This API returns immediately after the transfer initiates. Call the `SAI_RxGetTransferStatusIRQ` to poll the transfer status and check whether the transfer is finished. If the return status is not `kStatus_SAI_Busy`, the transfer is finished.

## Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	Pointer to the <code>sai_handle_t</code> structure which stores the transfer state.
<i>xfer</i>	Pointer to the <code>sai_transfer_t</code> structure.

## Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_RxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

**29.6.34 status\_t SAI\_TransferGetSendCount ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, size\_t \* *count* )**

## Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count sent.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is not a non-blocking transaction currently in progress.

### 29.6.35 **status\_t SAI\_TransferGetReceiveCount ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count received.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is not a non-blocking transaction currently in progress.

### 29.6.36 **void SAI\_TransferAbortSend ( I2S\_Type \* *base*, sai\_handle\_t \* *handle* )**

Note

This API can be called any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.

### 29.6.37 void SAI\_TransferAbortReceive ( I2S\_Type \* *base*, sai\_handle\_t \* *handle* )

Note

This API can be called when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.

### 29.6.38 void SAI\_TransferTxHandleIRQ ( I2S\_Type \* *base*, sai\_handle\_t \* *handle* )

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure.

### 29.6.39 void SAI\_TransferRxHandleIRQ ( I2S\_Type \* *base*, sai\_handle\_t \* *handle* )

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure.

### 29.7 SAI DMA Driver

#### 29.7.1 Overview

#### Data Structures

- struct `sai_dma_handle_t`  
*SAI DMA transfer handle, users should not touch the content of the handle.* [More...](#)

#### TypeDefs

- typedef void(\* `sai_dma_callback_t`)(I2S\_Type \*base, sai\_dma\_handle\_t \*handle, status\_t status, void \*userData)  
*Define SAI DMA callback.*

#### DMA Transactional

- void `SAI_TransferTxCreateHandleDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, `sai_dma_callback_t` callback, void \*userData, dma\_handle\_t \*dmaHandle)  
*Initializes the SAI master DMA handle.*
- void `SAI_TransferRxCreateHandleDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, `sai_dma_callback_t` callback, void \*userData, dma\_handle\_t \*dmaHandle)  
*Initializes the SAI slave DMA handle.*
- void `SAI_TransferTxSetFormatDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, `sai_transfer_format_t` \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
*Configures the SAI Tx audio format.*
- void `SAI_TransferRxSetFormatDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, `sai_transfer_format_t` \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
*Configures the SAI Rx audio format.*
- status\_t `SAI_TransferSendDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, `sai_transfer_t` \*xfer)  
*Performs a non-blocking SAI transfer using DMA.*
- status\_t `SAI_TransferReceiveDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, `sai_transfer_t` \*xfer)  
*Performs a non-blocking SAI transfer using DMA.*
- void `SAI_TransferAbortSendDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle)  
*Aborts a SAI transfer using DMA.*
- void `SAI_TransferAbortReceiveDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle)  
*Aborts a SAI transfer using DMA.*
- status\_t `SAI_TransferGetSendCountDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets byte count sent by SAI.*
- status\_t `SAI_TransferGetReceiveCountDMA` (I2S\_Type \*base, sai\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets byte count received by SAI.*

## 29.7.2 Data Structure Documentation

### 29.7.2.1 struct \_sai\_dma\_handle

#### Data Fields

- `dma_handle_t * dmaHandle`  
*DMA handler for SAI send.*
- `uint8_t bytesPerFrame`  
*Bytes in a frame.*
- `uint8_t channel`  
*Which Data channel SAI use.*
- `uint32_t state`  
*SAI DMA transfer internal state.*
- `sai_dma_callback_t callback`  
*Callback for users while transfer finish or error occurred.*
- `void * userData`  
*User callback parameter.*
- `sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]`  
*Transfer queue storing queued transfer.*
- `size_t transferSize [SAI_XFER_QUEUE_SIZE]`  
*Data bytes need to transfer.*
- `volatile uint8_t queueUser`  
*Index for user to queue transfer.*
- `volatile uint8_t queueDriver`  
*Index for driver to get the transfer data and size.*

#### 29.7.2.1.0.67 Field Documentation

##### 29.7.2.1.0.67.1 `sai_transfer_t sai_dma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

##### 29.7.2.1.0.67.2 `volatile uint8_t sai_dma_handle_t::queueUser`

## 29.7.3 Function Documentation

### 29.7.3.1 `void SAI_TransferTxCreateHandleDMA ( I2S_Type * base, sai_dma_handle_t * handle, sai_dma_callback_t callback, void * userData, dma_handle_t * dmaHandle )`

This function initializes the SAI master DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

Parameters

---

## SAI DMA Driver

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	DMA handle pointer, this handle shall be static allocated by users.

**29.7.3.2 void SAI\_TransferRxCreateHandleDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle*, sai\_dma\_callback\_t *callback*, void \* *userData*, dma\_handle\_t \* *dmaHandle* )**

This function initializes the SAI slave DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	DMA handle pointer, this handle shall be static allocated by users.

**29.7.3.3 void SAI\_TransferTxSetFormatDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )**

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to the format.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.

<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If bit clock source is master. clock, this value should equals to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input arguments is invalid.

#### 29.7.3.4 void SAI\_TransferRxSetFormatDMA ( *I2S\_Type \* base*, *sai\_dma\_handle\_t \* handle*, *sai\_transfer\_format\_t \* format*, *uint32\_t mclkSourceClockHz*, *uint32\_t bclkSourceClockHz* )

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets eDMA parameter according to format.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If bit clock source is master. clock, this value should equals to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input arguments is invalid.

#### 29.7.3.5 status\_t SAI\_TransferSendDMA ( *I2S\_Type \* base*, *sai\_dma\_handle\_t \* handle*, *sai\_transfer\_t \* xfer* )

Note

This interface returns immediately after the transfer initiates. Call the SAI\_GetTransferStatus to poll the transfer status to check whether the SAI transfer finished.

## SAI DMA Driver

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

<i>kStatus_Success</i>	Successfully start the data receive.
<i>kStatus_SAI_TxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

### 29.7.3.6 **status\_t SAI\_TransferReceiveDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle*, sai\_transfer\_t \* *xfer* )**

Note

This interface returns immediately after transfer initiates. Call SAI\_GetTransferStatus to poll the transfer status to check whether the SAI transfer is finished.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI DMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

<i>kStatus_Success</i>	Successfully start the data receive.
<i>kStatus_SAI_RxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

### 29.7.3.7 **void SAI\_TransferAbortSendDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle* )**

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.

### 29.7.3.8 void SAI\_TransferAbortReceiveDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle* )

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.

### 29.7.3.9 status\_t SAI\_TransferGetSendCountDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>count</i>	Bytes count sent by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is not a non-blocking transaction currently in progress.

### 29.7.3.10 status\_t SAI\_TransferGetReceiveCountDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

## SAI DMA Driver

<i>handle</i>	SAI DMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is not a non-blocking transaction currently in progress.

## 29.8 SAI eDMA Driver

### 29.8.1 Overview

#### Data Structures

- struct `sai_edma_handle_t`  
*SAI DMA transfer handle, users should not touch the content of the handle. [More...](#)*

#### TypeDefs

- typedef void(\* `sai_edma_callback_t`)(I2S\_Type \*base, sai\_edma\_handle\_t \*handle, status\_t status, void \*userData)  
*SAI eDMA transfer callback function for finish and error.*

#### eDMA Transactional

- void `SAI_TransferTxCreateHandleEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, `sai_edma_callback_t` callback, void \*userData, `edma_handle_t` \*dmaHandle)  
*Initializes the SAI eDMA handle.*
- void `SAI_TransferRxCreateHandleEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, `sai_edma_callback_t` callback, void \*userData, `edma_handle_t` \*dmaHandle)  
*Initializes the SAI Rx eDMA handle.*
- void `SAI_TransferTxSetFormatEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, `sai_transfer_format_t` \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
*Configures the SAI Tx audio format.*
- void `SAI_TransferRxSetFormatEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, `sai_transfer_format_t` \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
*Configures the SAI Rx audio format.*
- status\_t `SAI_TransferSendEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, `sai_transfer_t` \*xfer)  
*Performs a non-blocking SAI transfer using DMA.*
- status\_t `SAI_TransferReceiveEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, `sai_transfer_t` \*xfer)  
*Performs a non-blocking SAI receive using eDMA.*
- void `SAI_TransferAbortSendEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle)  
*Aborts a SAI transfer using eDMA.*
- void `SAI_TransferAbortReceiveEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle)  
*Aborts a SAI receive using eDMA.*
- status\_t `SAI_TransferGetSendCountEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, size\_t \*count)  
*Gets byte count sent by SAI.*
- status\_t `SAI_TransferGetReceiveCountEDMA` (I2S\_Type \*base, sai\_edma\_handle\_t \*handle, size\_t \*count)  
*Gets byte count received by SAI.*

### 29.8.2 Data Structure Documentation

#### 29.8.2.1 struct \_sai\_edma\_handle

##### Data Fields

- **edma\_handle\_t \* dmaHandle**  
*DMA handler for SAI send.*
- **uint8\_t nbytes**  
*eDMA minor byte transfer count initially configured.*
- **uint8\_t bytesPerFrame**  
*Bytes in a frame.*
- **uint8\_t channel**  
*Which data channel.*
- **uint8\_t count**  
*The transfer data count in a DMA request.*
- **uint32\_t state**  
*Internal state for SAI eDMA transfer.*
- **sai\_edma\_callback\_t callback**  
*Callback for users while transfer finish or error occurs.*
- **void \* userData**  
*User callback parameter.*
- **edma\_tcd\_t tcd [SAI\_XFER\_QUEUE\_SIZE+1U]**  
*TCD pool for eDMA transfer.*
- **sai\_transfer\_t saiQueue [SAI\_XFER\_QUEUE\_SIZE]**  
*Transfer queue storing queued transfer.*
- **size\_t transferSize [SAI\_XFER\_QUEUE\_SIZE]**  
*Data bytes need to transfer.*
- **volatile uint8\_t queueUser**  
*Index for user to queue transfer.*
- **volatile uint8\_t queueDriver**  
*Index for driver to get the transfer data and size.*

### 29.8.2.1.0.68 Field Documentation

29.8.2.1.0.68.1 `uint8_t sai_edma_handle_t::nbytes`

29.8.2.1.0.68.2 `edma_tcd_t sai_edma_handle_t::tcd[SAI_XFER_QUEUE_SIZE+1U]`

29.8.2.1.0.68.3 `sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

29.8.2.1.0.68.4 `volatile uint8_t sai_edma_handle_t::queueUser`

### 29.8.3 Function Documentation

29.8.3.1 `void SAI_TransferTxCreateHandleEDMA ( I2S_Type * base, sai_edma_handle_t * handle, sai_edma_callback_t callback, void * userData, edma_handle_t * dmaHandle )`

This function initializes the SAI master DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

## SAI eDMA Driver

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	eDMA handle pointer, this handle shall be static allocated by users.

**29.8.3.2 void SAI\_TransferRxCreateHandleEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_edma\_callback\_t *callback*, void \* *userData*, edma\_handle\_t \* *dmaHandle* )**

This function initializes the SAI slave DMA handle, which can be used for other SAI master transactional APIs. Usually, for a specified SAI instance, call this API once to get the initialized handle.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	eDMA handle pointer, this handle shall be static allocated by users.

**29.8.3.3 void SAI\_TransferTxSetFormatEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )**

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

<i>handle</i>	SAI eDMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSourceClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSourceClockHz</i>	SAI bit clock source frequency in Hz. If bit clock source is master clock, this value should equals to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

**29.8.3.4 void SAI\_TransferRxSetFormatEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )**

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSourceClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSourceClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is the master clock, this value should equal to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

**29.8.3.5 status\_t SAI\_TransferSendEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_transfer\_t \* *xfer* )**

## SAI eDMA Driver

### Note

This interface returns immediately after the transfer initiates. Call SAI\_GetTransferStatus to poll the transfer status and check whether the SAI transfer is finished.

### Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to the DMA transfer structure.

### Return values

<i>kStatus_Success</i>	Start a SAI eDMA send successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_TxBusy</i>	SAI is busy sending data.

### 29.8.3.6 **status\_t SAI\_TransferReceiveEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_transfer\_t \* *xfer* )**

### Note

This interface returns immediately after the transfer initiates. Call the SAI\_GetReceiveRemainingBytes to poll the transfer status and check whether the SAI transfer is finished.

### Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

### Return values

<i>kStatus_Success</i>	Start a SAI eDMA receive successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_RxBusy</i>	SAI is busy receiving data.

### 29.8.3.7 **void SAI\_TransferAbortSendEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle* )**

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

### 29.8.3.8 void SAI\_TransferAbortReceiveEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle* )

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.

### 29.8.3.9 status\_t SAI\_TransferGetSendCountEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count sent by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is no non-blocking transaction in progress.

### 29.8.3.10 status\_t SAI\_TransferGetReceiveCountEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

## SAI eDMA Driver

<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is no non-blocking transaction in progress.

# Chapter 30

## SDHC: Secure Digital Host Controller Driver

### 30.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Secure Digital Host Controller (SDHC) module of MCUXpresso SDK devices.

### 30.2 Typical use case

#### 30.2.1 SDHC Operation

```
/* Initializes the SDHC. */
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Fills state in the card driver. */
card->sdhcBase = BOARD_SDHC_BASEADDR;
card->sdhcSourceClock = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->sdhcTransfer = sdhc_transfer_function;

/* Initializes the card. */
if (SD_Init(card))
{
    PRINTF("\r\nSD card init failed.\r\n");
}

PRINTF("\r\nRead/Write/Erase the card continuously until it encounters error.....\r\n");
while (true)
{
    if (kStatus_Success != SD_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
                                             DATA_BLOCK_COUNT))
    {
        PRINTF("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != SD_ReadBlocks(card, g_dataRead, DATA_BLOCK_START, DATA_BLOCK_COUNT))
    {
        PRINTF("Read multiple data blocks failed.\r\n");
    }

    if (kStatus_Success != SD_EraseBlocks(card, DATA_BLOCK_START, DATA_BLOCK_COUNT))
    {
        PRINTF("Erase multiple data blocks failed.\r\n");
    }
}

SD_Deinit(card);
```

### Data Structures

- struct `sdhc_adma2_descriptor_t`

## Typical use case

- struct `sdhc_capability_t`  
*SDHC capability information.* [More...](#)
- struct `sdhc_transfer_config_t`  
*Card transfer configuration.* [More...](#)
- struct `sdhc_boot_config_t`  
*Data structure to configure the MMC boot feature.* [More...](#)
- struct `sdhc_config_t`  
*Data structure to initialize the SDHC.* [More...](#)
- struct `sdhc_data_t`  
*Card data descriptor.* [More...](#)
- struct `sdhc_command_t`  
*Card command descriptor.* [More...](#)
- struct `sdhc_transfer_t`  
*Transfer state.* [More...](#)
- struct `sdhc_transfer_callback_t`  
*SDHC callback functions.* [More...](#)
- struct `sdhc_handle_t`  
*SDHC handle.* [More...](#)
- struct `sdhc_host_t`  
*SDHC host descriptor.* [More...](#)

## Macros

- #define `SDHC_MAX_BLOCK_COUNT` (`SDHC_BLKATTR_BLKCNT_MASK >> SDHC_BLKATTR_BLKCNT_SHIFT`)  
*Maximum block count can be set one time.*
- #define `SDHC_ADMA1_ADDRESS_ALIGN` (4096U)  
*The alignment size for ADDRESS filed in ADMA1's descriptor.*
- #define `SDHC_ADMA1_LENGTH_ALIGN` (4096U)  
*The alignment size for LENGTH field in ADMA1's descriptor.*
- #define `SDHC_ADMA2_ADDRESS_ALIGN` (4U)  
*The alignment size for ADDRESS field in ADMA2's descriptor.*
- #define `SDHC_ADMA2_LENGTH_ALIGN` (4U)  
*The alignment size for LENGTH filed in ADMA2's descriptor.*
- #define `SDHC_ADMA1_DESCRIPTOR_ADDRESS_SHIFT` (12U)  
*The bit shift for ADDRESS filed in ADMA1's descriptor.*
- #define `SDHC_ADMA1_DESCRIPTOR_ADDRESS_MASK` (0xFFFFFU)  
*The bit mask for ADDRESS field in ADMA1's descriptor.*
- #define `SDHC_ADMA1_DESCRIPTOR_LENGTH_SHIFT` (12U)  
*The bit shift for LENGTH filed in ADMA1's descriptor.*
- #define `SDHC_ADMA1_DESCRIPTOR_LENGTH_MASK` (0xFFFFU)  
*The mask for LENGTH field in ADMA1's descriptor.*
- #define `SDHC_ADMA1_DESCRIPTOR_MAX_LENGTH_PER_ENTRY` (`SDHC_ADMA1_DESCRIPTOR_LENGTH_MASK + 1U`)  
*The maximum value of LENGTH filed in ADMA1's descriptor.*
- #define `SDHC_ADMA2_DESCRIPTOR_LENGTH_SHIFT` (16U)  
*The bit shift for LENGTH field in ADMA2's descriptor.*
- #define `SDHC_ADMA2_DESCRIPTOR_LENGTH_MASK` (0xFFFFU)  
*The bit mask for LENGTH field in ADMA2's descriptor.*
- #define `SDHC_ADMA2_DESCRIPTOR_MAX_LENGTH_PER_ENTRY` (`SDHC_ADMA2_DE-`

**SCRIPTOR\_LENGTH\_MASK)**

*The maximum value of LENGTH field in ADMA2's descriptor.*

**Typedefs**

- **typedef uint32\_t sdhc\_adma1\_descriptor\_t**  
*Defines the adma1 descriptor structure.*
- **typedef status\_t(\* sdhc\_transfer\_function\_t )(SDHC\_Type \*base, sdhc\_transfer\_t \*content)**  
*SDHC transfer function.*

**Enumerations**

- **enum \_sdhc\_status {**  
 kStatus\_SDHC\_BusyTransferring = MAKE\_STATUS(kStatusGroup\_SDHC, 0U),  
 kStatus\_SDHC\_PreparesAdmaDescriptorFailed = MAKE\_STATUS(kStatusGroup\_SDHC, 1U),  
 kStatus\_SDHC\_SendCommandFailed = MAKE\_STATUS(kStatusGroup\_SDHC, 2U),  
 kStatus\_SDHC\_TransferDataFailed = MAKE\_STATUS(kStatusGroup\_SDHC, 3U),  
 kStatus\_SDHC\_DMADataBufferAddrNotAlign }  
*SDHC status.*
- **enum \_sdhc\_capability\_flag {**  
 kSDHC\_SupportAdmaFlag = SDHC\_HTCAPBLT\_ADMAS\_MASK,  
 kSDHC\_SupportHighSpeedFlag = SDHC\_HTCAPBLT\_HSS\_MASK,  
 kSDHC\_SupportDmaFlag = SDHC\_HTCAPBLT\_DMAS\_MASK,  
 kSDHC\_SupportSuspendResumeFlag = SDHC\_HTCAPBLT\_SRS\_MASK,  
 kSDHC\_SupportV330Flag = SDHC\_HTCAPBLT\_VS33\_MASK,  
 kSDHC\_Support4BitFlag = (SDHC\_HTCAPBLT\_MBL\_SHIFT << 0U),  
 kSDHC\_Support8BitFlag = (SDHC\_HTCAPBLT\_MBL\_SHIFT << 1U) }  
*Host controller capabilities flag mask.*
- **enum \_sdhc\_wakeup\_event {**  
 kSDHC\_WakeupEventOnCardInt = SDHC\_PROCTL\_WECINT\_MASK,  
 kSDHC\_WakeupEventOnCardInsert = SDHC\_PROCTL\_WECINS\_MASK,  
 kSDHC\_WakeupEventOnCardRemove = SDHC\_PROCTL\_WECRM\_MASK,  
 kSDHC\_WakeupEventsAll }  
*Wakeup event mask.*
- **enum \_sdhc\_reset {**  
 kSDHC\_ResetAll = SDHC\_SYSCTL\_RSTA\_MASK,  
 kSDHC\_ResetCommand = SDHC\_SYSCTL\_RSTC\_MASK,  
 kSDHC\_ResetData = SDHC\_SYSCTL\_RSTD\_MASK,  
 kSDHC\_ResetsAll = (kSDHC\_ResetAll | kSDHC\_ResetCommand | kSDHC\_ResetData) }  
*Reset type mask.*
- **enum \_sdhc\_transfer\_flag {**

## Typical use case

```
kSDHC_EnableDmaFlag = SDHC_XFERTYP_DMAEN_MASK,  
kSDHC_CommandTypeSuspendFlag = (SDHC_XFERTYP_CMDTYP(1U)),  
kSDHC_CommandTypeResumeFlag = (SDHC_XFERTYP_CMDTYP(2U)),  
kSDHC_CommandTypeAbortFlag = (SDHC_XFERTYP_CMDTYP(3U)),  
kSDHC_EnableBlockCountFlag = SDHC_XFERTYP_BCEN_MASK,  
kSDHC_EnableAutoCommand12Flag = SDHC_XFERTYP_AC12EN_MASK,  
kSDHC_DataReadFlag = SDHC_XFERTYP_DTDSEL_MASK,  
kSDHC_MultipleBlockFlag = SDHC_XFERTYP_MSBSEL_MASK,  
kSDHC_ResponseLength136Flag = SDHC_XFERTYP_RSPTYP(1U),  
kSDHC_ResponseLength48Flag = SDHC_XFERTYP_RSPTYP(2U),  
kSDHC_ResponseLength48BusyFlag = SDHC_XFERTYP_RSPTYP(3U),  
kSDHC_EnableCrcCheckFlag = SDHC_XFERTYP_CCCEN_MASK,  
kSDHC_EnableIndexCheckFlag = SDHC_XFERTYP_CICEN_MASK,  
kSDHC_DataPresentFlag = SDHC_XFERTYP_DPSEL_MASK }
```

*Transfer flag mask.*

- enum \_sdhc\_present\_status\_flag {  
    kSDHC\_CommandInhibitFlag = SDHC\_PRSSTAT\_CIHB\_MASK,  
    kSDHC\_DataInhibitFlag = SDHC\_PRSSTAT\_CDIHB\_MASK,  
    kSDHC\_DataLineActiveFlag = SDHC\_PRSSTAT\_DLA\_MASK,  
    kSDHC\_SdClockStableFlag = SDHC\_PRSSTAT\_SDSTB\_MASK,  
    kSDHC\_WriteTransferActiveFlag = SDHC\_PRSSTAT\_WTA\_MASK,  
    kSDHC\_ReadTransferActiveFlag = SDHC\_PRSSTAT\_RTA\_MASK,  
    kSDHC\_BufferWriteEnableFlag = SDHC\_PRSSTAT\_BWEN\_MASK,  
    kSDHC\_BufferReadEnableFlag = SDHC\_PRSSTAT\_BREN\_MASK,  
    kSDHC\_CardInsertedFlag = SDHC\_PRSSTAT\_CINS\_MASK,  
    kSDHC\_CommandLineLevelFlag = SDHC\_PRSSTAT\_CLSL\_MASK,  
    kSDHC\_Data0LineLevelFlag = (1U << 24U),  
    kSDHC\_Data1LineLevelFlag = (1U << 25U),  
    kSDHC\_Data2LineLevelFlag = (1U << 26U),  
    kSDHC\_Data3LineLevelFlag = (1U << 27U),  
    kSDHC\_Data4LineLevelFlag = (1U << 28U),  
    kSDHC\_Data5LineLevelFlag = (1U << 29U),  
    kSDHC\_Data6LineLevelFlag = (1U << 30U),  
    kSDHC\_Data7LineLevelFlag = (1U << 31U) }

*Present status flag mask.*

- enum \_sdhc\_interrupt\_status\_flag {

```

kSDHC_CommandCompleteFlag = SDHC_IRQSTAT_CC_MASK,
kSDHC_DataCompleteFlag = SDHC_IRQSTAT_TC_MASK,
kSDHC_BlockGapEventFlag = SDHC_IRQSTAT_BGE_MASK,
kSDHC_DmaCompleteFlag = SDHC_IRQSTAT_DINT_MASK,
kSDHC_BufferWriteReadyFlag = SDHC_IRQSTAT_BWR_MASK,
kSDHC_BufferReadReadyFlag = SDHC_IRQSTAT_BRR_MASK,
kSDHC_CardInsertionFlag = SDHC_IRQSTAT_CINS_MASK,
kSDHC_CardRemovalFlag = SDHC_IRQSTAT_CRM_MASK,
kSDHC_CardInterruptFlag = SDHC_IRQSTAT_CINT_MASK,
kSDHC_CommandTimeoutFlag = SDHC_IRQSTAT_CTOE_MASK,
kSDHC_CommandCrcErrorFlag = SDHC_IRQSTAT_CCE_MASK,
kSDHC_CommandEndBitErrorFlag = SDHC_IRQSTAT_CEBE_MASK,
kSDHC_CommandIndexErrorFlag = SDHC_IRQSTAT_CIE_MASK,
kSDHC_DataTimeoutFlag = SDHC_IRQSTAT_DTOE_MASK,
kSDHC_DataCrcErrorFlag = SDHC_IRQSTAT_DCE_MASK,
kSDHC_DataEndBitErrorFlag = SDHC_IRQSTAT_DEBE_MASK,
kSDHC_AutoCommand12ErrorFlag = SDHC_IRQSTAT_AC12E_MASK,
kSDHC_DmaErrorFlag = SDHC_IRQSTAT_DMAE_MASK,
kSDHC_CommandErrorFlag,
kSDHC_DataErrorFlag,
kSDHC_ErrorFlag = (kSDHC_CommandErrorFlag | kSDHC_DataErrorFlag | kSDHC_DmaErrorFlag),
kSDHC_DataFlag,
kSDHC_CommandFlag = (kSDHC_CommandErrorFlag | kSDHC_CommandCompleteFlag),
kSDHC_CardDetectFlag = (kSDHC_CardInsertionFlag | kSDHC_CardRemovalFlag),
kSDHC_AllInterruptFlags }

```

*Interrupt status flag mask.*

- enum `_sdhc_auto_command12_error_status_flag` {
 kSDHC\_AutoCommand12NotExecutedFlag = SDHC\_AC12ERR\_AC12NE\_MASK,
 kSDHC\_AutoCommand12TimeoutFlag = SDHC\_AC12ERR\_AC12TOE\_MASK,
 kSDHC\_AutoCommand12EndBitErrorFlag = SDHC\_AC12ERR\_AC12EBE\_MASK,
 kSDHC\_AutoCommand12CrcErrorFlag = SDHC\_AC12ERR\_AC12CE\_MASK,
 kSDHC\_AutoCommand12IndexErrorFlag = SDHC\_AC12ERR\_AC12IE\_MASK,
 kSDHC\_AutoCommand12NotIssuedFlag = SDHC\_AC12ERR\_CNIBAC12E\_MASK }

*Auto CMD12 error status flag mask.*

- enum `_sdhc_adma_error_status_flag` {
 kSDHC\_AdmaLengthMismatchFlag = SDHC\_ADMAESADMALME\_MASK,
 kSDHC\_AdmaDescriptorErrorFlag = SDHC\_ADMAESADMADCE\_MASK }

*ADMA error status flag mask.*

- enum `sdhc_adma_error_state_t` {
 kSDHC\_AdmaErrorStateStopDma = 0x00U,
 kSDHC\_AdmaErrorStateFetchDescriptor = 0x01U,
 kSDHC\_AdmaErrorStateChangeAddress = 0x02U,
 kSDHC\_AdmaErrorStateTransferData = 0x03U }

*ADMA error state.*

- enum `_sdhc_force_event` {

## Typical use case

```
kSDHC_ForceEventAutoCommand12NotExecuted = SDHC_FEVT_AC12NE_MASK,  
kSDHC_ForceEventAutoCommand12Timeout = SDHC_FEVT_AC12TOE_MASK,  
kSDHC_ForceEventAutoCommand12CrcError = SDHC_FEVT_AC12CE_MASK,  
kSDHC_ForceEventEndBitError = SDHC_FEVT_AC12EBE_MASK,  
kSDHC_ForceEventAutoCommand12IndexError = SDHC_FEVT_AC12IE_MASK,  
kSDHC_ForceEventAutoCommand12NotIssued = SDHC_FEVT_CNIBAC12E_MASK,  
kSDHC_ForceEventCommandTimeout = SDHC_FEVT_CTOE_MASK,  
kSDHC_ForceEventCommandCrcError = SDHC_FEVT_CCE_MASK,  
kSDHC_ForceEventCommandEndBitError = SDHC_FEVT_CEBE_MASK,  
kSDHC_ForceEventCommandIndexError = SDHC_FEVT_CIE_MASK,  
kSDHC_ForceEventDataTimeout = SDHC_FEVT_DTOE_MASK,  
kSDHC_ForceEventDataCrcError = SDHC_FEVT_DCE_MASK,  
kSDHC_ForceEventDataEndBitError = SDHC_FEVT_DEBE_MASK,  
kSDHC_ForceEventAutoCommand12Error = SDHC_FEVT_AC12E_MASK,  
kSDHC_ForceEventCardInt = SDHC_FEVT_CINT_MASK,  
kSDHC_ForceEventDmaError = SDHC_FEVT_DMAE_MASK,  
kSDHC_ForceEventsAll }
```

*Force event mask.*

- enum `sdhc_data_bus_width_t` {  
    kSDHC\_DataBusWidth1Bit = 0U,  
    kSDHC\_DataBusWidth4Bit = 1U,  
    kSDHC\_DataBusWidth8Bit = 2U }

*Data transfer width.*

- enum `sdhc_endian_mode_t` {  
    kSDHC\_EndianModeBig = 0U,  
    kSDHC\_EndianModeHalfWordBig = 1U,  
    kSDHC\_EndianModeLittle = 2U }

*Endian mode.*

- enum `sdhc_dma_mode_t` {  
    kSDHC\_DmaModeNo = 0U,  
    kSDHC\_DmaModeAdma1 = 1U,  
    kSDHC\_DmaModeAdma2 = 2U }

*DMA mode.*

- enum `_sdhc_sdio_control_flag` {  
    kSDHC\_StopAtBlockGapFlag = 0x01,  
    kSDHC\_ReadWaitControlFlag = 0x02,  
    kSDHC\_InterruptAtBlockGapFlag = 0x04,  
    kSDHC\_ExactBlockNumberReadFlag = 0x08 }

*SDIO control flag mask.*

- enum `sdhc_boot_mode_t` {  
    kSDHC\_BootModeNormal = 0U,  
    kSDHC\_BootModeAlternative = 1U }

*MMC card boot mode.*

- enum `sdhc_card_command_type_t` {

```
kCARD_CommandTypeNormal = 0U,
kCARD_CommandTypeSuspend = 1U,
kCARD_CommandTypeResume = 2U,
kCARD_CommandTypeAbort = 3U }
```

*The command type.*

- enum `sdhc_card_response_type_t` {
 kCARD\_ResponseNone = 0U,
 kCARD\_ResponseTypeR1 = 1U,
 kCARD\_ResponseTypeR1b = 2U,
 kCARD\_ResponseTypeR2 = 3U,
 kCARD\_ResponseTypeR3 = 4U,
 kCARD\_ResponseTypeR4 = 5U,
 kCARD\_ResponseTypeR5 = 6U,
 kCARD\_ResponseTypeR5b = 7U,
 kCARD\_ResponseTypeR6 = 8U,
 kCARD\_ResponseTypeR7 = 9U }

*The command response type.*

- enum `_sdhc_adma1_descriptor_flag` {
 kSDHC\_Adma1DescriptorValidFlag = (1U << 0U),
 kSDHC\_Adma1DescriptorEndFlag = (1U << 1U),
 kSDHC\_Adma1DescriptorInterrupFlag = (1U << 2U),
 kSDHC\_Adma1DescriptorActivity1Flag = (1U << 4U),
 kSDHC\_Adma1DescriptorActivity2Flag = (1U << 5U),
 kSDHC\_Adma1DescriptorTypeNop = (kSDHC\_Adma1DescriptorValidFlag),
 kSDHC\_Adma1DescriptorTypeTransfer,
 kSDHC\_Adma1DescriptorTypeLink,
 kSDHC\_Adma1DescriptorTypeSetLength }

*The mask for the control/status field in ADMA1 descriptor.*

- enum `_sdhc_adma2_descriptor_flag` {
 kSDHC\_Adma2DescriptorValidFlag = (1U << 0U),
 kSDHC\_Adma2DescriptorEndFlag = (1U << 1U),
 kSDHC\_Adma2DescriptorInterruptFlag = (1U << 2U),
 kSDHC\_Adma2DescriptorActivity1Flag = (1U << 4U),
 kSDHC\_Adma2DescriptorActivity2Flag = (1U << 5U),
 kSDHC\_Adma2DescriptorTypeNop = (kSDHC\_Adma2DescriptorValidFlag),
 kSDHC\_Adma2DescriptorTypeReserved,
 kSDHC\_Adma2DescriptorTypeTransfer,
 kSDHC\_Adma2DescriptorTypeLink }

*ADMA1 descriptor control and status mask.*

## Driver version

- #define `FSL_SDHC_DRIVER_VERSION` (MAKE\_VERSION(2U, 1U, 5U))  
*Driver version 2.1.5.*

## Typical use case

### Initialization and deinitialization

- void **SDHC\_Init** (SDHC\_Type \*base, const **sdhc\_config\_t** \*config)  
*SDHC module initialization function.*
- void **SDHC\_Deinit** (SDHC\_Type \*base)  
*Deinitializes the SDHC.*
- bool **SDHC\_Reset** (SDHC\_Type \*base, uint32\_t mask, uint32\_t timeout)  
*Resets the SDHC.*

### DMA Control

- status\_t **SDHC\_SetAdmaTableConfig** (SDHC\_Type \*base, **sdhc\_dma\_mode\_t** dmaMode, uint32\_t \*table, uint32\_t tableWords, const uint32\_t \*data, uint32\_t dataBytes)  
*Sets the ADMA descriptor table configuration.*

### Interrupts

- static void **SDHC\_EnableInterruptStatus** (SDHC\_Type \*base, uint32\_t mask)  
*Enables the interrupt status.*
- static void **SDHC\_DisableInterruptStatus** (SDHC\_Type \*base, uint32\_t mask)  
*Disables the interrupt status.*
- static void **SDHC\_EnableInterruptSignal** (SDHC\_Type \*base, uint32\_t mask)  
*Enables the interrupt signal corresponding to the interrupt status flag.*
- static void **SDHC\_DisableInterruptSignal** (SDHC\_Type \*base, uint32\_t mask)  
*Disables the interrupt signal corresponding to the interrupt status flag.*

### Status

- static uint32\_t **SDHC\_GetInterruptStatusFlags** (SDHC\_Type \*base)  
*Gets the current interrupt status.*
- static void **SDHC\_ClearInterruptStatusFlags** (SDHC\_Type \*base, uint32\_t mask)  
*Clears a specified interrupt status.*
- static uint32\_t **SDHC\_GetAutoCommand12ErrorStatusFlags** (SDHC\_Type \*base)  
*Gets the status of auto command 12 error.*
- static uint32\_t **SDHC\_GetAdmaErrorStatusFlags** (SDHC\_Type \*base)  
*Gets the status of the ADMA error.*
- static uint32\_t **SDHC\_GetPresentStatusFlags** (SDHC\_Type \*base)  
*Gets a present status.*

### Bus Operations

- void **SDHC\_GetCapability** (SDHC\_Type \*base, **sdhc\_capability\_t** \*capability)  
*Gets the capability information.*
- static void **SDHC\_EnableSdClock** (SDHC\_Type \*base, bool enable)  
*Enables or disables the SD bus clock.*
- uint32\_t **SDHC\_SetSdClock** (SDHC\_Type \*base, uint32\_t srcClock\_Hz, uint32\_t busClock\_Hz)  
*Sets the SD bus clock frequency.*
- bool **SDHC\_SetCardActive** (SDHC\_Type \*base, uint32\_t timeout)  
*Sends 80 clocks to the card to set it to the active state.*
- static void **SDHC\_SetDataBusWidth** (SDHC\_Type \*base, **sdhc\_data\_bus\_width\_t** width)  
*Sets the data transfer width.*

- void **SDHC\_SetTransferConfig** (SDHC\_Type \*base, const **sdhc\_transfer\_config\_t** \*config)  
*Sets the card transfer-related configuration.*
- static uint32\_t **SDHC\_GetCommandResponse** (SDHC\_Type \*base, uint32\_t index)  
*Gets the command response.*
- static void **SDHC\_WriteData** (SDHC\_Type \*base, uint32\_t data)  
*Fills the the data port.*
- static uint32\_t **SDHC\_ReadData** (SDHC\_Type \*base)  
*Retrieves the data from the data port.*
- static void **SDHC\_EnableWakeupEvent** (SDHC\_Type \*base, uint32\_t mask, bool enable)  
*Enables or disables a wakeup event in low-power mode.*
- static void **SDHC\_EnableCardDetecfTest** (SDHC\_Type \*base, bool enable)  
*Enables or disables the card detection level for testing.*
- static void **SDHC\_SetCardDetectTestLevel** (SDHC\_Type \*base, bool high)  
*Sets the card detection test level.*
- void **SDHC\_EnableSdioControl** (SDHC\_Type \*base, uint32\_t mask, bool enable)  
*Enables or disables the SDIO card control.*
- static void **SDHC\_SetContinueRequest** (SDHC\_Type \*base)  
*Restarts a transaction which has stopped at the block GAP for the SDIO card.*
- void **SDHC\_SetMmcBootConfig** (SDHC\_Type \*base, const **sdhc\_boot\_config\_t** \*config)  
*Configures the MMC boot feature.*
- static void **SDHC\_SetForceEvent** (SDHC\_Type \*base, uint32\_t mask)  
*Forces generating events according to the given mask.*

## Transactional

- status\_t **SDHC\_TransferBlocking** (SDHC\_Type \*base, uint32\_t \*admaTable, uint32\_t admaTableWords, **sdhc\_transfer\_t** \*transfer)  
*Transfers the command/data using a blocking method.*
- void **SDHC\_TransferCreateHandle** (SDHC\_Type \*base, sdhc\_handle\_t \*handle, const **sdhc\_transfer\_callback\_t** \*callback, void \*userData)  
*Creates the SDHC handle.*
- status\_t **SDHC\_TransferNonBlocking** (SDHC\_Type \*base, sdhc\_handle\_t \*handle, uint32\_t \*admaTable, uint32\_t admaTableWords, **sdhc\_transfer\_t** \*transfer)  
*Transfers the command/data using an interrupt and an asynchronous method.*
- void **SDHC\_TransferHandleIRQ** (SDHC\_Type \*base, sdhc\_handle\_t \*handle)  
*IRQ handler for the SDHC.*

## 30.3 Data Structure Documentation

### 30.3.1 struct **sdhc\_adma2\_descriptor\_t**

#### Data Fields

- uint32\_t **attribute**  
*The control and status field.*
- const uint32\_t \* **address**  
*The address field.*

## Data Structure Documentation

### 30.3.2 struct sdhc\_capability\_t

Defines a structure to save the capability information of SDHC.

#### Data Fields

- uint32\_t **specVersion**  
*Specification version.*
- uint32\_t **vendorVersion**  
*Vendor version.*
- uint32\_t **maxBlockLength**  
*Maximum block length unitized as byte.*
- uint32\_t **maxBlockCount**  
*Maximum block count can be set one time.*
- uint32\_t **flags**  
*Capability flags to indicate the support information(\_sdhc\_capability\_flag)*

### 30.3.3 struct sdhc\_transfer\_config\_t

Define structure to configure the transfer-related command index/argument/flags and data block size/data block numbers. This structure needs to be filled each time a command is sent to the card.

#### Data Fields

- size\_t **dataBlockSize**  
*Data block size.*
- uint32\_t **dataBlockCount**  
*Data block count.*
- uint32\_t **commandArgument**  
*Command argument.*
- uint32\_t **commandIndex**  
*Command index.*
- uint32\_t **flags**  
*Transfer flags(\_sdhc\_transfer\_flag)*

### 30.3.4 struct sdhc\_boot\_config\_t

#### Data Fields

- uint32\_t **ackTimeoutCount**  
*Timeout value for the boot ACK.*
- **sdhc\_boot\_mode\_t bootMode**  
*Boot mode selection.*
- uint32\_t **blockCount**

- Stop at block gap value of automatic mode.
- bool [enableBootAck](#)  
Enable or disable boot ACK.
- bool [enableBoot](#)  
Enable or disable fast boot.
- bool [enableAutoStopAtBlockGap](#)  
Enable or disable auto stop at block gap function in boot period.

### 30.3.4.0.0.69 Field Documentation

#### 30.3.4.0.0.69.1 uint32\_t [sdhc\\_boot\\_config\\_t::ackTimeoutCount](#)

The available range is 0 ~ 15.

#### 30.3.4.0.0.69.2 [sdhc\\_boot\\_mode\\_t sdhc\\_boot\\_config\\_t::bootMode](#)

#### 30.3.4.0.0.69.3 uint32\_t [sdhc\\_boot\\_config\\_t::blockCount](#)

Available range is 0 ~ 65535.

## 30.3.5 struct [sdhc\\_config\\_t](#)

### Data Fields

- bool [cardDetectDat3](#)  
Enable DAT3 as card detection pin.
- [sdhc\\_endian\\_mode\\_t endianMode](#)  
Endian mode.
- [sdhc\\_dma\\_mode\\_t dmaMode](#)  
DMA mode.
- uint32\_t [readWatermarkLevel](#)  
Watermark level for DMA read operation.
- uint32\_t [writeWatermarkLevel](#)  
Watermark level for DMA write operation.

### 30.3.5.0.0.70 Field Documentation

#### 30.3.5.0.0.70.1 uint32\_t [sdhc\\_config\\_t::readWatermarkLevel](#)

Available range is 1 ~ 128.

#### 30.3.5.0.0.70.2 uint32\_t [sdhc\\_config\\_t::writeWatermarkLevel](#)

Available range is 1 ~ 128.

### 30.3.6 struct sdhc\_data\_t

Defines a structure to contain data-related attribute. 'enableIgnoreError' is used for the case that upper card driver want to ignore the error event to read/write all the data not to stop read/write immediately when error event happen for example bus testing procedure for MMC card.

#### Data Fields

- bool `enableAutoCommand12`  
*Enable auto CMD12.*
- bool `enableIgnoreError`  
*Enable to ignore error event to read/write all the data.*
- `size_t blockSize`  
*Block size.*
- `uint32_t blockCount`  
*Block count.*
- `uint32_t * rxData`  
*Buffer to save data read.*
- `const uint32_t * txData`  
*Data buffer to write.*

### 30.3.7 struct sdhc\_command\_t

Define card command-related attribute.

#### Data Fields

- `uint32_t index`  
*Command index.*
- `uint32_t argument`  
*Command argument.*
- `sdhc_card_command_type_t type`  
*Command type.*
- `sdhc_card_response_type_t responseType`  
*Command response type.*
- `uint32_t response [4U]`  
*Response for this command.*
- `uint32_t responseErrorFlags`  
*response error flag, the flag which need to check the command response*

### 30.3.8 struct sdhc\_transfer\_t

#### Data Fields

- `sdhc_data_t * data`  
*Data to transfer.*
- `sdhc_command_t * command`  
*Command to send.*

### 30.3.9 struct sdhc\_transfer\_callback\_t

#### Data Fields

- `void(* CardInserted )(void)`  
*Card inserted occurs when DAT3/CD pin is for card detect.*
- `void(* CardRemoved )(void)`  
*Card removed occurs.*
- `void(* SdioInterrupt )(void)`  
*SDIO card interrupt occurs.*
- `void(* SdioBlockGap )(void)`  
*SDIO card stopped at block gap occurs.*
- `void(* TransferComplete )(SDHC_Type *base, sdhc_handle_t *handle, status_t status, void *userData)`  
*Transfer complete callback.*

### 30.3.10 struct \_sdhc\_handle

SDHC handle typedef.

Defines the structure to save the SDHC state information and callback function. The detailed interrupt status when sending a command or transferring data can be obtained from the interruptFlags field by using the mask defined in `sdhc_interrupt_flag_t`.

Note

All the fields except interruptFlags and transferredWords must be allocated by the user.

#### Data Fields

- `sdhc_data_t *volatile data`  
*Data to transfer.*
- `sdhc_command_t *volatile command`  
*Command to send.*
- `volatile uint32_t interruptFlags`  
*Interrupt flags of last transaction.*

## Enumeration Type Documentation

- volatile uint32\_t **transferredWords**  
*Words transferred by DATAPORT way.*
- **sdhc\_transfer\_callback\_t callback**  
*Callback function.*
- void \* **userData**  
*Parameter for transfer complete callback.*

### 30.3.11 struct sdhc\_host\_t

#### Data Fields

- SDHC\_Type \* **base**  
*SDHC peripheral base address.*
- uint32\_t **sourceClock\_Hz**  
*SDHC source clock frequency united in Hz.*
- **sdhc\_config\_t config**  
*SDHC configuration.*
- **sdhc\_capability\_t capability**  
*SDHC capability information.*
- **sdhc\_transfer\_function\_t transfer**  
*SDHC transfer function.*

## 30.4 Macro Definition Documentation

### 30.4.1 #define FSL\_SDHC\_DRIVER\_VERSION (MAKE\_VERSION(2U, 1U, 5U))

## 30.5 Typedef Documentation

### 30.5.1 typedef uint32\_t sdhc\_adma1\_descriptor\_t

### 30.5.2 typedef status\_t(\* sdhc\_transfer\_function\_t)(SDHC\_Type \*base, sdhc\_transfer\_t \*content)

## 30.6 Enumeration Type Documentation

### 30.6.1 enum \_sdhc\_status

Enumerator

**kStatus\_SDHC\_BusyTransferring** Transfer is on-going.

**kStatus\_SDHC\_PrepardAdmaDescriptorFailed** Set DMA descriptor failed.

**kStatus\_SDHC\_SendCommandFailed** Send command failed.

**kStatus\_SDHC\_TransferDataFailed** Transfer data failed.

**kStatus\_SDHC\_DMADataBufferAddrNotAlign** data buffer addr not align in DMA mode

### 30.6.2 enum \_sdhc\_capability\_flag

Enumerator

- kSDHC\_SupportAdmaFlag* Support ADMA.
- kSDHC\_SupportHighSpeedFlag* Support high-speed.
- kSDHC\_SupportDmaFlag* Support DMA.
- kSDHC\_SupportSuspendResumeFlag* Support suspend/resume.
- kSDHC\_SupportV330Flag* Support voltage 3.3V.
- kSDHC\_Support4BitFlag* Support 4 bit mode.
- kSDHC\_Support8BitFlag* Support 8 bit mode.

### 30.6.3 enum \_sdhc\_wakeup\_event

Enumerator

- kSDHC\_WakeupEventOnCardInt* Wakeup on card interrupt.
- kSDHC\_WakeupEventOnCardInsert* Wakeup on card insertion.
- kSDHC\_WakeupEventOnCardRemove* Wakeup on card removal.
- kSDHC\_WakeupEventsAll* All wakeup events.

### 30.6.4 enum \_sdhc\_reset

Enumerator

- kSDHC\_ResetAll* Reset all except card detection.
- kSDHC\_ResetCommand* Reset command line.
- kSDHC\_ResetData* Reset data line.
- kSDHC\_ResetsAll* All reset types.

### 30.6.5 enum \_sdhc\_transfer\_flag

Enumerator

- kSDHC\_EnableDmaFlag* Enable DMA.
- kSDHC\_CommandTypeSuspendFlag* Suspend command.
- kSDHC\_CommandTypeResumeFlag* Resume command.
- kSDHC\_CommandTypeAbortFlag* Abort command.
- kSDHC\_EnableBlockCountFlag* Enable block count.
- kSDHC\_EnableAutoCommand12Flag* Enable auto CMD12.
- kSDHC\_DataReadFlag* Enable data read.
- kSDHC\_MultipleBlockFlag* Multiple block data read/write.

## Enumeration Type Documentation

*kSDHC\_ResponseLength136Flag* 136 bit response length  
*kSDHC\_ResponseLength48Flag* 48 bit response length  
*kSDHC\_ResponseLength48BusyFlag* 48 bit response length with busy status  
*kSDHC\_EnableCrcCheckFlag* Enable CRC check.  
*kSDHC\_EnableIndexCheckFlag* Enable index check.  
*kSDHC\_DataPresentFlag* Data present flag.

### 30.6.6 enum \_sdhc\_present\_status\_flag

Enumerator

*kSDHC\_CommandInhibitFlag* Command inhibit.  
*kSDHC\_DataInhibitFlag* Data inhibit.  
*kSDHC\_DataLineActiveFlag* Data line active.  
*kSDHC\_SdClockStableFlag* SD bus clock stable.  
*kSDHC\_WriteTransferActiveFlag* Write transfer active.  
*kSDHC\_ReadTransferActiveFlag* Read transfer active.  
*kSDHC\_BufferWriteEnableFlag* Buffer write enable.  
*kSDHC\_BufferReadEnableFlag* Buffer read enable.  
*kSDHC\_CardInsertedFlag* Card inserted.  
*kSDHC\_CommandLineLevelFlag* Command line signal level.  
*kSDHC\_Data0LineLevelFlag* Data0 line signal level.  
*kSDHC\_Data1LineLevelFlag* Data1 line signal level.  
*kSDHC\_Data2LineLevelFlag* Data2 line signal level.  
*kSDHC\_Data3LineLevelFlag* Data3 line signal level.  
*kSDHC\_Data4LineLevelFlag* Data4 line signal level.  
*kSDHC\_Data5LineLevelFlag* Data5 line signal level.  
*kSDHC\_Data6LineLevelFlag* Data6 line signal level.  
*kSDHC\_Data7LineLevelFlag* Data7 line signal level.

### 30.6.7 enum \_sdhc\_interrupt\_status\_flag

Enumerator

*kSDHC\_CommandCompleteFlag* Command complete.  
*kSDHC\_DataCompleteFlag* Data complete.  
*kSDHC\_BlockGapEventFlag* Block gap event.  
*kSDHC\_DmaCompleteFlag* DMA interrupt.  
*kSDHC\_BufferWriteReadyFlag* Buffer write ready.  
*kSDHC\_BufferReadReadyFlag* Buffer read ready.  
*kSDHC\_CardInsertionFlag* Card inserted.  
*kSDHC\_CardRemovalFlag* Card removed.

*kSDHC\_CardInterruptFlag* Card interrupt.  
*kSDHC\_CommandTimeoutFlag* Command timeout error.  
*kSDHC\_CommandCrcErrorFlag* Command CRC error.  
*kSDHC\_CommandEndBitErrorFlag* Command end bit error.  
*kSDHC\_CommandIndexErrorFlag* Command index error.  
*kSDHC\_DataTimeoutFlag* Data timeout error.  
*kSDHC\_DataCrcErrorFlag* Data CRC error.  
*kSDHC\_DataEndBitErrorFlag* Data end bit error.  
*kSDHC\_AutoCommand12ErrorFlag* Auto CMD12 error.  
*kSDHC\_DmaErrorFlag* DMA error.  
*kSDHC\_CommandErrorFlag* Command error.  
*kSDHC\_DataErrorFlag* Data error.  
*kSDHC\_ErrorFlag* All error.  
*kSDHC\_DataFlag* Data interrupts.  
*kSDHC\_CommandFlag* Command interrupts.  
*kSDHC\_CardDetectFlag* Card detection interrupts.  
*kSDHC\_AllInterruptFlags* All flags mask.

### 30.6.8 enum \_sdhc\_auto\_command12\_error\_status\_flag

Enumerator

*kSDHC\_AutoCommand12NotExecutedFlag* Not executed error.  
*kSDHC\_AutoCommand12TimeoutFlag* Timeout error.  
*kSDHC\_AutoCommand12EndBitErrorFlag* End bit error.  
*kSDHC\_AutoCommand12CrcErrorFlag* CRC error.  
*kSDHC\_AutoCommand12IndexErrorFlag* Index error.  
*kSDHC\_AutoCommand12NotIssuedFlag* Not issued error.

### 30.6.9 enum \_sdhc\_adma\_error\_status\_flag

Enumerator

*kSDHC\_AdmaLengthMismatchFlag* Length mismatch error.  
*kSDHC\_AdmaDescriptorErrorFlag* Descriptor error.

### 30.6.10 enum sdhc\_adma\_error\_state\_t

This state is the detail state when ADMA error has occurred.

## Enumeration Type Documentation

Enumerator

*kSDHC\_AdmaErrorStateStopDma* Stop DMA.  
*kSDHC\_AdmaErrorStateFetchDescriptor* Fetch descriptor.  
*kSDHC\_AdmaErrorStateChangeAddress* Change address.  
*kSDHC\_AdmaErrorStateTransferData* Transfer data.

### 30.6.11 enum \_sdhc\_force\_event

Enumerator

*kSDHC\_ForceEventAutoCommand12NotExecuted* Auto CMD12 not executed error.  
*kSDHC\_ForceEventAutoCommand12Timeout* Auto CMD12 timeout error.  
*kSDHC\_ForceEventAutoCommand12CrcError* Auto CMD12 CRC error.  
*kSDHC\_ForceEventEndBitError* Auto CMD12 end bit error.  
*kSDHC\_ForceEventAutoCommand12IndexError* Auto CMD12 index error.  
*kSDHC\_ForceEventAutoCommand12NotIssued* Auto CMD12 not issued error.  
*kSDHC\_ForceEventCommandTimeout* Command timeout error.  
*kSDHC\_ForceEventCommandCrcError* Command CRC error.  
*kSDHC\_ForceEventCommandEndBitError* Command end bit error.  
*kSDHC\_ForceEventCommandIndexError* Command index error.  
*kSDHC\_ForceEventDataTimeout* Data timeout error.  
*kSDHC\_ForceEventDataCrcError* Data CRC error.  
*kSDHC\_ForceEventDataEndBitError* Data end bit error.  
*kSDHC\_ForceEventAutoCommand12Error* Auto CMD12 error.  
*kSDHC\_ForceEventCardInt* Card interrupt.  
*kSDHC\_ForceEventDmaError* Dma error.  
*kSDHC\_ForceEventsAll* All force event flags mask.

### 30.6.12 enum sdhc\_data\_bus\_width\_t

Enumerator

*kSDHC\_DataBusWidth1Bit* 1-bit mode  
*kSDHC\_DataBusWidth4Bit* 4-bit mode  
*kSDHC\_DataBusWidth8Bit* 8-bit mode

### 30.6.13 enum sdhc\_endian\_mode\_t

Enumerator

*kSDHC\_EndianModeBig* Big endian mode.

*kSDHC\_EndianModeHalfWordBig* Half word big endian mode.

*kSDHC\_EndianModeLittle* Little endian mode.

### 30.6.14 enum sdhc\_dma\_mode\_t

Enumerator

*kSDHC\_DmaModeNo* No DMA.

*kSDHC\_DmaModeAdma1* ADMA1 is selected.

*kSDHC\_DmaModeAdma2* ADMA2 is selected.

### 30.6.15 enum \_sdhc\_sdio\_control\_flag

Enumerator

*kSDHC\_StopAtBlockGapFlag* Stop at block gap.

*kSDHC\_ReadWaitControlFlag* Read wait control.

*kSDHC\_InterruptAtBlockGapFlag* Interrupt at block gap.

*kSDHC\_ExactBlockNumberReadFlag* Exact block number read.

### 30.6.16 enum sdhc\_boot\_mode\_t

Enumerator

*kSDHC\_BootModeNormal* Normal boot.

*kSDHC\_BootModeAlternative* Alternative boot.

### 30.6.17 enum sdhc\_card\_command\_type\_t

Enumerator

*kCARD\_CommandTypeNormal* Normal command.

*kCARD\_CommandTypeSuspend* Suspend command.

*kCARD\_CommandTypeResume* Resume command.

*kCARD\_CommandTypeAbort* Abort command.

## Enumeration Type Documentation

### 30.6.18 enum sdhc\_card\_response\_type\_t

Define the command response type from card to host controller.

Enumerator

- kCARD\_ResponseNone* Response type: none.
- kCARD\_ResponseR1* Response type: R1.
- kCARD\_ResponseR1b* Response type: R1b.
- kCARD\_ResponseR2* Response type: R2.
- kCARD\_ResponseR3* Response type: R3.
- kCARD\_ResponseR4* Response type: R4.
- kCARD\_ResponseR5* Response type: R5.
- kCARD\_ResponseR5b* Response type: R5b.
- kCARD\_ResponseR6* Response type: R6.
- kCARD\_ResponseR7* Response type: R7.

### 30.6.19 enum \_sdhc\_adma1\_descriptor\_flag

Enumerator

- kSDHC\_Adma1DescriptorValidFlag* Valid flag.
- kSDHC\_Adma1DescriptorEndFlag* End flag.
- kSDHC\_Adma1DescriptorInterrupFlag* Interrupt flag.
- kSDHC\_Adma1DescriptorActivity1Flag* Activity 1 flag.
- kSDHC\_Adma1DescriptorActivity2Flag* Activity 2 flag.
- kSDHC\_Adma1DescriptorTypeNop* No operation.
- kSDHC\_Adma1DescriptorTypeTransfer* Transfer data.
- kSDHC\_Adma1DescriptorTypeLink* Link descriptor.
- kSDHC\_Adma1DescriptorTypeSetLength* Set data length.

### 30.6.20 enum \_sdhc\_adma2\_descriptor\_flag

Enumerator

- kSDHC\_Adma2DescriptorValidFlag* Valid flag.
- kSDHC\_Adma2DescriptorEndFlag* End flag.
- kSDHC\_Adma2DescriptorInterruptFlag* Interrupt flag.
- kSDHC\_Adma2DescriptorActivity1Flag* Activity 1 mask.
- kSDHC\_Adma2DescriptorActivity2Flag* Activity 2 mask.
- kSDHC\_Adma2DescriptorTypeNop* No operation.
- kSDHC\_Adma2DescriptorTypeReserved* Reserved.
- kSDHC\_Adma2DescriptorTypeTransfer* Transfer type.
- kSDHC\_Adma2DescriptorTypeLink* Link type.

## 30.7 Function Documentation

### 30.7.1 void SDHC\_Init ( **SDHC\_Type** \* *base*, **const sdhc\_config\_t** \* *config* )

Configures the SDHC according to the user configuration.

Example:

```
sdhc_config_t config;
config.cardDetectDat3 = false;
config.endianMode = kSDHC_EndianModeLittle;
config.dmaMode = kSDHC_DmaModeAdma2;
config.readWatermarkLevel = 128U;
config.writeWatermarkLevel = 128U;
SDHC_Init(SDHC, &config);
```

Parameters

<i>base</i>	SDHC peripheral base address.
<i>config</i>	SDHC configuration information.

Return values

<i>kStatus_Success</i>	Operate successfully.
------------------------	-----------------------

### 30.7.2 void SDHC\_Deinit ( **SDHC\_Type** \* *base* )

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

### 30.7.3 bool SDHC\_Reset ( **SDHC\_Type** \* *base*, **uint32\_t** *mask*, **uint32\_t** *timeout* )

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The reset type mask(_sdhc_reset).

## Function Documentation

<i>timeout</i>	Timeout for reset.
----------------	--------------------

Return values

<i>true</i>	Reset successfully.
<i>false</i>	Reset failed.

**30.7.4 `status_t SDHC_SetAdmaTableConfig ( SDHC_Type * base, sdhc_dma_mode_t dmaMode, uint32_t * table, uint32_t tableWords, const uint32_t * data, uint32_t dataBytes )`**

Parameters

<i>base</i>	SDHC peripheral base address.
<i>dmaMode</i>	DMA mode.
<i>table</i>	ADMA table address.
<i>tableWords</i>	ADMA table buffer length united as Words.
<i>data</i>	Data buffer address.
<i>dataBytes</i>	Data length united as bytes.

Return values

<i>kStatus_OutOfRange</i>	ADMA descriptor table length isn't enough to describe data.
<i>kStatus_Success</i>	Operate successfully.

**30.7.5 `static void SDHC_EnableInterruptStatus ( SDHC_Type * base, uint32_t mask ) [inline], [static]`**

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	Interrupt status flags mask(_sdhc_interrupt_status_flag).

**30.7.6 `static void SDHC_DisableInterruptStatus ( SDHC_Type * base, uint32_t mask ) [inline], [static]`**

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

### 30.7.7 static void SDHC\_EnableInterruptSignal ( **SDHC\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

### 30.7.8 static void SDHC\_DisableInterruptSignal ( **SDHC\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

### 30.7.9 static uint32\_t SDHC\_GetInterruptStatusFlags ( **SDHC\_Type** \* *base* ) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

Returns

Current interrupt status flags mask(\_sdhc\_interrupt\_status\_flag).

### 30.7.10 static void SDHC\_ClearInterruptStatusFlags ( **SDHC\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

## Function Documentation

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	The interrupt status flags mask(_sdhc_interrupt_status_flag).

**30.7.11 static uint32\_t SDHC\_GetAutoCommand12ErrorStatusFlags ( SDHC\_Type \* *base* ) [inline], [static]**

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

Returns

Auto command 12 error status flags mask(\_sdhc\_auto\_command12\_error\_status\_flag).

**30.7.12 static uint32\_t SDHC\_GetAdmaErrorStatusFlags ( SDHC\_Type \* *base* ) [inline], [static]**

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

Returns

ADMA error status flags mask(\_sdhc\_adma\_error\_status\_flag).

**30.7.13 static uint32\_t SDHC\_GetPresentStatusFlags ( SDHC\_Type \* *base* ) [inline], [static]**

This function gets the present SDHC's status except for an interrupt status and an error status.

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

Returns

Present SDHC's status flags mask(\_sdhc\_present\_status\_flag).

### 30.7.14 void SDHC\_GetCapability ( SDHC\_Type \* *base*, sdhc\_capability\_t \* *capability* )

Parameters

<i>base</i>	SDHC peripheral base address.
<i>capability</i>	Structure to save capability information.

### 30.7.15 static void SDHC\_EnableSdClock ( SDHC\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
<i>enable</i>	True to enable, false to disable.

### 30.7.16 uint32\_t SDHC\_SetSdClock ( SDHC\_Type \* *base*, uint32\_t *srcClock\_Hz*, uint32\_t *busClock\_Hz* )

Parameters

<i>base</i>	SDHC peripheral base address.
<i>srcClock_Hz</i>	SDHC source clock frequency united in Hz.
<i>busClock_Hz</i>	SD bus clock frequency united in Hz.

Returns

The nearest frequency of busClock\_Hz configured to SD bus.

## Function Documentation

### 30.7.17 **bool SDHC\_SetCardActive ( SDHC\_Type \* *base*, uint32\_t *timeout* )**

This function must be called each time the card is inserted to ensure that the card can receive the command correctly.

## Parameters

<i>base</i>	SDHC peripheral base address.
<i>timeout</i>	Timeout to initialize card.

## Return values

<i>true</i>	Set card active successfully.
<i>false</i>	Set card active failed.

**30.7.18 static void SDHC\_SetDataBusWidth ( SDHC\_Type \* *base*, sdhc\_data\_bus\_width\_t *width* ) [inline], [static]**

## Parameters

<i>base</i>	SDHC peripheral base address.
<i>width</i>	Data transfer width.

**30.7.19 void SDHC\_SetTransferConfig ( SDHC\_Type \* *base*, const sdhc\_transfer\_config\_t \* *config* )**

This function fills the card transfer-related command argument/transfer flag/data size. The command and data are sent by SDHC after calling this function.

Example:

```
sdhc_transfer_config_t transferConfig;
transferConfig.dataBlockSize = 512U;
transferConfig.dataBlockCount = 2U;
transferConfig.commandArgument = 0x01AAU;
transferConfig.commandIndex = 8U;
transferConfig.flags |= (kSDHC_EnableDmaFlag |
    kSDHC_EnableAutoCommand12Flag |
    kSDHC_MultipleBlockFlag);
SDHC_SetTransferConfig(SDHC, &transferConfig);
```

## Parameters

## Function Documentation

<i>base</i>	SDHC peripheral base address.
<i>config</i>	Command configuration structure.

**30.7.20 static uint32\_t SDHC\_GetCommandResponse ( SDHC\_Type \* *base*, uint32\_t *index* ) [inline], [static]**

Parameters

<i>base</i>	SDHC peripheral base address.
<i>index</i>	The index of response register, range from 0 to 3.

Returns

Response register transfer.

**30.7.21 static void SDHC\_WriteData ( SDHC\_Type \* *base*, uint32\_t *data* ) [inline], [static]**

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>data</i>	The data about to be sent.

**30.7.22 static uint32\_t SDHC\_ReadData ( SDHC\_Type \* *base* ) [inline], [static]**

This function is used to implement the data transfer by Data Port instead of DMA.

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

Returns

The data has been read.

30.7.23 **static void SDHC\_EnableWakeupEvent( SDHC\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]**

## Function Documentation

Parameters

<i>base</i>	SDHC peripheral base address.
<i>mask</i>	Wakeup events mask(_sdhc_wakeup_event).
<i>enable</i>	True to enable, false to disable.

**30.7.24 static void SDHC\_EnableCardDetectTest ( SDHC\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

<i>base</i>	SDHC peripheral base address.
<i>enable</i>	True to enable, false to disable.

**30.7.25 static void SDHC\_SetCardDetectTestLevel ( SDHC\_Type \* *base*, bool *high* ) [inline], [static]**

This function sets the card detection test level to indicate whether the card is inserted into the SDHC when DAT[3]/ CD pin is selected as a card detection pin. This function can also assert the pin logic when DAT[3]/CD pin is selected as the card detection pin.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>high</i>	True to set the card detect level to high.

**30.7.26 void SDHC\_EnableSdioControl ( SDHC\_Type \* *base*, uint32\_t *mask*, bool *enable* )**

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

<i>mask</i>	SDIO card control flags mask(_sdhc_sdio_control_flag).
<i>enable</i>	True to enable, false to disable.

### 30.7.27 static void SDHC\_SetContinueRequest ( SDHC\_Type \* *base* ) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

### 30.7.28 void SDHC\_SetMmcBootConfig ( SDHC\_Type \* *base*, const sdhc\_boot\_config\_t \* *config* )

Example:

```
sdhc_boot_config_t config;
config.ackTimeoutCount = 4;
config.bootMode = kSDHC_BootModeNormal;
config.blockCount = 5;
config.enableBootAck = true;
config.enableBoot = true;
config.enableAutoStopAtBlockGap = true;
SDHC_SetMmcBootConfig(SDHC, &config);
```

Parameters

<i>base</i>	SDHC peripheral base address.
<i>config</i>	The MMC boot configuration information.

### 30.7.29 static void SDHC\_SetForceEvent ( SDHC\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

<i>base</i>	SDHC peripheral base address.
-------------	-------------------------------

## Function Documentation

<i>mask</i>	The force events mask(_sdhc_force_event).
-------------	---

### 30.7.30 **status\_t SDHC\_TransferBlocking ( SDHC\_Type \* *base*, uint32\_t \* *admaTable*, uint32\_t *admaTableWords*, sdhc\_transfer\_t \* *transfer* )**

This function waits until the command response/data is received or the SDHC encounters an error by polling the status flag. This function support non word align data addr transfer support, if data buffer addr is not align in DMA mode, the API will continue finish the transfer by polling IO directly. The application must not call this API in multiple threads at the same time. Because of that this API doesn't support the re-entry mechanism.

#### Note

There is no need to call the API 'SDHC\_TransferCreateHandle' when calling this API.

#### Parameters

<i>base</i>	SDHC peripheral base address.
<i>admaTable</i>	ADMA table address, can't be null if transfer way is ADMA1/ADMA2.
<i>admaTable-Words</i>	ADMA table length united as words, can't be 0 if transfer way is ADMA1/ADMA2.
<i>transfer</i>	Transfer content.

#### Return values

<i>kStatus_InvalidArgument</i>	Argument is invalid.
<i>kStatus_SDHC_PreparesAdmaDescriptorFailed</i>	Prepare ADMA descriptor failed.
<i>kStatus_SDHC_SendCommandFailed</i>	Send command failed.
<i>kStatus_SDHC_TransferDataFailed</i>	Transfer data failed.
<i>kStatus_Success</i>	Operate successfully.

### 30.7.31 **void SDHC\_TransferCreateHandle ( SDHC\_Type \* *base*, sdhc\_handle\_t \* *handle*, const sdhc\_transfer\_callback\_t \* *callback*, void \* *userData* )**

## Parameters

<i>base</i>	SDHC peripheral base address.
<i>handle</i>	SDHC handle pointer.
<i>callback</i>	Structure pointer to contain all callback functions.
<i>userData</i>	Callback function parameter.

**30.7.32 `status_t SDHC_TransferNonBlocking ( SDHC_Type * base, sdhc_handle_t * handle, uint32_t * admaTable, uint32_t admaTableWords, sdhc_transfer_t * transfer )`**

This function sends a command and data and returns immediately. It doesn't wait the transfer complete or encounter an error. This function support non word align data addr transfer support, if data buffer addr is not align in DMA mode, the API will continue finish the transfer by polling IO directly. The application must not call this API in multiple threads at the same time. Because of that this API doesn't support the re-entry mechanism.

## Note

Call the API 'SDHC\_TransferCreateHandle' when calling this API.

## Parameters

<i>base</i>	SDHC peripheral base address.
<i>handle</i>	SDHC handle.
<i>admaTable</i>	ADMA table address, can't be null if transfer way is ADMA1/ADMA2.
<i>admaTable-Words</i>	ADMA table length united as words, can't be 0 if transfer way is ADMA1/ADMA2.
<i>transfer</i>	Transfer content.

## Return values

<i>kStatus_InvalidArgument</i>	Argument is invalid.
<i>kStatus_SDHC_Busy-Transferring</i>	Busy transferring.

## Function Documentation

<i>kStatus_SDHC_PrepAdmaDescriptorFailed</i>	Prepare ADMA descriptor failed.
<i>kStatus_Success</i>	Operate successfully.

### 30.7.33 void SDHC\_TransferHandleIRQ ( **SDHC\_Type** \* *base*, **sdhc\_handle\_t** \* *handle* )

This function deals with the IRQs on the given host controller.

Parameters

<i>base</i>	SDHC peripheral base address.
<i>handle</i>	SDHC handle.

# Chapter 31

## SIM: System Integration Module Driver

### 31.1 Overview

The MCUXpresso SDK provides a peripheral driver for the System Integration Module (SIM) of MCUXpresso SDK devices.

### Data Structures

- struct `sim_uid_t`  
*Unique ID. [More...](#)*

### Enumerations

- enum `_sim_usb_volt_reg_enable_mode` {  
  `kSIM_UsbVoltRegEnable` = SIM\_SOPT1\_USBREGEN\_MASK,  
  `kSIM_UsbVoltRegEnableInLowPower` = SIM\_SOPT1\_USBVSTBY\_MASK,  
  `kSIM_UsbVoltRegEnableInStop` = SIM\_SOPT1\_USBSSTBY\_MASK,  
  `kSIM_UsbVoltRegEnableInAllModes` }  
*USB voltage regulator enable setting.*
- enum `_sim_flash_mode` {  
  `kSIM_FlashDisableInWait` = SIM\_FCFG1\_FLASHDOZE\_MASK,  
  `kSIM_FlashDisable` = SIM\_FCFG1\_FLASHDIS\_MASK }  
*Flash enable mode.*

### Functions

- void `SIM_SetUsbVoltRegulatorEnableMode` (uint32\_t mask)  
*Sets the USB voltage regulator setting.*
- void `SIM_GetUniqueId` (`sim_uid_t *uid`)  
*Gets the unique identification register value.*
- static void `SIM_SetFlashMode` (uint8\_t mode)  
*Sets the flash enable mode.*

### Driver version

- #define `FSL_SIM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
*Driver version 2.0.0.*

## Function Documentation

### 31.2 Data Structure Documentation

#### 31.2.1 struct sim\_uid\_t

##### Data Fields

- uint32\_t **MH**  
*UIDMH.*
- uint32\_t **ML**  
*UIDML.*
- uint32\_t **L**  
*UIDL.*

##### 31.2.1.0.0.71 Field Documentation

###### 31.2.1.0.0.71.1 uint32\_t sim\_uid\_t::MH

###### 31.2.1.0.0.71.2 uint32\_t sim\_uid\_t::ML

###### 31.2.1.0.0.71.3 uint32\_t sim\_uid\_t::L

### 31.3 Enumeration Type Documentation

#### 31.3.1 enum \_sim\_usb\_volt\_reg\_enable\_mode

Enumerator

***kSIM\_UsbVoltRegEnable*** Enable voltage regulator.

***kSIM\_UsbVoltRegEnableInLowPower*** Enable voltage regulator in VLPR/VLPW modes.

***kSIM\_UsbVoltRegEnableInStop*** Enable voltage regulator in STOP/VLPS/LLS/VLLS modes.

***kSIM\_UsbVoltRegEnableInAllModes*** Enable voltage regulator in all power modes.

#### 31.3.2 enum \_sim\_flash\_mode

Enumerator

***kSIM\_FlashDisableInWait*** Disable flash in wait mode.

***kSIM\_FlashDisable*** Disable flash in normal mode.

### 31.4 Function Documentation

#### 31.4.1 void SIM\_SetUsbVoltRegulatorEnableMode ( uint32\_t *mask* )

This function configures whether the USB voltage regulator is enabled in normal RUN mode, STOP-/VLPS/LLS/VLLS modes, and VLPR/VLPW modes. The configurations are passed in as mask value of `_sim_usb_volt_reg_enable_mode`. For example, to enable USB voltage regulator in RUN/VLPR/VLPW modes and disable in STOP/VLPS/LLS/VLLS mode, use:

```
SIM_SetUsbVoltRegulatorEnableMode(kSIM_UsbVoltRegEnable | kSIM_UsbVoltRegEnableInLowPower);
```

## Function Documentation

Parameters

<i>mask</i>	USB voltage regulator enable setting.
-------------	---------------------------------------

### 31.4.2 void SIM\_GetUniqueId ( sim\_uid\_t \* *uid* )

Parameters

<i>uid</i>	Pointer to the structure to save the UID value.
------------	---

### 31.4.3 static void SIM\_SetFlashMode ( uint8\_t *mode* ) [inline], [static]

Parameters

<i>mode</i>	The mode to set; see <a href="#"><u>_sim_flash_mode</u></a> for mode details.
-------------	---

# Chapter 32

## Smart Card

### 32.1 Overview

The MCUXpresso SDK provides Peripheral drivers for the UART-ISO7816 and EMVSIM modules of MCUXpresso SDK devices.

Smart Card driver provides the necessary functions to access and control integrated circuit cards. The driver controls communication modules (UART/EMVSIM) and handles special ICC sequences, such as the activation/deactivation (using EMVSIM IP or external interface chip). The Smart Card driver consists of two IPs (SmartCard\_Uart and SmartCard\_EmvSim drivers) and three PHY drivers (smartcard\_phy\_emvsim, smartcard\_phy\_tda8035, and smartcard\_phy\_gpio drivers). These drivers can be combined, which means that the Smart Card driver wraps one IP (transmission) and one PHY (interface) driver.

The driver provides asynchronous functions to communicate with the Integrated Circuit Card (ICC). The driver contains RTOS adaptation layers which use semaphores as synchronization objects of synchronous transfers. The RTOS driver support also provides protection for multithreading.

### 32.2 SmartCard Driver Initialization

The Smart Card Driver is initialized by calling the [SMARTCARD\\_Init\(\)](#) and [SMARTCARD\\_PHY\\_Init\(\)](#) functions. The Smart Card Driver initialization configuration structure requires these settings:

- Smart Card voltage class
- Smart Card Interface options such as the RST, IRQ, CLK pins, and so on.

The driver also supports user callbacks for assertion/de-assertion Smart Card events and transfer finish event. This feature is useful to detect the card presence or for handling transfer events i.e., in RTOS. The user should initialize the Smart Card driver, which consist of IP and PHY drivers.

### 32.3 SmartCard Call diagram

Because the call diagram is complex, the detailed use of the Smart Card driver is not described in this part. For details about using the Smart Card driver, see the Smart Card driver example which describes a simple use case.

## PHY driver

The Smart Card interface driver is initialized by calling the function [SMARTCARD\\_PHY\\_Init\(\)](#). During the initialization phase, Smart Card clock is configured and all hardware pins for IC handling are configured.

## Modules

- Smart Card EMVSIM Driver

## SmartCard Call diagram

- Smart Card FreeRTOS Driver
- Smart Card PHY EMVSIM Driver
- Smart Card PHY GPIO Driver
- Smart Card PHY TDA8035 Driver
- Smart Card UART Driver

## Data Structures

- struct `smartcard_card_params_t`  
*Defines card-specific parameters for Smart card driver. [More...](#)*
- struct `smartcard_timers_state_t`  
*Smart card defines the state of the EMV timers in the Smart card driver. [More...](#)*
- struct `smartcard_interface_config_t`  
*Defines user specified configuration of Smart card interface. [More...](#)*
- struct `smartcard_xfer_t`  
*Defines user transfer structure used to initialize transfer. [More...](#)*
- struct `smartcard_context_t`  
*Runtime state of the Smart card driver. [More...](#)*

## Macros

- #define `SMARTCARD_INIT_DELAY_CLOCK_CYCLES` (42000u)  
*Smart card global define which specify number of clock cycles until initial 'TS' character has to be received.*
- #define `SMARTCARD_EMV_ATR_DURATION_ETU` (20150u)  
*Smart card global define which specify number of clock cycles during which ATR string has to be received.*
- #define `SMARTCARD_TS_DIRECT_CONVENTION` (0x3Bu)  
*Smart card specification initial TS character definition of direct convention.*
- #define `SMARTCARD_TS_INVERSE_CONVENTION` (0x3Fu)  
*Smart card specification initial TS character definition of inverse convention.*

## Typedefs

- typedef void(\* `smartcard_interface_callback_t`)(void \*smartcardContext, void \*param)  
*Smart card interface interrupt callback function type.*
- typedef void(\* `smartcard_transfer_callback_t`)(void \*smartcardContext, void \*param)  
*Smart card transfer interrupt callback function type.*
- typedef void(\* `smartcard_time_delay_t`)(uint32\_t us)  
*Time Delay function used to passive waiting using RTOS [us].*

## Enumerations

- enum `smartcard_status_t` {  
  kStatus\_SMARTCARD\_Success = MAKE\_STATUS(kStatusGroup\_SMARTCARD, 0),  
  kStatus\_SMARTCARD\_TxBusy = MAKE\_STATUS(kStatusGroup\_SMARTCARD, 1),  
  kStatus\_SMARTCARD\_RxBusy = MAKE\_STATUS(kStatusGroup\_SMARTCARD, 2),  
  kStatus\_SMARTCARD\_NoTransferInProgress = MAKE\_STATUS(kStatusGroup\_SMARTCAR-

```
D, 3),
kStatus_SMARTCARD_Timeout = MAKE_STATUS(kStatusGroup_SMARTCARD, 4),
kStatus_SMARTCARD_Initialized,
kStatus_SMARTCARD_PhyInitialized,
kStatus_SMARTCARD_CardNotActivated = MAKE_STATUS(kStatusGroup_SMARTCARD, 7),
kStatus_SMARTCARD_InvalidInput,
kStatus_SMARTCARD_OtherError = MAKE_STATUS(kStatusGroup_SMARTCARD, 9) }
```

*Smart card Error codes.*

- enum **smartcard\_control\_t**  
*Control codes for the Smart card protocol timers and misc.*
- enum **smartcard\_card\_voltage\_class\_t**  
*Defines Smart card interface voltage class values.*
- enum **smartcard\_transfer\_state\_t**  
*Defines Smart card I/O transfer states.*
- enum **smartcard\_reset\_type\_t**  
*Defines Smart card reset types.*
- enum **smartcard\_transport\_type\_t**  
*Defines Smart card transport protocol types.*
- enum **smartcard\_parity\_type\_t**  
*Defines Smart card data parity types.*
- enum **smartcard\_card\_convention\_t**  
*Defines data Convention format.*
- enum **smartcard\_interface\_control\_t**  
*Defines Smart card interface IC control types.*
- enum **smartcard\_direction\_t**  
*Defines transfer direction.*

## Driver version

- #define **FSL\_SMARTCARD\_DRIVER\_VERSION** (MAKE\_VERSION(2, 2, 0))  
*Smart card driver version 2.2.0.*

## 32.4 Data Structure Documentation

### 32.4.1 struct smartcard\_card\_params\_t

#### Data Fields

- uint16\_t **Fi**  
*4 bits Fi - clock rate conversion integer*
- uint8\_t **fMax**  
*Maximum Smart card frequency in MHz.*
- uint8\_t **WI**  
*8 bits WI - work wait time integer*
- uint8\_t **Di**  
*4 bits DI - baud rate divisor*
- uint8\_t **BWI**  
*4 bits BWI - block wait time integer*
- uint8\_t **CWI**

## Data Structure Documentation

- `uint8_t BGI`  
*4 bits CWI - character wait time integer*
- `uint8_t GTN`  
*4 bits BGI - block guard time integer*
- `uint8_t IFSC`  
*8 bits GTN - extended guard time integer*
- `uint8_t modeNegotiable`  
*Indicates IFSC value of the card.*
- `uint8_t currentD`  
*Indicates if the card acts in negotiable or a specific mode.*
- `uint8_t status`  
*Indicates smart card status.*
- `bool t0Indicated`  
*Indicates ff T=0 indicated in TD1 byte.*
- `bool t1Indicated`  
*Indicates if T=1 indicated in TD2 byte.*
- `bool atrComplete`  
*Indicates whether the ATR received from the card was complete or not.*
- `bool atrValid`  
*Indicates whether the ATR received from the card was valid or not.*
- `bool present`  
*Indicates if a smart card is present.*
- `bool active`  
*Indicates if the smart card is activated.*
- `bool faulty`  
*Indicates whether smart card/interface is faulty.*
- `smartcard_card_convention_t convention`  
*Card convention, kSMARTCARD\_DirectConvention for direct convention, kSMARTCARD\_InverseConvention for inverse convention.*

### 32.4.1.0.0.72 Field Documentation

#### 32.4.1.0.0.72.1 `uint8_t smartcard_card_params_t::modeNegotiable`

### 32.4.2 `struct smartcard_timers_state_t`

#### Data Fields

- `volatile bool adtExpired`  
*Indicates whether ADT timer expired.*
- `volatile bool wwtExpired`  
*Indicates whether WWT timer expired.*
- `volatile bool cwtExpired`  
*Indicates whether CWT timer expired.*
- `volatile bool bwtExpired`  
*Indicates whether BWT timer expired.*
- `volatile bool initCharTimerExpired`  
*Indicates whether reception timer for initialization character (TS) after the RST has expired*

### 32.4.3 struct smartcard\_interface\_config\_t

#### Data Fields

- `uint32_t smartCardClock`  
*Smart card interface clock [Hz].*
- `uint32_t clockToResetDelay`  
*Indicates clock to RST apply delay [smart card clock cycles].*
- `uint8_t clockModule`  
*Smart card clock module number.*
- `uint8_t clockModuleChannel`  
*Smart card clock module channel number.*
- `uint8_t clockModuleSourceClock`  
*Smart card clock module source clock [e.g., BusClk].*
- `smartcard_card_voltage_class_t vcc`  
*Smart card voltage class.*
- `uint8_t controlPort`  
*Smart card PHY control port instance.*
- `uint8_t controlPin`  
*Smart card PHY control pin instance.*
- `uint8_t irqPort`  
*Smart card PHY Interrupt port instance.*
- `uint8_t irqPin`  
*Smart card PHY Interrupt pin instance.*
- `uint8_t resetPort`  
*Smart card reset port instance.*
- `uint8_t resetPin`  
*Smart card reset pin instance.*
- `uint8_t vsel0Port`  
*Smart card PHY Vsel0 control port instance.*
- `uint8_t vsel0Pin`  
*Smart card PHY Vsel0 control pin instance.*
- `uint8_t vsel1Port`  
*Smart card PHY Vsel1 control port instance.*
- `uint8_t vsel1Pin`  
*Smart card PHY Vsel1 control pin instance.*
- `uint8_t dataPort`  
*Smart card PHY data port instance.*
- `uint8_t dataPin`  
*Smart card PHY data pin instance.*
- `uint8_t dataPinMux`  
*Smart card PHY data pin mux option.*
- `uint8_t tsTimerId`  
*Numerical identifier of the External HW timer for Initial character detection.*

## Data Structure Documentation

### 32.4.4 struct smartcard\_xfer\_t

#### Data Fields

- **smartcard\_direction\_t direction**  
*Direction of communication.*
- **uint8\_t \* buff**  
*The buffer of data.*
- **size\_t size**  
*The number of transferred units.*

#### 32.4.4.0.0.73 Field Documentation

##### 32.4.4.0.0.73.1 smartcard\_direction\_t smartcard\_xfer\_t::direction

(RX/TX)

##### 32.4.4.0.0.73.2 uint8\_t\* smartcard\_xfer\_t::buff

##### 32.4.4.0.0.73.3 size\_t smartcard\_xfer\_t::size

### 32.4.5 struct smartcard\_context\_t

#### Data Fields

- **void \* base**  
*Smart card module base address.*
- **smartcard\_direction\_t direction**  
*Direction of communication.*
- **uint8\_t \* xBuff**  
*The buffer of data being transferred.*
- **volatile size\_t xSize**  
*The number of bytes to be transferred.*
- **volatile bool xIsBusy**  
*True if there is an active transfer.*
- **uint8\_t txFifoEntryCount**  
*Number of data word entries in transmit FIFO.*
- **uint8\_t rxFifoThreshold**  
*The max value of the receiver FIFO threshold.*
- **smartcard\_interface\_callback\_t interfaceCallback**  
*Callback to invoke after interface IC raised interrupt.*
- **smartcard\_transfer\_callback\_t transferCallback**  
*Callback to invoke after transfer event occur.*
- **void \* interfaceCallbackParam**  
*Interface callback parameter pointer.*
- **void \* transferCallbackParam**  
*Transfer callback parameter pointer.*
- **smartcard\_time\_delay\_t timeDelay**  
*Function which handles time delay defined by user or RTOS.*

- **smartcard\_reset\_type\_t resetType**  
*Indicates whether a Cold reset or Warm reset was requested.*
- **smartcard\_transport\_type\_t tType**  
*Indicates current transfer protocol (T0 or T1)*
- volatile **smartcard\_transfer\_state\_t transferState**  
*Indicates the current transfer state.*
- **smartcard\_timers\_state\_t timersState**  
*Indicates the state of different protocol timers used in driver.*
- **smartcard\_card\_params\_t cardParams**  
*Smart card parameters(ATR and current) and interface slots states(ATR and current)*
- **uint8\_t IFSD**  
*Indicates the terminal IFSD.*
- **smartcard\_parity\_type\_t parity**  
*Indicates current parity even/odd.*
- volatile bool **rxtCrossed**  
*Indicates whether RXT thresholds has been crossed.*
- volatile bool **txtCrossed**  
*Indicates whether TXT thresholds has been crossed.*
- volatile bool **wtxRequested**  
*Indicates whether WTX has been requested or not.*
- volatile bool **parityError**  
*Indicates whether a parity error has been detected.*
- **uint8\_t statusBytes [2]**  
*Used to store Status bytes SW1, SW2 of the last executed card command response.*
- **smartcard\_interface\_config\_t interfaceConfig**  
*Smart card interface configuration structure.*
- bool **abortTransfer**  
*Used to abort transfer.*

### 32.4.5.0.0.74 Field Documentation

#### 32.4.5.0.0.74.1 smartcard\_direction\_t smartcard\_context\_t::direction

(RX/TX)

## Enumeration Type Documentation

- 32.4.5.0.0.74.2 `uint8_t* smartcard_context_t::xBuff`
- 32.4.5.0.0.74.3 `volatile size_t smartcard_context_t::xSize`
- 32.4.5.0.0.74.4 `volatile bool smartcard_context_t::xIsBusy`
- 32.4.5.0.0.74.5 `uint8_t smartcard_context_t::txFifoEntryCount`
- 32.4.5.0.0.74.6 `uint8_t smartcard_context_t::rxFifoThreshold`
- 32.4.5.0.0.74.7 `smartcard_interface_callback_t smartcard_context_t::interfaceCallback`
- 32.4.5.0.0.74.8 `smartcard_transfer_callback_t smartcard_context_t::transferCallback`
- 32.4.5.0.0.74.9 `void* smartcard_context_t::interfaceCallbackParam`
- 32.4.5.0.0.74.10 `void* smartcard_context_t::transferCallbackParam`
- 32.4.5.0.0.74.11 `smartcard_time_delay_t smartcard_context_t::timeDelay`
- 32.4.5.0.0.74.12 `smartcard_reset_type_t smartcard_context_t::resetType`
- 32.4.5.0.0.74.13 `bool smartcard_context_t::abortTransfer`

## 32.5 Enumeration Type Documentation

### 32.5.1 enum smartcard\_status\_t

Enumerator

- `kStatus_SMARTCARD_Success` Transfer ends successfully.
- `kStatus_SMARTCARD_TxBusy` Transmit in progress.
- `kStatus_SMARTCARD_RxBusy` Receiving in progress.
- `kStatus_SMARTCARD_NoTransferInProgress` No transfer in progress.
- `kStatus_SMARTCARD_Timeout` Transfer ends with time-out.
- `kStatus_SMARTCARD_Initialized` Smart card driver is already initialized.
- `kStatus_SMARTCARD_PhyInitialized` Smart card PHY drive is already initialized.
- `kStatus_SMARTCARD_CardNotActivated` Smart card is not activated.
- `kStatus_SMARTCARD_InvalidInput` Function called with invalid input arguments.
- `kStatus_SMARTCARD_OtherError` Some other error occur.

### 32.5.2 enum smartcard\_control\_t

### 32.5.3 enum smartcard\_direction\_t

## 32.6 Smart Card PHY TDA8035 Driver

### 32.6.1 Overview

The Smart Card interface TDA8035 driver handles the external interface chip TDA8035 which supports all necessary functions to control the ICC. These functions involve PHY pin initialization, ICC voltage selection and activation, ICC clock generation, ICC card detection, and activation/deactivation sequences.

## Macros

- #define **SMARTCARD\_ATR\_DURATION\_ADJUSTMENT** (360u)  
*Smart card definition which specifies the adjustment number of clock cycles during which an ATR string has to be received.*
- #define **SMARTCARD\_INIT\_DELAY\_CLOCK\_CYCLES\_ADJUSTMENT** (4200u)  
*Smart card definition which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.*
- #define **SMARTCARD\_TDA8035\_STATUS\_PRES** (0x01u)  
*Masks for TDA8035 status register.*
- #define **SMARTCARD\_TDA8035\_STATUS\_ACTIVE** (0x02u)  
*Smart card PHY TDA8035 Smart card active status.*
- #define **SMARTCARD\_TDA8035\_STATUS\_FAULTY** (0x04u)  
*Smart card PHY TDA8035 Smart card faulty status.*
- #define **SMARTCARD\_TDA8035\_STATUS\_CARD\_REMOVED** (0x08u)  
*Smart card PHY TDA8035 Smart card removed status.*
- #define **SMARTCARD\_TDA8035\_STATUS\_CARD\_DEACTIVATED** (0x10u)  
*Smart card PHY TDA8035 Smart card deactivated status.*

## Functions

- void **SMARTCARD\_PHY\_TDA8035\_GetDefaultConfig** (smartcard\_interface\_config\_t \*config)  
*Fills in the configuration structure with default values.*
- status\_t **SMARTCARD\_PHY\_TDA8035\_Init** (void \*base, smartcard\_interface\_config\_t const \*config, uint32\_t srcClock\_Hz)  
*Initializes a Smart card interface instance.*
- void **SMARTCARD\_PHY\_TDA8035\_Deinit** (void \*base, smartcard\_interface\_config\_t \*config)  
*De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.*
- status\_t **SMARTCARD\_PHY\_TDA8035\_Activate** (void \*base, smartcard\_context\_t \*context, smartcard\_reset\_type\_t resetType)  
*Activates the Smart card IC.*
- status\_t **SMARTCARD\_PHY\_TDA8035\_Deactivate** (void \*base, smartcard\_context\_t \*context)  
*De-activates the Smart card IC.*
- status\_t **SMARTCARD\_PHY\_TDA8035\_Control** (void \*base, smartcard\_context\_t \*context, smartcard\_interface\_control\_t control, uint32\_t param)  
*Controls the Smart card interface IC.*
- void **SMARTCARD\_PHY\_TDA8035\_IRQHandler** (void \*base, smartcard\_context\_t \*context)  
*Smart card interface IC IRQ ISR.*

### 32.6.2 Macro Definition Documentation

32.6.2.1 `#define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)`

32.6.2.2 `#define SMARTCARD_TDA8035_STATUS_PRES (0x01u)`

Smart card PHY TDA8035 Smart card present status

### 32.6.3 Function Documentation

32.6.3.1 `void SMARTCARD_PHY_TDA8035_GetDefaultConfig ( smartcard_interface_config_t * config )`

Parameters

<i>config</i>	The Smart card user configuration structure which contains configuration structure of type <code>smartcard_interface_config_t</code> . Function fill in members: <code>clockToResetDelay</code> = 42000, <code>vcc</code> = <code>kSmartcardVoltageClassB3_3V</code> , with default values.
---------------	---

32.6.3.2 `status_t SMARTCARD_PHY_TDA8035_Init ( void * base, smartcard_interface_config_t const * config, uint32_t srcClock_Hz )`

Parameters

<i>base</i>	The Smart card peripheral base address.
<i>config</i>	The user configuration structure of type <code>smartcard_interface_config_t</code> . The user can call to fill out configuration structure function <code>SMARTCARD_PHY_TDA8035_GetDefaultConfig()</code> .
<i>srcClock_Hz</i>	Smart card clock generation module source clock.

Return values

<i>kStatus_SMARTCARD_Success</i>	or <code>kStatus_SMARTCARD_OtherError</code> for an error.
----------------------------------	--

32.6.3.3 `void SMARTCARD_PHY_TDA8035_Deinit ( void * base, smartcard_interface_config_t * config )`

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>config</i>	The user configuration structure of type <a href="#">smartcard_interface_config_t</a> .

### 32.6.3.4 status\_t SMARTCARD\_PHY\_TDA8035\_Activate ( void \* *base*, smartcard\_context\_t \* *context*, smartcard\_reset\_type\_t *resetType* )

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>resetType</i>	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcardWarmReset

Return values

<i>kStatus_SMARTCARD_Success</i>	or kStatus_SMARTCARD_OtherError for an error.
----------------------------------	---

### 32.6.3.5 status\_t SMARTCARD\_PHY\_TDA8035\_Deactivate ( void \* *base*, smartcard\_context\_t \* *context* )

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.

Return values

<i>kStatus_SMARTCARD_Success</i>	or kStatus_SMARTCARD_OtherError for an error.
----------------------------------	---

### 32.6.3.6 status\_t SMARTCARD\_PHY\_TDA8035\_Control ( void \* *base*, smartcard\_context\_t \* *context*, smartcard\_interface\_control\_t *control*, uint32\_t *param* )

## Smart Card PHY TDA8035 Driver

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>control</i>	A interface command type.
<i>param</i>	Integer value specific to control type

Return values

<i>kStatus_SMARTCARD_Success</i>	or kStatus_SMARTCARD_OtherError for an error.
----------------------------------	---

### 32.6.3.7 void SMARTCARD\_PHY\_TDA8035\_IRQHandler ( void \* *base*, smartcard\_context\_t \* *context* )

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	The Smart card context pointer.

## 32.7 Smart Card PHY EMVSIM Driver

### 32.7.1 Overview

The Smart Card interface EMVSIM driver handles the EMVSIM peripheral, which covers all necessary functions to control the ICC. These functions are ICC clock setup, ICC voltage turning on/off, ICC card detection, activation/deactivation, and ICC reset sequences. The EMVSIM peripheral covers all features of interface ICC chips.

## Macros

- #define **SMARTCARD\_ATR\_DURATION\_ADJUSTMENT** (360u)  
*Smart card define which specifies the adjustment number of clock cycles during which an ATR string has to be received.*
- #define **SMARTCARD\_INIT\_DELAY\_CLOCK\_CYCLES\_ADJUSTMENT** (4200u)  
*Smart card define which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.*

## Functions

- void **SMARTCARD\_PHY\_EMVSIM\_GetDefaultConfig** (smartcard\_interface\_config\_t \*config)  
*Fills in the smartcardInterfaceConfig structure with default values.*
- status\_t **SMARTCARD\_PHY\_EMVSIM\_Init** (EMVSIM\_Type \*base, const smartcard\_interface\_config\_t \*config, uint32\_t srcClock\_Hz)  
*Configures a Smart card interface.*
- void **SMARTCARD\_PHY\_EMVSIM\_Deinit** (EMVSIM\_Type \*base, const smartcard\_interface\_config\_t \*config)  
*De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.*
- status\_t **SMARTCARD\_PHY\_EMVSIM\_Activate** (EMVSIM\_Type \*base, smartcard\_context\_t \*context, smartcard\_reset\_type\_t resetType)  
*Activates the Smart card IC.*
- status\_t **SMARTCARD\_PHY\_EMVSIM\_Deactivate** (EMVSIM\_Type \*base, smartcard\_context\_t \*context)  
*De-activates the Smart card IC.*
- status\_t **SMARTCARD\_PHY\_EMVSIM\_Control** (EMVSIM\_Type \*base, smartcard\_context\_t \*context, smartcard\_interface\_control\_t control, uint32\_t param)  
*Controls the Smart card interface IC.*

### 32.7.2 Macro Definition Documentation

32.7.2.1 `#define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)`

### 32.7.3 Function Documentation

32.7.3.1 `void SMARTCARD_PHY_EMVSIM_GetDefaultConfig ( smartcard_interface_config_t * config )`

Parameters

<i>config</i>	The user configuration structure of type <a href="#">smartcard_interface_config_t</a> . Function fill in members: clockToResetDelay = 42000, vcc = kSmartcardVoltageClassB3_3V, with default values.
---------------	--

### 32.7.3.2 **status\_t SMARTCARD\_PHY\_EMVSIM\_Init ( EMVSIM\_Type \* *base*, const smartcard\_interface\_config\_t \* *config*, uint32\_t *srcClock\_Hz* )**

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>config</i>	The user configuration structure of type <a href="#">smartcard_interface_config_t</a> . The user is responsible to fill out the members of this structure and to pass the pointer of this structure into this function or call SMARTCARD_PHY_EMVSIMInitUserConfig. Default to fill out structure with default values.
<i>srcClock_Hz</i>	Smart card clock generation module source clock.

Return values

<i>kStatus_SMARTCARD_-Success</i>	or <i>kStatus_SMARTCARD_OtherError</i> for an error.
-----------------------------------	--

### 32.7.3.3 **void SMARTCARD\_PHY\_EMVSIM\_Deinit ( EMVSIM\_Type \* *base*, const smartcard\_interface\_config\_t \* *config* )**

Parameters

<i>base</i>	Smart card peripheral module base address.
<i>config</i>	Smart card configuration structure.

### 32.7.3.4 **status\_t SMARTCARD\_PHY\_EMVSIM\_Activate ( EMVSIM\_Type \* *base*, smartcard\_context\_t \* *context*, smartcard\_reset\_type\_t *resetType* )**

## Smart Card PHY EMVSIM Driver

Parameters

<i>base</i>	The EMVSIM peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>resetType</i>	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Return values

<i>kStatus_SMARTCARD_Success</i>	or kStatus_SMARTCARD_OtherError for an error.
----------------------------------	---

### 32.7.3.5 status\_t SMARTCARD\_PHY\_EMVSIM\_Deactivate ( **EMVSIM\_Type \* base**, **smartcard\_context\_t \* context** )

Parameters

<i>base</i>	The EMVSIM peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.

Return values

<i>kStatus_SMARTCARD_Success</i>	or kStatus_SMARTCARD_OtherError for an error.
----------------------------------	---

### 32.7.3.6 status\_t SMARTCARD\_PHY\_EMVSIM\_Control ( **EMVSIM\_Type \* base**, **smartcard\_context\_t \* context**, **smartcard\_interface\_control\_t control**, **uint32\_t param** )

Parameters

<i>base</i>	The EMVSIM peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>control</i>	A interface command type.

<i>param</i>	Integer value specific to control type
--------------	--

Return values

<i>kStatus_SMARTCARD_-Success</i>	or kStatus_SMARTCARD_OtherError for an error.
-----------------------------------	---

### 32.8 Smart Card PHY GPIO Driver

#### 32.8.1 Overview

The Smart Card interface GPIO driver handles the GPIO and FTM/TPM peripheral for clock generation, which covers all necessary functions to control the ICC. These functions are ICC clock setup, ICC voltage turning on/off, activation/deactivation, and ICC reset sequences. This driver doesn't support the ICC pin short circuit protection and an emergency deactivation.

## Macros

- #define **SMARTCARD\_ATR\_DURATION\_ADJUSTMENT** (360u)  
*Smart card define which specifies the adjustment number of clock cycles during which an ATR string has to be received.*
- #define **SMARTCARD\_INIT\_DELAY\_CLOCK\_CYCLES\_ADJUSTMENT** (4200u)  
*Smart card define which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.*

## Functions

- void **SMARTCARD\_PHY\_GPIO\_GetDefaultConfig** (smartcard\_interface\_config\_t \*config)  
*Fills in the configuration structure with default values.*
- status\_t **SMARTCARD\_PHY\_GPIO\_Init** (UART\_Type \*base, smartcard\_interface\_config\_t const \*config, uint32\_t srcClock\_Hz)  
*Initializes a Smart card interface instance.*
- void **SMARTCARD\_PHY\_GPIO\_Deinit** (UART\_Type \*base, smartcard\_interface\_config\_t \*config)  
*De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.*
- status\_t **SMARTCARD\_PHY\_GPIO\_Activate** (UART\_Type \*base, smartcard\_context\_t \*context, smartcard\_reset\_type\_t resetType)  
*Activates the Smart card IC.*
- status\_t **SMARTCARD\_PHY\_GPIO\_Deactivate** (UART\_Type \*base, smartcard\_context\_t \*context)  
*De-activates the Smart card IC.*
- status\_t **SMARTCARD\_PHY\_GPIO\_Control** (UART\_Type \*base, smartcard\_context\_t \*context, smartcard\_interface\_control\_t control, uint32\_t param)  
*Controls the Smart card interface IC.*

## 32.8.2 Macro Definition Documentation

32.8.2.1 `#define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)`

## 32.8.3 Function Documentation

32.8.3.1 `void SMARTCARD_PHY_GPIO_GetDefaultConfig ( smartcard_interface_config_t * config )`

## Smart Card PHY GPIO Driver

Parameters

<i>config</i>	The Smart card user configuration structure which contains configuration structure of type <a href="#">smartcard_interface_config_t</a> . Function fill in members: <code>clockToResetDelay</code> = 42000, <code>vcc</code> = <code>kSmartcardVoltageClassB3_3V</code> , with default values.
---------------	--

### 32.8.3.2 **status\_t SMARTCARD\_PHY\_GPIO\_Init ( UART\_Type \* *base*, smartcard\_interface\_config\_t const \* *config*, uint32\_t *srcClock\_Hz* )**

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>config</i>	The user configuration structure of type <a href="#">smartcard_interface_config_t</a> . Call to fill out configuration structure function <a href="#">SMARTCARD_PHY_GPIO_GetDefaultConfig()</a> .
<i>srcClock_Hz</i>	Smart card clock generation module source clock.

Return values

<i>kStatus_SMARTCARD_Success</i>	or <code>kStatus_SMARTCARD_OtherError</code> in case of error.
----------------------------------	--

### 32.8.3.3 **void SMARTCARD\_PHY\_GPIO\_Deinit ( UART\_Type \* *base*, smartcard\_interface\_config\_t \* *config* )**

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>config</i>	The user configuration structure of type <a href="#">smartcard_interface_config_t</a> .

### 32.8.3.4 **status\_t SMARTCARD\_PHY\_GPIO\_Activate ( UART\_Type \* *base*, smartcard\_context\_t \* *context*, smartcard\_reset\_type\_t *resetType* )**

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>resetType</i>	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Return values

<i>kStatus_SMARTCARD_-Success</i>	or kStatus_SMARTCARD_OtherError in case of error.
-----------------------------------	---

### 32.8.3.5 status\_t SMARTCARD\_PHY\_GPIO\_Deactivate ( **UART\_Type \* base**, **smartcard\_context\_t \* context** )

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.

Return values

<i>kStatus_SMARTCARD_-Success</i>	or kStatus_SMARTCARD_OtherError in case of error.
-----------------------------------	---

### 32.8.3.6 status\_t SMARTCARD\_PHY\_GPIO\_Control ( **UART\_Type \* base**, **smartcard\_context\_t \* context**, **smartcard\_interface\_control\_t control**, **uint32\_t param** )

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>control</i>	An interface command type.
<i>param</i>	Integer value specific to the control type.

Return values

## Smart Card PHY GPIO Driver

<i>kStatus_SMARTCARD_Success</i>	or kStatus_SMARTCARD_OtherError in case of error.
----------------------------------	---

## 32.9 Smart Card UART Driver

### 32.9.1 Overview

The Smart Card UART driver uses a standard UART peripheral which supports the ISO-7816 standard. The driver supports transmission functionality in the CPU mode. The driver also supports non-blocking (asynchronous) type of data transfers. The blocking (synchronous) transfer is supported only by the RTOS adaptation layer.

## Macros

- #define **SMARTCARD\_EMV\_RX\_NACK\_THRESHOLD** (5u)  
*EMV RX NACK interrupt generation threshold.*
- #define **SMARTCARD\_EMV\_TX\_NACK\_THRESHOLD** (4u)  
*EMV TX NACK interrupt generation threshold.*
- #define **SMARTCARD\_EMV\_RX\_TO\_TX\_GUARD\_TIME\_T0** (0x0u)  
*EMV TX & RX GUART TIME default value.*

## Functions

- void **SMARTCARD\_UART\_GetDefaultConfig** (smartcard\_card\_params\_t \*cardParams)  
*Fills in the smartcard\_card\_params structure with default values according to the EMV 4.3 specification.*
- status\_t **SMARTCARD\_UART\_Init** (UART\_Type \*base, smartcard\_context\_t \*context, uint32\_t srcClock\_Hz)  
*Initializes a UART peripheral for the Smart card/ISO-7816 operation.*
- void **SMARTCARD\_UART\_Deinit** (UART\_Type \*base)  
*This function disables the UART interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs) and gates UART clock in SIM.*
- int32\_t **SMARTCARD\_UART\_GetTransferRemainingBytes** (UART\_Type \*base, smartcard\_context\_t \*context)  
*Returns whether the previous UART transfer has finished.*
- status\_t **SMARTCARD\_UART\_AbortTransfer** (UART\_Type \*base, smartcard\_context\_t \*context)  
*Terminates an asynchronous UART transfer early.*
- status\_t **SMARTCARD\_UART\_TransferNonBlocking** (UART\_Type \*base, smartcard\_context\_t \*context, smartcard\_xfer\_t \*xfer)  
*Transfers data using interrupts.*
- status\_t **SMARTCARD\_UART\_Control** (UART\_Type \*base, smartcard\_context\_t \*context, smartcard\_control\_t control, uint32\_t param)  
*Controls the UART module per different user requests.*
- void **SMARTCARD\_UART\_IRQHandler** (UART\_Type \*base, smartcard\_context\_t \*context)  
*Interrupt handler for UART.*
- void **SMARTCARD\_UART\_ErrIRQHandler** (UART\_Type \*base, smartcard\_context\_t \*context)  
*Error interrupt handler for UART.*
- void **SMARTCARD\_UART\_TSExpiryCallback** (UART\_Type \*base, smartcard\_context\_t \*context)  
*Handles initial TS character timer time-out event.*

### 32.9.2 Function Documentation

32.9.2.1 `void SMARTCARD_UART_GetDefaultConfig ( smartcard_card_params_t *  
cardParams )`

Parameters

<i>cardParams</i>	The configuration structure of type <code>smartcard_interface_config_t</code> . Function fill in members: Fi = 372; Di = 1; currentD = 1; WI = 0x0A; GTN = 0x00; with default values.
-------------------	---

### 32.9.2.2 `status_t SMARTCARD_UART_Init ( UART_Type * base, smartcard_context_t * context, uint32_t srcClock_Hz )`

This function un-gates the UART clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core, and initializes the driver context.

Parameters

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a smart card driver context structure.
<i>srcClock_Hz</i>	Smart card clock generation module source clock.

Returns

An error code or `kStatus_SMARTCARD_Success`.

### 32.9.2.3 `void SMARTCARD_UART_Deinit ( UART_Type * base )`

Parameters

<i>base</i>	The UART peripheral base address.
-------------	-----------------------------------

### 32.9.2.4 `int32_t SMARTCARD_UART_GetTransferRemainingBytes ( UART_Type * base, smartcard_context_t * context )`

When performing an async transfer, call this function to ascertain the context of the current transfer: in progress (or busy) or complete (success). If the transfer is still in progress, the user can obtain the number of words that have not been transferred by reading `xSize` of smart card context structure.

Parameters

## Smart Card UART Driver

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.

Returns

The number of bytes not transferred.

### 32.9.2.5 status\_t SMARTCARD\_UART\_AbortTransfer ( **UART\_Type \* base,** **smartcard\_context\_t \* context** )

During an async UART transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.

Return values

<i>kStatus_SMARTCARD_-Success</i>	The transfer abort was successful.
<i>kStatus_SMARTCARD_-NoTransmitInProgress</i>	No transmission is currently in progress.

### 32.9.2.6 status\_t SMARTCARD\_UART\_TransferNonBlocking ( **UART\_Type \* base,** **smartcard\_context\_t \* context, smartcard\_xfer\_t \* xfer** )

A non-blocking (also known as asynchronous) function means that the function returns immediately after initiating the transfer function. The application has to get the transfer status to see when the transfer is complete. In other words, after calling non-blocking (asynchronous) transfer function, the application must get the transfer status to check if transmit is completed or not.

Parameters

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.

<i>xfer</i>	A pointer to Smart card transfer structure where the linked buffers and sizes are stored.
-------------	---

Returns

An error code or kStatus\_SMARTCARD\_Success.

### 32.9.2.7 **status\_t SMARTCARD\_UART\_Control ( UART\_Type \* *base*, smartcard\_context\_t \* *context*, smartcard\_control\_t *control*, uint32\_t *param* )**

Parameters

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a smart card driver context structure.
<i>control</i>	Smart card command type.
<i>param</i>	Integer value specific to a control command.

return An kStatus\_SMARTCARD\_OtherError in case of error return kStatus\_SMARTCARD\_Success in success

### 32.9.2.8 **void SMARTCARD\_UART\_IRQHandler ( UART\_Type \* *base*, smartcard\_context\_t \* *context* )**

This handler uses the buffers stored in the [smartcard\\_context\\_t](#) structures to transfer data. The Smart card driver requires this function to call when the UART interrupt occurs.

Parameters

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.

### 32.9.2.9 **void SMARTCARD\_UART\_ErrIRQHandler ( UART\_Type \* *base*, smartcard\_context\_t \* *context* )**

This function handles error conditions during a transfer.

Parameters

## Smart Card UART Driver

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.

**32.9.2.10 void SMARTCARD\_UART\_TSExpiryCallback ( **UART\_Type** \* *base*,  
smartcard\_context\_t \* *context* )**

Parameters

<i>base</i>	The UART peripheral base address.
<i>context</i>	A pointer to a Smart card driver context structure.

## 32.10 Smart Card EMVSIM Driver

### 32.10.1 Overview

The SmartCard EMVSIM driver covers the transmission functionality in the CPU mode. The driver supports non-blocking (asynchronous) type of data transfers. The blocking (synchronous) transfer is supported only by the RTOS adaptation layer.

### Macros

- #define **SMARTCARD\_EMV\_RX\_NACK\_THRESHOLD** (5u)  
*EMV RX NACK interrupt generation threshold.*
- #define **SMARTCARD\_EMV\_TX\_NACK\_THRESHOLD** (5u)  
*EMV TX NACK interrupt generation threshold.*
- #define **SMARTCARD\_WWT\_ADJUSTMENT** (160u)  
*Smart card Word Wait Timer adjustment value.*
- #define **SMARTCARD\_CWT\_ADJUSTMENT** (3u)  
*Smart card Character Wait Timer adjustment value.*

### Enumerations

- enum **emvsim\_gpc\_clock\_select\_t** {
   
    kEMVSIM\_GPCClockDisable = 0u,
   
    kEMVSIM\_GPCCardClock = 1u,
   
    kEMVSIM\_GPCRxClock = 2u,
   
    kEMVSIM\_GPCTxClock = 3u }
   
*General Purpose Counter clock selections.*
- enum **emvsim\_presence\_detect\_edge\_t** {
   
    kEMVSIM\_DetectOnFallingEdge = 0u,
   
    kEMVSIM\_DetectOnRisingEdge = 1u }
   
*EMVSIM card presence detection edge control.*
- enum **emvsim\_presence\_detect\_status\_t** {
   
    kEMVSIM\_DetectPinIsLow = 0u,
   
    kEMVSIM\_DetectPinIsHigh = 1u }
   
*EMVSIM card presence detection status.*

### Smart card EMVSIM Driver

- void **SMARTCARD\_EMVSIM\_GetDefaultConfig** (smartcard\_card\_params\_t \*cardParams)
   
*Fills in the smartcard\_card\_params structure with default values according to the EMV 4.3 specification.*
- status\_t **SMARTCARD\_EMVSIM\_Init** (EMVSIM\_Type \*base, smartcard\_context\_t \*context, uint32\_t srcClock\_Hz)
   
*Initializes an EMVSIM peripheral for the Smart card/ISO-7816 operation.*
- void **SMARTCARD\_EMVSIM\_Deinit** (EMVSIM\_Type \*base)

## Smart Card EMVSIM Driver

*This function disables the EMVSIM interrupts, disables the transmitter and receiver, flushes the FIFOs, and gates EMVSIM clock in SIM.*

- int32\_t **SMARTCARD\_EMVSIM\_GetTransferRemainingBytes** (EMVSIM\_Type \*base, smartcard\_context\_t \*context)  
*Returns whether the previous EMVSIM transfer has finished.*
- status\_t **SMARTCARD\_EMVSIM\_AbortTransfer** (EMVSIM\_Type \*base, smartcard\_context\_t \*context)  
*Terminates an asynchronous EMVSIM transfer early.*
- status\_t **SMARTCARD\_EMVSIM\_TransferNonBlocking** (EMVSIM\_Type \*base, smartcard\_context\_t \*context, smartcard\_xfer\_t \*xfer)  
*Transfer data using interrupts.*
- status\_t **SMARTCARD\_EMVSIM\_Control** (EMVSIM\_Type \*base, smartcard\_context\_t \*context, smartcard\_control\_t control, uint32\_t param)  
*Controls the EMVSIM module per different user request.*
- void **SMARTCARD\_EMVSIM\_IRQHandler** (EMVSIM\_Type \*base, smartcard\_context\_t \*context)  
*Handles EMVSIM module interrupts.*

### 32.10.2 Enumeration Type Documentation

#### 32.10.2.1 enum emvsim\_gpc\_clock\_select\_t

Enumerator

**kEMVSIM\_GPCClockDisable** Disabled.

**kEMVSIM\_GPCCardClock** Card clock.

**kEMVSIM\_GPCRxClock** Receive clock.

**kEMVSIM\_GPCTxClock** Transmit ETU clock.

#### 32.10.2.2 enum emvsim\_presence\_detect\_edge\_t

Enumerator

**kEMVSIM\_DetectOnFallingEdge** Presence detected on the falling edge.

**kEMVSIM\_DetectOnRisingEdge** Presence detected on the rising edge.

#### 32.10.2.3 enum emvsim\_presence\_detect\_status\_t

Enumerator

**kEMVSIM\_DetectPinIsLow** Presence detected pin is logic low.

**kEMVSIM\_DetectPinIsHigh** Presence detected pin is logic high.

### 32.10.3 Function Documentation

32.10.3.1 **void SMARTCARD\_EMVSIM\_GetDefaultConfig ( smartcard\_card\_params\_t \*  
*cardParams* )**

## Smart Card EMVSIM Driver

Parameters

<i>cardParams</i>	The configuration structure of type <a href="#">smartcard_interface_config_t</a> . Function fill in members: Fi = 372; Di = 1; currentD = 1; WI = 0x0A; GTN = 0x00; with default values.
-------------------	--

### 32.10.3.2 **status\_t SMARTCARD\_EMVSIM\_Init ( *EMVSIM\_Type* \* *base*, *smartcard\_context\_t* \* *context*, *uint32\_t srcClock\_Hz* )**

This function un-gates the EMVSIM clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core and, initializes the driver context.

Parameters

<i>base</i>	The EMVSIM peripheral base address.
<i>context</i>	A pointer to the smart card driver context structure.
<i>srcClock_Hz</i>	Smart card clock generation module source clock.

Returns

An error code or kStatus\_SMARTCARD\_Success.

### 32.10.3.3 **void SMARTCARD\_EMVSIM\_Deinit ( *EMVSIM\_Type* \* *base* )**

Parameters

<i>base</i>	The EMVSIM module base address.
-------------	---------------------------------

### 32.10.3.4 **int32\_t SMARTCARD\_EMVSIM\_GetTransferRemainingBytes ( *EMVSIM\_Type* \* *base*, *smartcard\_context\_t* \* *context* )**

When performing an async transfer, call this function to ascertain the context of the current transfer: in progress (or busy) or complete (success). If the transfer is still in progress, the user can obtain the number of words that have not been transferred.

Parameters

<i>base</i>	The EMVSIM module base address.
<i>context</i>	A pointer to a smart card driver context structure.

Returns

The number of bytes not transferred.

### 32.10.3.5 status\_t SMARTCARD\_EMVSIM\_AbortTransfer ( EMVSIM\_Type \* *base*, smartcard\_context\_t \* *context* )

During an async EMVSIM transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

<i>base</i>	The EMVSIM peripheral address.
<i>context</i>	A pointer to a smart card driver context structure.

Return values

<i>kStatus_SMARTCARD_Success</i>	The transmit abort was successful.
<i>kStatus_SMARTCARD_NoTransmitInProgress</i>	No transmission is currently in progress.

### 32.10.3.6 status\_t SMARTCARD\_EMVSIM\_TransferNonBlocking ( EMVSIM\_Type \* *base*, smartcard\_context\_t \* *context*, smartcard\_xfer\_t \* *xfer* )

A non-blocking (also known as asynchronous) function means that the function returns immediately after initiating the transfer function. The application has to get the transfer status to see when the transfer is complete. In other words, after calling the non-blocking (asynchronous) transfer function, the application must get the transfer status to check if the transmit is completed or not.

Parameters

<i>base</i>	The EMVSIM peripheral base address.
-------------	-------------------------------------

## Smart Card EMVSIM Driver

<i>context</i>	A pointer to a smart card driver context structure.
<i>xfer</i>	A pointer to the smart card transfer structure where the linked buffers and sizes are stored.

Returns

An error code or kStatus\_SMARTCARD\_Success.

### 32.10.3.7 **status\_t SMARTCARD\_EMVSIM\_Control ( *EMVSIM\_Type* \* *base*, *smartcard\_context\_t* \* *context*, *smartcard\_control\_t* *control*, *uint32\_t* *param* )**

Parameters

<i>base</i>	The EMVSIM peripheral base address.
<i>context</i>	A pointer to a smart card driver context structure.
<i>control</i>	Control type.
<i>param</i>	Integer value of specific to control command.

return kStatus\_SMARTCARD\_Success in success. return kStatus\_SMARTCARD\_OtherError in case of error.

### 32.10.3.8 **void SMARTCARD\_EMVSIM\_IRQHandler ( *EMVSIM\_Type* \* *base*, *smartcard\_context\_t* \* *context* )**

Parameters

<i>base</i>	The EMVSIM peripheral base address.
<i>context</i>	A pointer to a smart card driver context structure.

## 32.11 Smart Card FreeRTOS Driver

### 32.11.1 Overview

#### Data Structures

- struct `rtos_smartcard_context_t`  
*Runtime RTOS Smart card driver context.* [More...](#)

#### Macros

- #define `RTOS_SMARTCARD_COMPLETE` 0x1u  
*Smart card RTOS transfer complete flag.*
- #define `RTOS_SMARTCARD_TIMEOUT` 0x2u  
*Smart card RTOS transfer time-out flag.*
- #define `SMARTCARD_Control`(base, context, control, param) `SMARTCARD_UART_Control`(base, context, control, 0)  
*Common Smart card driver API defines.*
- #define `SMARTCARD_Transfer`(base, context, xfer) `SMARTCARD_UART_TransferNonBlocking`(base, context, xfer)  
*Common Smart card API macro.*
- #define `SMARTCARD_AbortTransfer`(base, context) `SMARTCARD_UART_AbortTransfer`(base, context)  
*Common Smart card API macro.*
- #define `SMARTCARD_Init`(base, context, sourceClockHz) `SMARTCARD_UART_Init`(base, context, sourceClockHz)  
*Common Smart card API macro.*
- #define `SMARTCARD_Deinit`(base) `SMARTCARD_UART_Deinit`(base)  
*Common Smart card API macro.*
- #define `SMARTCARD_GetTransferRemainingBytes`(base, context) `SMARTCARD_UART_GetTransferRemainingBytes`(base, context)  
*Common Smart card API macro.*
- #define `SMARTCARD_GetDefaultConfig`(cardParams) `SMARTCARD_UART_GetDefaultConfig`(cardParams)  
*Common Smart card API macro.*
- #define `SMARTCARD_PHY_Activate`(base, context, resetType) `SMARTCARD_PHY_GPIO_Activate`(base, context, resetType)  
*Common Smart card API macro.*
- #define `SMARTCARD_PHY_Deactivate`(base, context) `SMARTCARD_PHY_GPIO_Deactivate`(base, context)  
*Common Smart card API macro.*
- #define `SMARTCARD_PHY_Control`(base, context, control, param) `SMARTCARD_PHY_GPIO_Control`(base, context, control, param)  
*Common Smart card API macro.*
- #define `SMARTCARD_PHY_Init`(base, config, sourceClockHz) `SMARTCARD_PHY_GPIO_Init`(base, config, sourceClockHz)  
*Common Smart card API macro.*

## Smart Card FreeRTOS Driver

- #define **SMARTCARD\_PHY\_Deinit**(base, config) **SMARTCARD\_PHY\_GPIO\_Deinit**(base, config)  
*Common Smart card API macro.*
- #define **SMARTCARD\_PHY\_GetDefaultConfig**(config) **SMARTCARD\_PHY\_GPIO\_GetDefaultConfig**(config)  
*Common Smart card API macro.*

## Functions

- int **SMARTCARD\_RTOS\_Init** (void \*base, **rtos\_smartcard\_context\_t** \*ctx, uint32\_t sourceClock-Hz)  
*Initializes a Smart card (EMVSIM/UART) peripheral for Smart card/ISO-7816 operation.*
- int **SMARTCARD\_RTOS\_Deinit** (**rtos\_smartcard\_context\_t** \*ctx)  
*This function disables the Smart card (EMVSIM/UART) interrupts, disables the transmitter and receiver, and flushes the FIFOs (for modules that support FIFOs) and gates Smart card clock in SIM.*
- int **SMARTCARD\_RTOS\_Transfer** (**rtos\_smartcard\_context\_t** \*ctx, **smartcard\_xfer\_t** \*xfer)  
*Transfers data using interrupts.*
- int **SMARTCARD\_RTOS\_AbortTransfer** (**rtos\_smartcard\_context\_t** \*ctx)  
*Terminates an asynchronous EMVSIM transfer early.*
- int **SMARTCARD\_RTOS\_WaitForXevent** (**rtos\_smartcard\_context\_t** \*ctx)  
*Waits until the transfer is finished.*
- int **SMARTCARD\_RTOS\_Control** (**rtos\_smartcard\_context\_t** \*ctx, **smartcard\_control\_t** control, uint32\_t param)  
*Controls the Smart card module per different user requests.*
- int **SMARTCARD\_RTOS\_PHY\_Control** (**rtos\_smartcard\_context\_t** \*ctx, **smartcard\_interface-control\_t** control, uint32\_t param)  
*Controls the Smart card module as per different user request.*
- int **SMARTCARD\_RTOS\_PHY\_Activate** (**rtos\_smartcard\_context\_t** \*ctx, **smartcard\_reset\_type\_t** resetType)  
*Activates the Smart card interface.*
- int **SMARTCARD\_RTOS\_PHY\_Deactivate** (**rtos\_smartcard\_context\_t** \*ctx)  
*Deactivates the Smart card interface.*

### 32.11.2 Data Structure Documentation

#### 32.11.2.1 struct **rtos\_smartcard\_context\_t**

##### Data Fields

- **SemaphoreHandle\_t x\_sem**  
*RTOS unique access assurance object.*
- **SemaphoreHandle\_t x\_event**  
*RTOS synchronization object.*
- **smartcard\_context\_t x\_context**  
*transactional layer state*

### 32.11.3 Macro Definition Documentation

**32.11.3.1 #define SMARTCARD\_Control( *base*, *context*, *control*, *param* ) SMARTCARD\_UART\_Control(*base*, *context*, *control*, 0)**

Common Smart card API macro

### 32.11.4 Function Documentation

**32.11.4.1 int SMARTCARD\_RTOS\_Init ( void \* *base*, rtos\_smartcard\_context\_t \* *ctx*, uint32\_t *sourceClockHz* )**

Also initialize Smart card PHY interface.

This function ungates the Smart card clock, initializes the module to EMV default settings, configures the IRQ state structure, and enables the module-level interrupt to the core. Initializes RTOS synchronization objects and context.

Parameters

<i>base</i>	The Smart card peripheral base address.
<i>ctx</i>	The Smart card RTOS structure.
<i>sourceClockHz</i>	Smart card clock generation module source clock.

Returns

A zero in success or error code.

**32.11.4.2 int SMARTCARD\_RTOS\_Deinit ( rtos\_smartcard\_context\_t \* *ctx* )**

It also deactivates Smart card PHY interface, stops Smart card clocks, and frees all synchronization objects allocated in the RTOS Smart card context.

Parameters

<i>ctx</i>	The Smart card RTOS state.
------------	----------------------------

Returns

A zero in success or error code.

### 32.11.4.3 int SMARTCARD\_RTOs\_Transfer ( rtos\_smartcard\_context\_t \* *ctx*, smartcard\_xfer\_t \* *xfer* )

A blocking (also known as synchronous) function means that the function returns after the transfer is done. User can cancel this transfer by calling the function AbortTransfer.

Parameters

<i>ctx</i>	A pointer to the RTOS Smart card driver context.
<i>xfer</i>	Smart card transfer structure.

Returns

A zero in success or error code.

#### **32.11.4.4 int SMARTCARD\_RTOs\_AbortTransfer ( rtos\_smartcard\_context\_t \* ctx )**

During an async EMVSIM transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

<i>ctx</i>	A pointer to the RTOS Smart card driver context.
------------	--

Returns

A zero in success or error code.

#### **32.11.4.5 int SMARTCARD\_RTOs\_WaitForXevent ( rtos\_smartcard\_context\_t \* ctx )**

Task waits on a transfer finish event. Don't initialize the transfer. Instead, wait for a transfer callback. This function can be used while waiting on an initial TS character.

Parameters

<i>ctx</i>	A pointer to the RTOS Smart card driver context.
------------	--

Returns

A zero in success or error code.

#### **32.11.4.6 int SMARTCARD\_RTOs\_Control ( rtos\_smartcard\_context\_t \* ctx, smartcard\_control\_t control, uint32\_t param )**

## Smart Card FreeRTOS Driver

Parameters

<i>ctx</i>	The Smart card RTOS context pointer.
<i>control</i>	Control type.
<i>param</i>	Integer value to control the command.

Returns

A zero in success or error code.

### 32.11.4.7 int SMARTCARD\_RTOS\_PHY\_Control ( *rtos\_smartcard\_context\_t \* ctx*, *smartcard\_interface\_control\_t control*, *uint32\_t param* )

Parameters

<i>ctx</i>	The Smart card RTOS context pointer.
<i>control</i>	Control type
<i>param</i>	Integer value to control the command.

Returns

A zero in success or error code.

### 32.11.4.8 int SMARTCARD\_RTOS\_PHY\_Activate ( *rtos\_smartcard\_context\_t \* ctx*, *smartcard\_reset\_type\_t resetType* )

Parameters

<i>ctx</i>	The Smart card RTOS driver context structure.
<i>resetType</i>	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcardWarmReset

Returns

A zero in success or error code.

### 32.11.4.9 int SMARTCARD\_RTOS\_PHY\_Deactivate ( *rtos\_smartcard\_context\_t \* ctx* )

Parameters

<i>ctx</i>	The Smart card RTOS driver context structure.
------------	---

Returns

A zero in success or error code.



# Chapter 33

## SMC: System Mode Controller Driver

### 33.1 Overview

The MCUXpresso SDK provides a peripheral driver for the System Mode Controller (SMC) module of MCUXpresso SDK devices. The SMC module sequences the system in and out of all low-power stop and run modes.

API functions are provided to configure the system for working in a dedicated power mode. For different power modes, SMC\_SetPowerModeXXX() function accepts different parameters. System power mode state transitions are not available between power modes. For details about available transitions, see the power mode transitions section in the SoC reference manual.

### 33.2 Typical use case

#### 33.2.1 Enter wait or stop modes

SMC driver provides APIs to set MCU to different wait modes and stop modes. Pre and post functions are used for setting the modes. The pre functions and post functions are used as follows.

1. Disable/enable the interrupt through PRIMASK. This is an example use case. The application sets the wakeup interrupt and calls SMC function `SMC_SetPowerModeStop` to set the MCU to STOP mode, but the wakeup interrupt happens so quickly that the ISR completes before the function `SMC_SetPowerModeStop`. As a result, the MCU enters the STOP mode and never is woken up by the interrupt. In this use case, the application first disables the interrupt through PRIMASK, sets the wakeup interrupt, and enters the STOP mode. After wakeup, enable the interrupt through PRIMASK. The MCU can still be woken up by disabling the interrupt through PRIMASK. The pre and post functions handle the PRIMASK.
2. Disable/enable the flash speculation. When entering stop modes, the flash speculation might be interrupted. As a result, pre functions disable the flash speculation and post functions enable it.

```
SMC_PreEnterStopModes();  
/* Enable the wakeup interrupt here. */  
SMC_SetPowerModeStop(SMC, kSMC_PartialStop);  
SMC_PostExitStopModes();
```

### Data Structures

- struct `smc_power_mode_vlls_config_t`  
*SMC Very Low-Leakage Stop power mode configuration.* [More...](#)

## Typical use case

### Enumerations

- enum `smc_power_mode_protection_t` {  
    `kSMC_AllowPowerModeVlls` = `SMC_PMPROT_AVLLS_MASK`,  
    `kSMC_AllowPowerModeLls` = `SMC_PMPROT_ALLS_MASK`,  
    `kSMC_AllowPowerModeVlp` = `SMC_PMPROT_AVLP_MASK`,  
    `kSMC_AllowPowerModeAll` }  
        *Power Modes Protection.*
- enum `smc_power_state_t` {  
    `kSMC_PowerStateRun` = `0x01U << 0U`,  
    `kSMC_PowerStateStop` = `0x01U << 1U`,  
    `kSMC_PowerStateVlpr` = `0x01U << 2U`,  
    `kSMC_PowerStateVlpw` = `0x01U << 3U`,  
    `kSMC_PowerStateVlps` = `0x01U << 4U`,  
    `kSMC_PowerStateLls` = `0x01U << 5U`,  
    `kSMC_PowerStateVlls` = `0x01U << 6U` }  
        *Power Modes in PMSTAT.*
- enum `smc_run_mode_t` {  
    `kSMC_RunNormal` = `0U`,  
    `kSMC_RunVlpr` = `2U` }  
        *Run mode definition.*
- enum `smc_stop_mode_t` {  
    `kSMC_StopNormal` = `0U`,  
    `kSMC_StopVlps` = `2U`,  
    `kSMC_StopLls` = `3U`,  
    `kSMC_StopVlls` = `4U` }  
        *Stop mode definition.*
- enum `smc_stop_submode_t` {  
    `kSMC_StopSub0` = `0U`,  
    `kSMC_StopSub1` = `1U`,  
    `kSMC_StopSub2` = `2U`,  
    `kSMC_StopSub3` = `3U` }  
        *VLLS/LLS stop sub mode definition.*
- enum `smc_partial_stop_option_t` {  
    `kSMC_PartialStop` = `0U`,  
    `kSMC_PartialStop1` = `1U`,  
    `kSMC_PartialStop2` = `2U` }  
        *Partial STOP option.*
- enum `_smc_status` { `kStatus_SMC_StopAbort` = `MAKE_STATUS(kStatusGroup_POWER, 0)` }  
        *SMC configuration status.*

### Driver version

- #define `FSL_SMC_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 3)`)  
    *SMC driver version 2.0.3.*

## System mode controller APIs

- static void [SMC\\_SetPowerModeProtection](#) (SMC\_Type \*base, uint8\_t allowedModes)  
*Configures all power mode protection settings.*
- static smc\_power\_state\_t [SMC\\_GetPowerModeState](#) (SMC\_Type \*base)  
*Gets the current power mode status.*
- void [SMC\\_PreEnterStopModes](#) (void)  
*Prepares to enter stop modes.*
- void [SMC\\_PostExitStopModes](#) (void)  
*Recoveries after wake up from stop modes.*
- static void [SMC\\_PreEnterWaitModes](#) (void)  
*Prepares to enter wait modes.*
- static void [SMC\\_PostExitWaitModes](#) (void)  
*Recoveries after wake up from stop modes.*
- status\_t [SMC\\_SetPowerModeRun](#) (SMC\_Type \*base)  
*Configures the system to RUN power mode.*
- status\_t [SMC\\_SetPowerModeWait](#) (SMC\_Type \*base)  
*Configures the system to WAIT power mode.*
- status\_t [SMC\\_SetPowerModeStop](#) (SMC\_Type \*base, [smc\\_partial\\_stop\\_option\\_t](#) option)  
*Configures the system to Stop power mode.*
- status\_t [SMC\\_SetPowerModeVlpr](#) (SMC\_Type \*base, bool wakeupMode)  
*Configures the system to VLPR power mode.*
- status\_t [SMC\\_SetPowerModeVlpw](#) (SMC\_Type \*base)  
*Configures the system to VLPW power mode.*
- status\_t [SMC\\_SetPowerModeVlps](#) (SMC\_Type \*base)  
*Configures the system to VLPS power mode.*
- status\_t [SMC\\_SetPowerModeLls](#) (SMC\_Type \*base)  
*Configures the system to LLS power mode.*
- status\_t [SMC\\_SetPowerModeVlls](#) (SMC\_Type \*base, const [smc\\_power\\_mode\\_vlls\\_config\\_t](#) \*config)  
*Configures the system to VLLS power mode.*

## 33.3 Data Structure Documentation

### 33.3.1 struct smc\_power\_mode\_vlls\_config\_t

#### Data Fields

- [smc\\_stop\\_submode\\_t](#) subMode  
*Very Low-leakage Stop sub-mode.*
- bool [enablePorDetectInVlls0](#)  
*Enable Power on reset detect in VLLS mode.*

## 33.4 Macro Definition Documentation

### 33.4.1 #define FSL\_SMC\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 3))

## Enumeration Type Documentation

### 33.5 Enumeration Type Documentation

#### 33.5.1 enum smc\_power\_mode\_protection\_t

Enumerator

*kSMC\_AllowPowerModeVlls* Allow Very-low-leakage Stop Mode.

*kSMC\_AllowPowerModeLls* Allow Low-leakage Stop Mode.

*kSMC\_AllowPowerModeVlp* Allow Very-Low-power Mode.

*kSMC\_AllowPowerModeAll* Allow all power mode.

#### 33.5.2 enum smc\_power\_state\_t

Enumerator

*kSMC\_PowerStateRun* 0000\_0001 - Current power mode is RUN

*kSMC\_PowerStateStop* 0000\_0010 - Current power mode is STOP

*kSMC\_PowerStateVlpr* 0000\_0100 - Current power mode is VLPR

*kSMC\_PowerStateVlpw* 0000\_1000 - Current power mode is VLPW

*kSMC\_PowerStateVlps* 0001\_0000 - Current power mode is VLPS

*kSMC\_PowerStateLls* 0010\_0000 - Current power mode is LLS

*kSMC\_PowerStateVlls* 0100\_0000 - Current power mode is VLLS

#### 33.5.3 enum smc\_run\_mode\_t

Enumerator

*kSMC\_RunNormal* Normal RUN mode.

*kSMC\_RunVlpr* Very-low-power RUN mode.

#### 33.5.4 enum smc\_stop\_mode\_t

Enumerator

*kSMC\_StopNormal* Normal STOP mode.

*kSMC\_StopVlps* Very-low-power STOP mode.

*kSMC\_StopLls* Low-leakage Stop mode.

*kSMC\_StopVlls* Very-low-leakage Stop mode.

### 33.5.5 enum smc\_stop\_submode\_t

Enumerator

- kSMC\_StopSub0* Stop submode 0, for VLLS0/LLS0.
- kSMC\_StopSub1* Stop submode 1, for VLLS1/LLS1.
- kSMC\_StopSub2* Stop submode 2, for VLLS2/LLS2.
- kSMC\_StopSub3* Stop submode 3, for VLLS3/LLS3.

### 33.5.6 enum smc\_partial\_stop\_option\_t

Enumerator

- kSMC\_PartialStop* STOP - Normal Stop mode.
- kSMC\_PartialStop1* Partial Stop with both system and bus clocks disabled.
- kSMC\_PartialStop2* Partial Stop with system clock disabled and bus clock enabled.

### 33.5.7 enum \_smc\_status

Enumerator

- kStatus\_SMC\_StopAbort* Entering Stop mode is abort.

## 33.6 Function Documentation

### 33.6.1 static void SMC\_SetPowerModeProtection ( **SMC\_Type** \* *base*, **uint8\_t allowedModes** ) [inline], [static]

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the `smc_power_mode_protection_t`. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map. For example, to allow LLS and VLLS, use `SMC_SetPowerModeProtection(kSMC_AllowPowerModeVlls | kSMC_AllowPowerModeVlps)`. To allow all modes, use `SMC_SetPowerModeProtection(kSMC_AllowPowerModeAll)`.

Parameters

---

## Function Documentation

<i>base</i>	SMC peripheral base address.
<i>allowedModes</i>	Bitmap of the allowed power modes.

### **33.6.2 static smc\_power\_state\_t SMC\_GetPowerModeState ( SMC\_Type \* *base* ) [inline], [static]**

This function returns the current power mode status. After the application switches the power mode, it should always check the status to check whether it runs into the specified mode or not. The application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the smc\_power\_state\_t for information about the power status.

Parameters

<i>base</i>	SMC peripheral base address.
-------------	------------------------------

Returns

Current power mode status.

### **33.6.3 void SMC\_PreEnterStopModes ( void )**

This function should be called before entering STOP/VLPS/LLS/VLLS modes.

### **33.6.4 void SMC\_PostExitStopModes ( void )**

This function should be called after wake up from STOP/VLPS/LLS/VLLS modes. It is used with [SMC\\_PreEnterStopModes](#).

### **33.6.5 static void SMC\_PreEnterWaitModes ( void ) [inline], [static]**

This function should be called before entering WAIT/VLPW modes.

### **33.6.6 static void SMC\_PostExitWaitModes ( void ) [inline], [static]**

This function should be called after wake up from WAIT/VLPW modes. It is used with [SMC\\_PreEnterWaitModes](#).

33.6.7 **status\_t SMC\_SetPowerModeRun ( SMC\_Type \* *base* )**

## Function Documentation

Parameters

<i>base</i>	SMC peripheral base address.
-------------	------------------------------

Returns

SMC configuration error code.

### 33.6.8 **status\_t SMC\_SetPowerModeWait ( SMC\_Type \* *base* )**

Parameters

<i>base</i>	SMC peripheral base address.
-------------	------------------------------

Returns

SMC configuration error code.

### 33.6.9 **status\_t SMC\_SetPowerModeStop ( SMC\_Type \* *base*, smc\_partial\_stop\_option\_t *option* )**

Parameters

<i>base</i>	SMC peripheral base address.
<i>option</i>	Partial Stop mode option.

Returns

SMC configuration error code.

### 33.6.10 **status\_t SMC\_SetPowerModeVlpr ( SMC\_Type \* *base*, bool *wakeupMode* )**

Parameters

<i>base</i>	SMC peripheral base address.
<i>wakeupMode</i>	Enter Normal Run mode if true, else stay in VLPR mode.

Returns

SMC configuration error code.

### 33.6.11 status\_t SMC\_SetPowerModeVlpw ( SMC\_Type \* *base* )

Parameters

<i>base</i>	SMC peripheral base address.
-------------	------------------------------

Returns

SMC configuration error code.

### 33.6.12 status\_t SMC\_SetPowerModeVlps ( SMC\_Type \* *base* )

Parameters

<i>base</i>	SMC peripheral base address.
-------------	------------------------------

Returns

SMC configuration error code.

### 33.6.13 status\_t SMC\_SetPowerModeLls ( SMC\_Type \* *base* )

Parameters

<i>base</i>	SMC peripheral base address.
-------------	------------------------------

Returns

SMC configuration error code.

### 33.6.14 status\_t SMC\_SetPowerModeVlls ( SMC\_Type \* *base*, const smc\_power\_mode\_vlls\_config\_t \* *config* )

## Function Documentation

Parameters

<i>base</i>	SMC peripheral base address.
<i>config</i>	The VLLS power mode configuration structure.

Returns

SMC configuration error code.

## Chapter 34

# UART: Universal Asynchronous Receiver/Transmitter Driver

### 34.1 Overview

#### Modules

- [UART DMA Driver](#)
- [UART Driver](#)
- [UART FreeRTOS Driver](#)
- [UART eDMA Driver](#)

### 34.2 UART Driver

#### 34.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) module of MCUXpresso SDK devices.

The UART driver includes functional APIs and transactional APIs.

Functional APIs are used for UART initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the UART peripheral and how to organize functional APIs to meet the application requirements. All functional APIs use the peripheral base address as the first parameter. UART functional operation groups provide the functional API set.

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the `uart_handle_t` as the second parameter. Initialize the handle by calling the [UART\\_TransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer, which means that the functions [UART\\_TransferSendNonBlocking\(\)](#) and [UART\\_TransferReceiveNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_UART_TxIdle` and `kStatus_UART_RxIdle`.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the [UART\\_TransferCreateHandle\(\)](#). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The [UART\\_TransferReceiveNonBlocking\(\)](#) function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the `kStatus_UART_RxIdle`.

If the receive ring buffer is full, the upper layer is informed through a callback with the `kStatus_UART_RxRingBufferOverrun`. In the callback function, the upper layer reads data out from the ring buffer. If not, existing data is overwritten by the new data.

The ring buffer size is specified when creating the handle. Note that one byte is reserved for the ring buffer maintenance. When creating handle using the following code.

```
UART_TransferCreateHandle(UART0, &handle, UART_UserCallback, NULL);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

#### 34.2.2 Typical use case

##### 34.2.2.1 UART Send/receive using a polling method

```
uint8_t ch;
```

```

UART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableTx = true;
user_config.enableRx = true;

UART_Init(UART1, &user_config, 120000000U);

while(1)
{
    UART_ReadBlocking(UART1, &ch, 1);
    UART_WriteBlocking(UART1, &ch, 1);
}

```

### 34.2.2.2 UART Send/receive using an interrupt method

```

uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_TxIdle == status)
    {
        txFinished = true;
    }

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_TransferCreateHandle(UART1, &g_uartHandle, UART_UserCallback, NULL);

    // Prepare to send.
    sendXfer.data = sendData
    sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
    txFinished = false;

    // Send out.
    UART_TransferSendNonBlocking(&g_uartHandle, &g_uartHandle, &sendXfer);

    // Wait send finished.
    while (!txFinished)
    {

    }

    // Prepare to receive.

```

## UART Driver

```
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
rxFinished = false;

// Receive.
UART_TransferReceiveNonBlocking(&g_uartHandle, &g_uartHandle, &
    receiveXfer);

// Wait receive finished.
while (!rxFinished)
{
}

// ...
}
```

### 34.2.2.3 UART Receive using the ringbuffer feature

```
#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE      32

uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    size_t bytesRead;
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_TransferCreateHandle(UART1, &g_uartHandle, UART_UserCallback, NULL);

    // Now the RX is working in background, receive in to ring buffer.

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = RX_DATA_SIZE;
    rxFinished = false;

    // Receive.
    UART_TransferReceiveNonBlocking(UART1, &g_uartHandle, &receiveXfer);

    if (bytesRead = RX_DATA_SIZE) /* Have read enough data. */
    {

```

```

        ;
    }
    else
    {
        if (bytesRead) /* Received some data, process first. */
        {
            ;
        }

        // Wait receive finished.
        while (!rxFinished)
        {
        }
    }

    // ...
}

```

### 34.2.2.4 UART Send/Receive using the DMA method

```

uart_handle_t g_uartHandle;
dma_handle_t g_uartTxDmaHandle;
dma_handle_t g_uartRxDmaHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_TxIdle == status)
    {
        txFinished = true;
    }

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);

    // Set up the DMA
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, UART_TX_DMA_CHANNEL, UART_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_TX_DMA_CHANNEL);
    DMAMUX_SetSource(DMAMUX0, UART_RX_DMA_CHANNEL, UART_RX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_RX_DMA_CHANNEL);

    DMA_Init(DMA0);
}

```

## UART Driver

```
/* Create DMA handle. */
DMA_CreateHandle(&g_uartTxHandle, DMA0, UART_TX_DMA_CHANNEL);
DMA_CreateHandle(&g_uartRxHandle, DMA0, UART_RX_DMA_CHANNEL);

UART_TransferCreateHandleDMA(UART1, &g_uartHandle, UART_UserCallback, NULL,
    &g_uartTxHandle, &g_uartRxHandle);

// Prepare to send.
sendXfer.data = sendData;
sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
txFinished = false;

// Send out.
UART_TransferSendDMA(UART1, &g_uartHandle, &sendXfer);

// Wait send finished.
while (!txFinished)
{
}

// Prepare to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
rxFinished = false;

// Receive.
UART_TransferReceiveDMA(UART1, &g_uartHandle, &receiveXfer);

// Wait receive finished.
while (!rxFinished)
{
}

// ...
}
```

## Data Structures

- struct [uart\\_config\\_t](#)  
*UART configuration structure.* [More...](#)
- struct [uart\\_transfer\\_t](#)  
*UART transfer structure.* [More...](#)
- struct [uart\\_handle\\_t](#)  
*UART handle structure.* [More...](#)

## Typedefs

- [typedef void\(\\* uart\\_transfer\\_callback\\_t \)](#)(UART\_Type \*base, [uart\\_handle\\_t](#) \*handle, [status\\_t](#) status, void \*userData)  
*UART transfer callback function.*

## Enumerations

- enum `_uart_status` {
   
kStatus\_UART\_TxBusy = MAKE\_STATUS(kStatusGroup\_UART, 0),
   
kStatus\_UART\_RxBusy = MAKE\_STATUS(kStatusGroup\_UART, 1),
   
kStatus\_UART\_TxIdle = MAKE\_STATUS(kStatusGroup\_UART, 2),
   
kStatus\_UART\_RxIdle = MAKE\_STATUS(kStatusGroup\_UART, 3),
   
kStatus\_UART\_TxWatermarkTooLarge = MAKE\_STATUS(kStatusGroup\_UART, 4),
   
kStatus\_UART\_RxWatermarkTooLarge = MAKE\_STATUS(kStatusGroup\_UART, 5),
   
kStatus\_UART\_FlagCannotClearManually,
   
kStatus\_UART\_Error = MAKE\_STATUS(kStatusGroup\_UART, 7),
   
kStatus\_UART\_RxRingBufferOverrun = MAKE\_STATUS(kStatusGroup\_UART, 8),
   
kStatus\_UART\_RxHardwareOverrun = MAKE\_STATUS(kStatusGroup\_UART, 9),
   
kStatus\_UART\_NoiseError = MAKE\_STATUS(kStatusGroup\_UART, 10),
   
kStatus\_UART\_FramingError = MAKE\_STATUS(kStatusGroup\_UART, 11),
   
kStatus\_UART\_ParityError = MAKE\_STATUS(kStatusGroup\_UART, 12),
   
kStatus\_UART\_BaudrateNotSupport }
   
*Error codes for the UART driver.*
- enum `uart_parity_mode_t` {
   
kUART\_ParityDisabled = 0x0U,
   
kUART\_ParityEven = 0x2U,
   
kUART\_ParityOdd = 0x3U }
   
*UART parity mode.*
- enum `uart_stop_bit_count_t` {
   
kUART\_OneStopBit = 0U,
   
kUART\_TwoStopBit = 1U }
   
*UART stop bit count.*
- enum `_uart_interrupt_enable` {
   
kUART\_LinBreakInterruptEnable = (UART\_BDH\_LBKDIIE\_MASK),
   
kUART\_RxActiveEdgeInterruptEnable = (UART\_BDH\_RXEDGIE\_MASK),
   
kUART\_TxDataRegEmptyInterruptEnable = (UART\_C2\_TIE\_MASK << 8),
   
kUART\_TransmissionCompleteInterruptEnable = (UART\_C2\_TCIE\_MASK << 8),
   
kUART\_RxDataRegFullInterruptEnable = (UART\_C2\_RIE\_MASK << 8),
   
kUART\_IdleLineInterruptEnable = (UART\_C2\_ILIE\_MASK << 8),
   
kUART\_RxOverrunInterruptEnable = (UART\_C3\_ORIE\_MASK << 16),
   
kUART\_NoiseErrorInterruptEnable = (UART\_C3\_NEIE\_MASK << 16),
   
kUART\_FramingErrorInterruptEnable = (UART\_C3\_FEIE\_MASK << 16),
   
kUART\_ParityErrorInterruptEnable = (UART\_C3\_PEIE\_MASK << 16),
   
kUART\_RxFifoOverflowInterruptEnable = (UART\_CFIFO\_RXOFE\_MASK << 24),
   
kUART\_TxFifoOverflowInterruptEnable = (UART\_CFIFO\_TXOFE\_MASK << 24),
   
kUART\_RxFifoUnderflowInterruptEnable = (UART\_CFIFO\_RXUFE\_MASK << 24) }
   
*UART interrupt configuration structure, default settings all disabled.*
- enum `_uart_flags` {

## UART Driver

```
kUART_TxDataRegEmptyFlag = (UART_S1_TDRE_MASK),  
kUART_TransmissionCompleteFlag = (UART_S1_TC_MASK),  
kUART_RxDataRegFullFlag = (UART_S1_RDRF_MASK),  
kUART_IdleLineFlag = (UART_S1_IDLE_MASK),  
kUART_RxOverrunFlag = (UART_S1_OR_MASK),  
kUART_NoiseErrorFlag = (UART_S1_NF_MASK),  
kUART_FramingErrorFlag = (UART_S1_FE_MASK),  
kUART_ParityErrorFlag = (UART_S1_PF_MASK),  
kUART_LinBreakFlag,  
kUART_RxActiveEdgeFlag,  
kUART_RxActiveFlag,  
kUART_NoiseErrorInRxDataRegFlag = (UART_ED_NOISY_MASK << 16),  
kUART_ParityErrorInRxDataRegFlag = (UART_ED_PARITYE_MASK << 16),  
kUART_TxFifoEmptyFlag = (UART_SFIFO_TXEMPT_MASK << 24),  
kUART_RxFifoEmptyFlag = (UART_SFIFO_RXEMPT_MASK << 24),  
kUART_TxFifoOverflowFlag = (UART_SFIFO_TXOF_MASK << 24),  
kUART_RxFifoOverflowFlag = (UART_SFIFO_RXOF_MASK << 24),  
kUART_RxFifoUnderflowFlag = (UART_SFIFO_RXUF_MASK << 24) }
```

*UART status flags.*

## Driver version

- #define **FSL\_UART\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 4))  
*UART driver version 2.1.4.*

## Initialization and deinitialization

- status\_t **UART\_Init** (UART\_Type \*base, const **uart\_config\_t** \*config, uint32\_t srcClock\_Hz)  
*Initializes a UART instance with a user configuration structure and peripheral clock.*
- void **UART\_Deinit** (UART\_Type \*base)  
*Deinitializes a UART instance.*
- void **UART\_GetDefaultConfig** (**uart\_config\_t** \*config)  
*Gets the default configuration structure.*
- status\_t **UART\_SetBaudRate** (UART\_Type \*base, uint32\_t baudRate\_Bps, uint32\_t srcClock\_Hz)  
*Sets the UART instance baud rate.*

## Status

- uint32\_t **UART\_GetStatusFlags** (UART\_Type \*base)  
*Gets UART status flags.*
- status\_t **UART\_ClearStatusFlags** (UART\_Type \*base, uint32\_t mask)  
*Clears status flags with the provided mask.*

## Interrupts

- void **UART\_EnableInterrupts** (UART\_Type \*base, uint32\_t mask)  
*Enables UART interrupts according to the provided mask.*
- void **UART\_DisableInterrupts** (UART\_Type \*base, uint32\_t mask)  
*Disables the UART interrupts according to the provided mask.*
- uint32\_t **UART\_GetEnabledInterrupts** (UART\_Type \*base)  
*Gets the enabled UART interrupts.*

## DMA Control

- static uint32\_t **UART\_GetDataRegisterAddress** (UART\_Type \*base)  
*Gets the UART data register address.*
- static void **UART\_EnableTxDMA** (UART\_Type \*base, bool enable)  
*Enables or disables the UART transmitter DMA request.*
- static void **UART\_EnableRxDMA** (UART\_Type \*base, bool enable)  
*Enables or disables the UART receiver DMA.*

## Bus Operations

- static void **UART\_EnableTx** (UART\_Type \*base, bool enable)  
*Enables or disables the UART transmitter.*
- static void **UART\_EnableRx** (UART\_Type \*base, bool enable)  
*Enables or disables the UART receiver.*
- static void **UART\_WriteByte** (UART\_Type \*base, uint8\_t data)  
*Writes to the TX register.*
- static uint8\_t **UART\_ReadByte** (UART\_Type \*base)  
*Reads the RX register directly.*
- void **UART\_WriteBlocking** (UART\_Type \*base, const uint8\_t \*data, size\_t length)  
*Writes to the TX register using a blocking method.*
- status\_t **UART\_ReadBlocking** (UART\_Type \*base, uint8\_t \*data, size\_t length)  
*Read RX data register using a blocking method.*

## Transactional

- void **UART\_TransferCreateHandle** (UART\_Type \*base, uart\_handle\_t \*handle, **uart\_transfer\_callback\_t** callback, void \*userData)  
*Initializes the UART handle.*
- void **UART\_TransferStartRingBuffer** (UART\_Type \*base, uart\_handle\_t \*handle, uint8\_t \*ringBuffer, size\_t ringBufferSize)  
*Sets up the RX ring buffer.*
- void **UART\_TransferStopRingBuffer** (UART\_Type \*base, uart\_handle\_t \*handle)  
*Aborts the background transfer and uninstalls the ring buffer.*
- status\_t **UART\_TransferSendNonBlocking** (UART\_Type \*base, uart\_handle\_t \*handle, **uart\_transfer\_t** \*xfer)  
*Transmits a buffer of data using the interrupt method.*

## UART Driver

- void **UART\_TransferAbortSend** (UART\_Type \*base, uart\_handle\_t \*handle)  
*Aborts the interrupt-driven data transmit.*
- status\_t **UART\_TransferGetSendCount** (UART\_Type \*base, uart\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes written to the UART TX register.*
- status\_t **UART\_TransferReceiveNonBlocking** (UART\_Type \*base, uart\_handle\_t \*handle, **uart\_transfer\_t** \*xfer, size\_t \*receivedBytes)  
*Receives a buffer of data using an interrupt method.*
- void **UART\_TransferAbortReceive** (UART\_Type \*base, uart\_handle\_t \*handle)  
*Aborts the interrupt-driven data receiving.*
- status\_t **UART\_TransferGetReceiveCount** (UART\_Type \*base, uart\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes that have been received.*
- void **UART\_TransferHandleIRQ** (UART\_Type \*base, uart\_handle\_t \*handle)  
*UART IRQ handle function.*
- void **UART\_TransferHandleErrorIRQ** (UART\_Type \*base, uart\_handle\_t \*handle)  
*UART Error IRQ handle function.*

### 34.2.3 Data Structure Documentation

#### 34.2.3.1 struct uart\_config\_t

##### Data Fields

- uint32\_t baudRate\_Bps  
*UART baud rate.*
- **uart\_parity\_mode\_t parityMode**  
*Parity mode, disabled (default), even, odd.*
- uint8\_t txFifoWatermark  
*TX FIFO watermark.*
- uint8\_t rxFifoWatermark  
*RX FIFO watermark.*
- bool enableTx  
*Enable TX.*
- bool enableRx  
*Enable RX.*

#### 34.2.3.2 struct uart\_transfer\_t

##### Data Fields

- uint8\_t \* data  
*The buffer of data to be transfer.*
- size\_t dataSize  
*The byte count to be transfer.*

### 34.2.3.2.0.75 Field Documentation

34.2.3.2.0.75.1 `uint8_t* uart_transfer_t::data`

34.2.3.2.0.75.2 `size_t uart_transfer_t::dataSize`

### 34.2.3.3 struct \_uart\_handle

#### Data Fields

- `uint8_t *volatile txData`  
*Address of remaining data to send.*
- `volatile size_t txDataSize`  
*Size of the remaining data to send.*
- `size_t txDataSizeAll`  
*Size of the data to send out.*
- `uint8_t *volatile rxData`  
*Address of remaining data to receive.*
- `volatile size_t rxDataSize`  
*Size of the remaining data to receive.*
- `size_t rxDataSizeAll`  
*Size of the data to receive.*
- `uint8_t * rxRingBuffer`  
*Start address of the receiver ring buffer.*
- `size_t rxRingBufferSize`  
*Size of the ring buffer.*
- `volatile uint16_t rxRingBufferHead`  
*Index for the driver to store received data into ring buffer.*
- `volatile uint16_t rxRingBufferTail`  
*Index for the user to get data from the ring buffer.*
- `uart_transfer_callback_t callback`  
*Callback function.*
- `void * userData`  
*UART callback function parameter.*
- `volatile uint8_t txState`  
*TX transfer state.*
- `volatile uint8_t rxState`  
*RX transfer state.*

## UART Driver

### 34.2.3.3.0.76 Field Documentation

- 34.2.3.3.0.76.1 `uint8_t* volatile uart_handle_t::txData`
- 34.2.3.3.0.76.2 `volatile size_t uart_handle_t::txDataSize`
- 34.2.3.3.0.76.3 `size_t uart_handle_t::txDataSizeAll`
- 34.2.3.3.0.76.4 `uint8_t* volatile uart_handle_t::rxData`
- 34.2.3.3.0.76.5 `volatile size_t uart_handle_t::rxDataSize`
- 34.2.3.3.0.76.6 `size_t uart_handle_t::rxDataSizeAll`
- 34.2.3.3.0.76.7 `uint8_t* uart_handle_t::rxRingBuffer`
- 34.2.3.3.0.76.8 `size_t uart_handle_t::rxRingBufferSize`
- 34.2.3.3.0.76.9 `volatile uint16_t uart_handle_t::rxRingBufferHead`
- 34.2.3.3.0.76.10 `volatile uint16_t uart_handle_t::rxRingBufferTail`
- 34.2.3.3.0.76.11 `uart_transfer_callback_t uart_handle_t::callback`
- 34.2.3.3.0.76.12 `void* uart_handle_t::userData`
- 34.2.3.3.0.76.13 `volatile uint8_t uart_handle_t::txState`

### 34.2.4 Macro Definition Documentation

- 34.2.4.1 `#define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))`

### 34.2.5 Typedef Documentation

- 34.2.5.1 `typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)`

### 34.2.6 Enumeration Type Documentation

#### 34.2.6.1 enum \_uart\_status

Enumerator

- `kStatus_UART_TxBusy` Transmitter is busy.
- `kStatus_UART_RxBusy` Receiver is busy.
- `kStatus_UART_TxIdle` UART transmitter is idle.
- `kStatus_UART_RxIdle` UART receiver is idle.
- `kStatus_UART_TxWatermarkTooLarge` TX FIFO watermark too large.

***kStatus\_UART\_RxWatermarkTooLarge*** RX FIFO watermark too large.  
***kStatus\_UART\_FlagCannotClearManually*** UART flag can't be manually cleared.  
***kStatus\_UART\_Error*** Error happens on UART.  
***kStatus\_UART\_RxRingBufferOverrun*** UART RX software ring buffer overrun.  
***kStatus\_UART\_RxHardwareOverrun*** UART RX receiver overrun.  
***kStatus\_UART\_NoiseError*** UART noise error.  
***kStatus\_UART\_FramingError*** UART framing error.  
***kStatus\_UART\_ParityError*** UART parity error.  
***kStatus\_UART\_BaudrateNotSupport*** Baudrate is not support in current clock source.

#### 34.2.6.2 enum uart\_parity\_mode\_t

Enumerator

***kUART\_ParityDisabled*** Parity disabled.  
***kUART\_ParityEven*** Parity enabled, type even, bit setting: PE|PT = 10.  
***kUART\_ParityOdd*** Parity enabled, type odd, bit setting: PE|PT = 11.

#### 34.2.6.3 enum uart\_stop\_bit\_count\_t

Enumerator

***kUART\_OneStopBit*** One stop bit.  
***kUART\_TwoStopBit*** Two stop bits.

#### 34.2.6.4 enum \_uart\_interrupt\_enable

This structure contains the settings for all of the UART interrupt configurations.

Enumerator

***kUART\_LinBreakInterruptEnable*** LIN break detect interrupt.  
***kUART\_RxActiveEdgeInterruptEnable*** RX active edge interrupt.  
***kUART\_TxDataRegEmptyInterruptEnable*** Transmit data register empty interrupt.  
***kUART\_TransmissionCompleteInterruptEnable*** Transmission complete interrupt.  
***kUART\_RxDataRegFullInterruptEnable*** Receiver data register full interrupt.  
***kUART\_IdleLineInterruptEnable*** Idle line interrupt.  
***kUART\_RxOverrunInterruptEnable*** Receiver overrun interrupt.  
***kUART\_NoiseErrorInterruptEnable*** Noise error flag interrupt.  
***kUART\_FramingErrorInterruptEnable*** Framing error flag interrupt.  
***kUART\_ParityErrorInterruptEnable*** Parity error flag interrupt.  
***kUART\_RxFifoOverflowInterruptEnable*** RX FIFO overflow interrupt.  
***kUART\_TxFifoOverflowInterruptEnable*** TX FIFO overflow interrupt.  
***kUART\_RxFifoUnderflowInterruptEnable*** RX FIFO underflow interrupt.

### 34.2.6.5 enum \_uart\_flags

This provides constants for the UART status flags for use in the UART functions.

Enumerator

- kUART\_TxDataRegEmptyFlag* TX data register empty flag.
- kUART\_TransmissionCompleteFlag* Transmission complete flag.
- kUART\_RxDataRegFullFlag* RX data register full flag.
- kUART\_IdleLineFlag* Idle line detect flag.
- kUART\_RxOverrunFlag* RX overrun flag.
- kUART\_NoiseErrorFlag* RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets
- kUART\_FramingErrorFlag* Frame error flag, sets if logic 0 was detected where stop bit expected.
- kUART\_ParityErrorFlag* If parity enabled, sets upon parity error detection.
- kUART\_LinBreakFlag* LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.
- kUART\_RxActiveEdgeFlag* RX pin active edge interrupt flag, sets when active edge detected.
- kUART\_RxActiveFlag* Receiver Active Flag (RAF), sets at beginning of valid start bit.
- kUART\_NoiseErrorInRxDataRegFlag* Noisy bit, sets if noise detected.
- kUART\_ParityErrorInRxDataRegFlag* Parity bit, sets if parity error detected.
- kUART\_TxFifoEmptyFlag* TXEMPT bit, sets if TX buffer is empty.
- kUART\_RxFifoEmptyFlag* RXEMPT bit, sets if RX buffer is empty.
- kUART\_TxFifoOverflowFlag* TXOF bit, sets if TX buffer overflow occurred.
- kUART\_RxFifoOverflowFlag* RXOF bit, sets if receive buffer overflow.
- kUART\_RxFifoUnderflowFlag* RXUF bit, sets if receive buffer underflow.

### 34.2.7 Function Documentation

#### 34.2.7.1 status\_t **UART\_Init** ( **UART\_Type** \* *base*, **const uart\_config\_t** \* *config*, **uint32\_t** *srcClock\_Hz* )

This function configures the UART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the [UART\\_GetDefaultConfig\(\)](#) function. The example below shows how to use this API to configure UART.

```
* uart_config_t uartConfig;
* uartConfig.baudRate_Bps = 115200U;
* uartConfig.parityMode = kUART_ParityDisabled;
* uartConfig.stopBitCount = kUART_OneStopBit;
* uartConfig.txFifoWatermark = 0;
* uartConfig.rxFifoWatermark = 1;
* UART_Init(UART1, &uartConfig, 20000000U);
*
```

Parameters

<i>base</i>	UART peripheral base address.
<i>config</i>	Pointer to the user-defined configuration structure.
<i>srcClock_Hz</i>	UART clock source frequency in HZ.

Return values

<i>kStatus_UART_Baudrate-NotSupport</i>	Baudrate is not support in current clock source.
<i>kStatus_Success</i>	Status UART initialize succeed

### 34.2.7.2 void **UART\_Deinit** ( **UART\_Type** \* *base* )

This function waits for TX complete, disables TX and RX, and disables the UART clock.

Parameters

<i>base</i>	UART peripheral base address.
-------------	-------------------------------

### 34.2.7.3 void **UART\_GetDefaultConfig** ( **uart\_config\_t** \* *config* )

This function initializes the UART configuration structure to a default value. The default values are as follows. *uartConfig->baudRate\_Bps* = 115200U; *uartConfig->bitCountPerChar* = kUART\_8BitsPerChar; *uartConfig->parityMode* = kUART\_ParityDisabled; *uartConfig->stopBitCount* = kUART\_OneStopBit; *uartConfig->txFifoWatermark* = 0; *uartConfig->rxFifoWatermark* = 1; *uartConfig->enableTx* = false; *uartConfig->enableRx* = false;

Parameters

<i>config</i>	Pointer to configuration structure.
---------------	-------------------------------------

### 34.2.7.4 status\_t **UART\_SetBaudRate** ( **UART\_Type** \* *base*, **uint32\_t** *baudRate\_Bps*, **uint32\_t** *srcClock\_Hz* )

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the **UART\_Init**.

```
*   UART\_SetBaudRate(UART1, 115200U, 20000000U);
*
```

## UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>baudRate_Bps</i>	UART baudrate to be set.
<i>srcClock_Hz</i>	UART clock source frequency in Hz.

Return values

<i>kStatus_UART_Baudrate-NotSupport</i>	Baudrate is not support in the current clock source.
<i>kStatus_Success</i>	Set baudrate succeeded.

### 34.2.7.5 `uint32_t UART_GetStatusFlags ( UART_Type * base )`

This function gets all UART status flags. The flags are returned as the logical OR value of the enumerators `_uart_flags`. To check a specific status, compare the return value with enumerators in `_uart_flags`. For example, to check whether the TX is empty, do the following.

```
*     if (kUART_TxDataRegEmptyFlag & UART_GetStatusFlags(UART1))
*     {
*     ...
*     }
```

Parameters

<i>base</i>	UART peripheral base address.
-------------	-------------------------------

Returns

UART status flags which are ORed by the enumerators in the `_uart_flags`.

### 34.2.7.6 `status_t UART_ClearStatusFlags ( UART_Type * base, uint32_t mask )`

This function clears UART status flags with a provided mask. An automatically cleared flag can't be cleared by this function. These flags can only be cleared or set by hardware. kUART\_TxDataRegEmptyFlag, kUART\_TransmissionCompleteFlag, kUART\_RxDataRegFullFlag, kUART\_RxActiveFlag, kUART\_NoiseErrorInRxDataRegFlag, kUART\_ParityErrorInRxDataRegFlag, kUART\_TxFifoEmptyFlag, kUART\_RxFifoEmptyFlag Note that this API should be called when the Tx/Rx is idle. Otherwise it has no effect.

## Parameters

<i>base</i>	UART peripheral base address.
<i>mask</i>	The status flags to be cleared; it is logical OR value of <a href="#">_uart_flags</a> .

## Return values

<i>kStatus_UART_FlagCannotClearManually</i>	The flag can't be cleared by this function but it is cleared automatically by hardware.
<i>kStatus_Success</i>	Status in the mask is cleared.

**34.2.7.7 void UART\_EnableInterrupts ( **UART\_Type** \* *base*, **uint32\_t** *mask* )**

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [\\_uart\\_interrupt\\_enable](#). For example, to enable TX empty interrupt and RX full interrupt, do the following.

```
*     UART_EnableInterrupts(UART1,
                           kUART_TxDataRegEmptyInterruptEnable |
                           kUART_RxDataRegFullInterruptEnable);
*
```

## Parameters

<i>base</i>	UART peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of <a href="#">_uart_interrupt_enable</a> .

**34.2.7.8 void UART\_DisableInterrupts ( **UART\_Type** \* *base*, **uint32\_t** *mask* )**

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [\\_uart\\_interrupt\\_enable](#). For example, to disable TX empty interrupt and RX full interrupt do the following.

```
*     UART_DisableInterrupts(UART1,
                            kUART_TxDataRegEmptyInterruptEnable |
                            kUART_RxDataRegFullInterruptEnable);
*
```

## UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>mask</i>	The interrupts to disable. Logical OR of <a href="#">_uart_interrupt_enable</a> .

### 34.2.7.9 **uint32\_t UART\_GetEnabledInterrupts ( *UART\_Type* \* *base* )**

This function gets the enabled UART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators [\\_uart\\_interrupt\\_enable](#). To check a specific interrupts enable status, compare the return value with enumerators in [\\_uart\\_interrupt\\_enable](#). For example, to check whether TX empty interrupt is enabled, do the following.

```
*     uint32_t enabledInterrupts = UART_GetEnabledInterrupts(UART1);
*
*     if (kUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
*     {
*         ...
*     }
```

Parameters

<i>base</i>	UART peripheral base address.
-------------	-------------------------------

Returns

UART interrupt flags which are logical OR of the enumerators in [\\_uart\\_interrupt\\_enable](#).

### 34.2.7.10 **static uint32\_t UART\_GetDataRegisterAddress ( *UART\_Type* \* *base* ) [inline], [static]**

This function returns the UART data register address, which is mainly used by DMA/eDMA.

Parameters

<i>base</i>	UART peripheral base address.
-------------	-------------------------------

Returns

UART data register addresses which are used both by the transmitter and the receiver.

### 34.2.7.11 **static void UART\_EnableTxDMA ( *UART\_Type* \* *base*, *bool enable* ) [inline], [static]**

This function enables or disables the transmit data register empty flag, S1[TDRE], to generate the DMA requests.

Parameters

<i>base</i>	UART peripheral base address.
<i>enable</i>	True to enable, false to disable.

#### 34.2.7.12 static void UART\_EnableRxDMA ( **UART\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

This function enables or disables the receiver data register full flag, S1[RDRF], to generate DMA requests.

Parameters

<i>base</i>	UART peripheral base address.
<i>enable</i>	True to enable, false to disable.

#### 34.2.7.13 static void UART\_EnableTx ( **UART\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

This function enables or disables the UART transmitter.

Parameters

<i>base</i>	UART peripheral base address.
<i>enable</i>	True to enable, false to disable.

#### 34.2.7.14 static void UART\_EnableRx ( **UART\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

This function enables or disables the UART receiver.

Parameters

<i>base</i>	UART peripheral base address.
<i>enable</i>	True to enable, false to disable.

#### 34.2.7.15 static void UART\_WriteByte ( **UART\_Type** \* *base*, **uint8\_t** *data* ) [**inline**], [**static**]

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

## UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>data</i>	The byte to write.

### 34.2.7.16 static uint8\_t UART\_ReadByte ( **UART\_Type** \* *base* ) [inline], [static]

This function reads data from the RX register directly. The upper layer must ensure that the RX register is full or that the TX FIFO has data before calling this function.

Parameters

<i>base</i>	UART peripheral base address.
-------------	-------------------------------

Returns

The byte read from UART data register.

### 34.2.7.17 void UART\_WriteBlocking ( **UART\_Type** \* *base*, **const uint8\_t** \* *data*, **size\_t** *length* )

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Note

This function does not check whether all data is sent out to the bus. Before disabling the TX, check kUART\_TransmissionCompleteFlag to ensure that the TX is finished.

Parameters

<i>base</i>	UART peripheral base address.
<i>data</i>	Start address of the data to write.
<i>length</i>	Size of the data to write.

### 34.2.7.18 status\_t UART\_ReadBlocking ( **UART\_Type** \* *base*, **uint8\_t** \* *data*, **size\_t** *length* )

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data, and reads data from the TX register.

Parameters

<i>base</i>	UART peripheral base address.
<i>data</i>	Start address of the buffer to store the received data.
<i>length</i>	Size of the buffer.

Return values

<i>kStatus_UART_Rx-HardwareOverrun</i>	Receiver overrun occurred while receiving data.
<i>kStatus_UART_Noise-Error</i>	A noise error occurred while receiving data.
<i>kStatus_UART_Framing-Error</i>	A framing error occurred while receiving data.
<i>kStatus_UART_Parity-Error</i>	A parity error occurred while receiving data.
<i>kStatus_Success</i>	Successfully received all data.

#### 34.2.7.19 void **UART\_TransferCreateHandle** ( **UART\_Type \* base**, **uart\_handle\_t \* handle**, **uart\_transfer\_callback\_t callback**, **void \* userData** )

This function initializes the UART handle which can be used for other UART transactional APIs. Usually, for a specified UART instance, call this API once to get the initialized handle.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>callback</i>	The callback function.
<i>userData</i>	The parameter of the callback function.

#### 34.2.7.20 void **UART\_TransferStartRingBuffer** ( **UART\_Type \* base**, **uart\_handle\_t \* handle**, **uint8\_t \* ringBuffer**, **size\_t ringBufferSize** )

This function sets up the RX ring buffer to a specific UART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the [UART\\_TransferReceiveNonBlocking\(\)](#) API. If data is already received in the ring buffer, the user can get the received data from the ring buffer directly.

## UART Driver

### Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

### Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>ringBuffer</i>	Start address of the ring buffer for background receiving. Pass NULL to disable the ring buffer.
<i>ringBufferSize</i>	Size of the ring buffer.

### 34.2.7.21 void `UART_TransferStopRingBuffer` ( `UART_Type * base, uart_handle_t * handle` )

This function aborts the background transfer and uninstalls the ring buffer.

### Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.

### 34.2.7.22 status\_t `UART_TransferSendNonBlocking` ( `UART_Type * base, uart_handle_t * handle, uart_transfer_t * xfer` )

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the ISR, the UART driver calls the callback function and passes the `kStatus_UART_TxIdle` as status parameter.

### Note

The `kStatus_UART_TxIdle` is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out. Before disabling the TX, check the `kUART_TxTransmissionCompleteFlag` to ensure that the TX is finished.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART transfer structure. See <a href="#">uart_transfer_t</a> .

Return values

<i>kStatus_Success</i>	Successfully start the data transmission.
<i>kStatus_UART_TxBusy</i>	Previous transmission still not finished; data not all written to TX register yet.
<i>kStatus_InvalidArgument</i>	Invalid argument.

#### 34.2.7.23 void **UART\_TransferAbortSend** ( **UART\_Type** \* *base*, **uart\_handle\_t** \* *handle* )

This function aborts the interrupt-driven data sending. The user can get the remainBytes to find out how many bytes are not sent out.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.

#### 34.2.7.24 **status\_t** **UART\_TransferGetSendCount** ( **UART\_Type** \* *base*, **uart\_handle\_t** \* *handle*, **uint32\_t** \* *count* )

This function gets the number of bytes written to the UART TX register by using the interrupt method.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Send bytes count.

Return values

## UART Driver

<i>kStatus_NoTransferInProgress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	The parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

### 34.2.7.25 `status_t UART_TransferReceiveNonBlocking ( UART_Type * base, uart_handle_t * handle, uart_transfer_t * xfer, size_t * receivedBytes )`

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter `receivedBytes` shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter [k-Status\\_UART\\_RxIdle](#). For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the `xfer->data` and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the `xfer->data[5]`. When 5 bytes are received, the UART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the `xfer->data`. When all data is received, the upper layer is notified.

#### Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART transfer structure, see <a href="#">uart_transfer_t</a> .
<i>receivedBytes</i>	Bytes received from the ring buffer directly.

#### Return values

<i>kStatus_Success</i>	Successfully queue the transfer into transmit queue.
<i>kStatus_UART_RxBusy</i>	Previous receive request is not finished.
<i>kStatus_InvalidArgument</i>	Invalid argument.

### 34.2.7.26 `void UART_TransferAbortReceive ( UART_Type * base, uart_handle_t * handle )`

This function aborts the interrupt-driven data receiving. The user can get the `remainBytes` to know how many bytes are not received yet.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.

### 34.2.7.27 **status\_t UART\_TransferGetReceiveCount ( *UART\_Type* \* *base*, *uart\_handle\_t* \* *handle*, *uint32\_t* \* *count* )**

This function gets the number of bytes that have been received.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Receive bytes count.

Return values

<i>kStatus_NoTransferInProgress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

### 34.2.7.28 **void UART\_TransferHandleIRQ ( *UART\_Type* \* *base*, *uart\_handle\_t* \* *handle* )**

This function handles the UART transmit and receive IRQ request.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.

### 34.2.7.29 **void UART\_TransferHandleErrorIRQ ( *UART\_Type* \* *base*, *uart\_handle\_t* \* *handle* )**

This function handles the UART error IRQ request.

## UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.

## 34.3 UART DMA Driver

### 34.3.1 Overview

#### Data Structures

- struct [uart\\_dma\\_handle\\_t](#)  
*UART DMA handle. [More...](#)*

#### Typedefs

- [typedef void\(\\* uart\\_dma\\_transfer\\_callback\\_t \)](#)(UART\_Type \*base, uart\_dma\_handle\_t \*handle, status\_t status, void \*userData)  
*UART transfer callback function.*

#### eDMA transactional

- void [UART\\_TransferCreateHandleDMA](#) (UART\_Type \*base, uart\_dma\_handle\_t \*handle, [uart\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData, dma\_handle\_t \*txDmaHandle, dma\_handle\_t \*rxDmaHandle)  
*Initializes the UART handle which is used in transactional functions and sets the callback.*
- status\_t [UART\\_TransferSendDMA](#) (UART\_Type \*base, uart\_dma\_handle\_t \*handle, [uart\\_transfer\\_t](#) \*xfer)  
*Sends data using DMA.*
- status\_t [UART\\_TransferReceiveDMA](#) (UART\_Type \*base, uart\_dma\_handle\_t \*handle, [uart\\_transfer\\_t](#) \*xfer)  
*Receives data using DMA.*
- void [UART\\_TransferAbortSendDMA](#) (UART\_Type \*base, uart\_dma\_handle\_t \*handle)  
*Aborts the send data using DMA.*
- void [UART\\_TransferAbortReceiveDMA](#) (UART\_Type \*base, uart\_dma\_handle\_t \*handle)  
*Aborts the received data using DMA.*
- status\_t [UART\\_TransferGetSendCountDMA](#) (UART\_Type \*base, uart\_dma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes written to UART TX register.*
- status\_t [UART\\_TransferGetReceiveCountDMA](#) (UART\_Type \*base, uart\_dma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes that have been received.*

### 34.3.2 Data Structure Documentation

#### 34.3.2.1 struct \_uart\_dma\_handle

##### Data Fields

- [UART\\_Type](#) \* [base](#)

## UART DMA Driver

- *UART peripheral base address.*
- `uart_dma_transfer_callback_t callback`  
*Callback function.*
- `void *userData`  
*UART callback function parameter.*
- `size_t rxDataSizeAll`  
*Size of the data to receive.*
- `size_t txDataSizeAll`  
*Size of the data to send out.*
- `dma_handle_t *txDmaHandle`  
*The DMA TX channel used.*
- `dma_handle_t *rxDmaHandle`  
*The DMA RX channel used.*
- `volatile uint8_t txState`  
*TX transfer state.*
- `volatile uint8_t rxState`  
*RX transfer state.*

### 34.3.2.1.0.77 Field Documentation

34.3.2.1.0.77.1 `UART_Type* uart_dma_handle_t::base`

34.3.2.1.0.77.2 `uart_dma_transfer_callback_t uart_dma_handle_t::callback`

34.3.2.1.0.77.3 `void* uart_dma_handle_t::userData`

34.3.2.1.0.77.4 `size_t uart_dma_handle_t::rxDataSizeAll`

34.3.2.1.0.77.5 `size_t uart_dma_handle_t::txDataSizeAll`

34.3.2.1.0.77.6 `dma_handle_t* uart_dma_handle_t::txDmaHandle`

34.3.2.1.0.77.7 `dma_handle_t* uart_dma_handle_t::rxDmaHandle`

34.3.2.1.0.77.8 `volatile uint8_t uart_dma_handle_t::txState`

### 34.3.3 Typedef Documentation

34.3.3.1 `typedef void(* uart_dma_transfer_callback_t)(UART_Type *base,  
uart_dma_handle_t *handle, status_t status, void *userData)`

### 34.3.4 Function Documentation

34.3.4.1 `void UART_TransferCreateHandleDMA ( UART_Type * base, uart_dma_handle_t  
* handle, uart_dma_transfer_callback_t callback, void * userData,  
dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle )`

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_dma_handle_t</code> structure.
<i>callback</i>	UART callback, NULL means no callback.
<i>userData</i>	User callback function data.
<i>rxDmaHandle</i>	User requested DMA handle for the RX DMA transfer.
<i>txDmaHandle</i>	User requested DMA handle for the TX DMA transfer.

#### 34.3.4.2 `status_t UART_TransferSendDMA ( UART_Type * base, uart_dma_handle_t * handle, uart_transfer_t * xfer )`

This function sends data using DMA. This is non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART DMA transfer structure. See <a href="#">uart_transfer_t</a> .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

#### 34.3.4.3 `status_t UART_TransferReceiveDMA ( UART_Type * base, uart_dma_handle_t * handle, uart_transfer_t * xfer )`

This function receives data using DMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

## UART DMA Driver

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_dma_handle_t</code> structure.
<i>xfer</i>	UART DMA transfer structure. See <a href="#">uart_transfer_t</a> .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_RxBusy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

### 34.3.4.4 void `UART_TransferAbortSendDMA` ( `UART_Type * base, uart_dma_handle_t * handle` )

This function aborts the sent data using DMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to <code>uart_dma_handle_t</code> structure.

### 34.3.4.5 void `UART_TransferAbortReceiveDMA` ( `UART_Type * base, uart_dma_handle_t * handle` )

This function abort receive data which using DMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to <code>uart_dma_handle_t</code> structure.

### 34.3.4.6 status\_t `UART_TransferGetSendCountDMA` ( `UART_Type * base, uart_dma_handle_t * handle, uint32_t * count` )

This function gets the number of bytes written to UART TX register by DMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Send bytes count.

Return values

<i>kStatus_NoTransferIn-Progress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

#### 34.3.4.7 **status\_t UART\_TransferGetReceiveCountDMA ( *UART\_Type \* base, uart\_dma\_handle\_t \* handle, uint32\_t \* count* )**

This function gets the number of bytes that have been received.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Receive bytes count.

Return values

<i>kStatus_NoTransferIn-Progress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

### 34.4 UART eDMA Driver

#### 34.4.1 Overview

#### Data Structures

- struct [uart\\_edma\\_handle\\_t](#)  
*UART eDMA handle.* [More...](#)

#### TypeDefs

- [typedef void\(\\* uart\\_edma\\_transfer\\_callback\\_t \)\(UART\\_Type \\*base, uart\\_edma\\_handle\\_t \\*handle, status\\_t status, void \\*userData\)](#)  
*UART transfer callback function.*

#### eDMA transactional

- void [UART\\_TransferCreateHandleEDMA](#) (UART\_Type \*base, uart\_edma\_handle\_t \*handle, [uart\\_edma\\_transfer\\_callback\\_t](#)  callback, void \*userData, [edma\\_handle\\_t](#)  \*txEdmaHandle, [edma\\_handle\\_t](#)  \*rxEdmaHandle)  
*Initializes the UART handle which is used in transactional functions.*
- status\_t [UART\\_SendEDMA](#) (UART\_Type \*base, uart\_edma\_handle\_t \*handle, [uart\\_transfer\\_t](#)  \*xfer)  
*Sends data using eDMA.*
- status\_t [UART\\_ReceiveEDMA](#) (UART\_Type \*base, uart\_edma\_handle\_t \*handle, [uart\\_transfer\\_t](#)  \*xfer)  
*Receives data using eDMA.*
- void [UART\\_TransferAbortSendEDMA](#) (UART\_Type \*base, uart\_edma\_handle\_t \*handle)  
*Aborts the sent data using eDMA.*
- void [UART\\_TransferAbortReceiveEDMA](#) (UART\_Type \*base, uart\_edma\_handle\_t \*handle)  
*Aborts the receive data using eDMA.*
- status\_t [UART\\_TransferGetSendCountEDMA](#) (UART\_Type \*base, uart\_edma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes that have been written to UART TX register.*
- status\_t [UART\\_TransferGetReceiveCountEDMA](#) (UART\_Type \*base, uart\_edma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of received bytes.*

#### 34.4.2 Data Structure Documentation

##### 34.4.2.1 struct \_uart\_edma\_handle

###### Data Fields

- [uart\\_edma\\_transfer\\_callback\\_t](#)  callback

- *Callback function.*
- `void *userData`  
*UART callback function parameter.*
- `size_t rxDataSizeAll`  
*Size of the data to receive.*
- `size_t txDataSizeAll`  
*Size of the data to send out.*
- `edma_handle_t *txEdmaHandle`  
*The eDMA TX channel used.*
- `edma_handle_t *rxEdmaHandle`  
*The eDMA RX channel used.*
- `uint8_t nbytes`  
*eDMA minor byte transfer count initially configured.*
- `volatile uint8_t txState`  
*TX transfer state.*
- `volatile uint8_t rxState`  
*RX transfer state.*

#### 34.4.2.1.0.78 Field Documentation

34.4.2.1.0.78.1 `uart_edma_transfer_callback_t uart_edma_handle_t::callback`

34.4.2.1.0.78.2 `void* uart_edma_handle_t::userData`

34.4.2.1.0.78.3 `size_t uart_edma_handle_t::rxDataSizeAll`

34.4.2.1.0.78.4 `size_t uart_edma_handle_t::txDataSizeAll`

34.4.2.1.0.78.5 `edma_handle_t* uart_edma_handle_t::txEdmaHandle`

34.4.2.1.0.78.6 `edma_handle_t* uart_edma_handle_t::rxEdmaHandle`

34.4.2.1.0.78.7 `uint8_t uart_edma_handle_t::nbytes`

34.4.2.1.0.78.8 `volatile uint8_t uart_edma_handle_t::txState`

#### 34.4.3 Typedef Documentation

34.4.3.1 `typedef void(* uart_edma_transfer_callback_t)(UART_Type *base,  
uart_edma_handle_t *handle, status_t status, void *userData)`

#### 34.4.4 Function Documentation

34.4.4.1 `void UART_TransferCreateHandleEDMA ( UART_Type * base,  
uart_edma_handle_t * handle, uart_edma_transfer_callback_t callback, void *  
userData, edma_handle_t * txEdmaHandle, edma_handle_t * rxEdmaHandle )`

## UART eDMA Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.
<i>callback</i>	UART callback, NULL means no callback.
<i>userData</i>	User callback function data.
<i>rxEdmaHandle</i>	User-requested DMA handle for RX DMA transfer.
<i>txEdmaHandle</i>	User-requested DMA handle for TX DMA transfer.

### 34.4.4.2 `status_t UART_SendEDMA ( UART_Type * base, uart_edma_handle_t * handle, uart_transfer_t * xfer )`

This function sends data using eDMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART eDMA transfer structure. See <a href="#">uart_transfer_t</a> .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

### 34.4.4.3 `status_t UART_ReceiveEDMA ( UART_Type * base, uart_edma_handle_t * handle, uart_transfer_t * xfer )`

This function receives data using eDMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.
<i>xfer</i>	UART eDMA transfer structure. See <a href="#">uart_transfer_t</a> .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_RxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

#### **34.4.4.4 void `UART_TransferAbortSendEDMA` ( `UART_Type * base, uart_edma_handle_t * handle` )**

This function aborts sent data using eDMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.

#### **34.4.4.5 void `UART_TransferAbortReceiveEDMA` ( `UART_Type * base, uart_edma_handle_t * handle` )**

This function aborts receive data using eDMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.

#### **34.4.4.6 status\_t `UART_TransferGetSendCountEDMA` ( `UART_Type * base, uart_edma_handle_t * handle, uint32_t * count` )**

This function gets the number of bytes that have been written to UART TX register by DMA.

## UART eDMA Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Send bytes count.

Return values

<i>kStatus_NoTransferIn-Progress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

### 34.4.4.7 **status\_t UART\_TransferGetReceiveCountEDMA ( *UART\_Type \* base*, *uart\_edma\_handle\_t \* handle*, *uint32\_t \* count* )**

This function gets the number of received bytes.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Receive bytes count.

Return values

<i>kStatus_NoTransferIn-Progress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

## 34.5 UART FreeRTOS Driver

### 34.5.1 Overview

#### Data Structures

- struct `uart_rtos_config_t`  
*UART configuration structure.* [More...](#)

#### UART RTOS Operation

- int `UART_RTOS_Init` (`uart_rtos_handle_t *handle`, `uart_handle_t *t_handle`, const `uart_rtos_config_t *cfg`)  
*Initializes a UART instance for operation in RTOS.*
- int `UART_RTOS_Deinit` (`uart_rtos_handle_t *handle`)  
*Deinitializes a UART instance for operation.*

#### UART transactional Operation

- int `UART_RTOS_Send` (`uart_rtos_handle_t *handle`, const `uint8_t *buffer`, `uint32_t length`)  
*Sends data in the background.*
- int `UART_RTOS_Receive` (`uart_rtos_handle_t *handle`, `uint8_t *buffer`, `uint32_t length`, `size_t *received`)  
*Receives data.*

### 34.5.2 Data Structure Documentation

#### 34.5.2.1 struct `uart_rtos_config_t`

##### Data Fields

- `UART_Type * base`  
*UART base address.*
- `uint32_t sreclk`  
*UART source clock in Hz.*
- `uint32_t baudrate`  
*Desired communication speed.*
- `uart_parity_mode_t parity`  
*Parity setting.*
- `uart_stop_bit_count_t stopbits`  
*Number of stop bits to use.*
- `uint8_t * buffer`  
*Buffer for background reception.*
- `uint32_t buffer_size`  
*Size of buffer for background reception.*

### 34.5.3 Function Documentation

34.5.3.1 `int UART_RTOS_Init( uart_rtos_handle_t * handle, uart_handle_t * t_handle,  
const uart_rtos_config_t * cfg )`

Parameters

<i>handle</i>	The RTOS UART handle, the pointer to an allocated space for RTOS context.
<i>t_handle</i>	The pointer to the allocated space to store the transactional layer internal state.
<i>cfg</i>	The pointer to the parameters required to configure the UART after initialization.

Returns

0 succeed; otherwise fail.

#### 34.5.3.2 int **UART\_RTOS\_Deinit** ( *uart\_rtos\_handle\_t \* handle* )

This function deinitializes the UART module, sets all register values to reset value, and frees the resources.

Parameters

<i>handle</i>	The RTOS UART handle.
---------------	-----------------------

#### 34.5.3.3 int **UART\_RTOS\_Send** ( *uart\_rtos\_handle\_t \* handle, const uint8\_t \* buffer, uint32\_t length* )

This function sends data. It is a synchronous API. If the hardware buffer is full, the task is in the blocked state.

Parameters

<i>handle</i>	The RTOS UART handle.
<i>buffer</i>	The pointer to the buffer to send.
<i>length</i>	The number of bytes to send.

#### 34.5.3.4 int **UART\_RTOS\_Receive** ( *uart\_rtos\_handle\_t \* handle, uint8\_t \* buffer, uint32\_t length, size\_t \* received* )

This function receives data from UART. It is a synchronous API. If data is immediately available, it is returned immediately and the number of bytes received.

## UART FreeRTOS Driver

### Parameters

<i>handle</i>	The RTOS UART handle.
<i>buffer</i>	The pointer to the buffer to write received data.
<i>length</i>	The number of bytes to receive.
<i>received</i>	The pointer to a variable of size_t where the number of received data is filled.

# Chapter 35

## VREF: Voltage Reference Driver

### 35.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Crossbar Voltage Reference (VREF) block of MCUXpresso SDK devices.

The Voltage Reference(VREF) supplies an accurate 1.2 V voltage output that can be trimmed in 0.5 mV steps. VREF can be used in applications to provide a reference voltage to external devices and to internal analog peripherals, such as the ADC, DAC, or CMP. The voltage reference has operating modes that provide different levels of supply rejection and power consumption.

To configure the VREF driver, configure `vref_config_t` structure in one of two ways.

1. Use the `VREF_GetDefaultConfig()` function.
2. Set the parameter in the `vref_config_t` structure.

To initialize the VREF driver, call the `VREF_Init()` function and pass a pointer to the `vref_config_t` structure.

To de-initialize the VREF driver, call the `VREF_Deinit()` function.

### 35.2 Typical use case and example

This example shows how to generate a reference voltage by using the VREF module.

```
vref_config_t vrefUserConfig;
VREF_GetDefaultConfig(&vrefUserConfig); /* Gets a default configuration. */
VREF_Init(VREF, &vrefUserConfig);      /* Initializes and configures the VREF module */

/* Do something */

VREF_Deinit(VREF); /* De-initializes the VREF module */
```

## Data Structures

- struct `vref_config_t`  
*The description structure for the VREF module. [More...](#)*

## Enumerations

- enum `vref_buffer_mode_t` {  
    `kVREF_ModeBandgapOnly` = 0U,  
    `kVREF_ModeHighPowerBuffer` = 1U,  
    `kVREF_ModeLowPowerBuffer` = 2U }  
*VREF modes.*

## Function Documentation

### Driver version

- #define **FSL\_VREF\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 0))  
*Version 2.1.0.*

### VREF functional operation

- void **VREF\_Init** (VREF\_Type \*base, const vref\_config\_t \*config)  
*Enables the clock gate and configures the VREF module according to the configuration structure.*
- void **VREF\_Deinit** (VREF\_Type \*base)  
*Stops and disables the clock for the VREF module.*
- void **VREF\_GetDefaultConfig** (vref\_config\_t \*config)  
*Initializes the VREF configuration structure.*
- void **VREF\_SetTrimVal** (VREF\_Type \*base, uint8\_t trimValue)  
*Sets a TRIM value for the reference voltage.*
- static uint8\_t **VREF\_GetTrimVal** (VREF\_Type \*base)  
*Reads the value of the TRIM meaning output voltage.*

### 35.3 Data Structure Documentation

#### 35.3.1 struct vref\_config\_t

##### Data Fields

- vref\_buffer\_mode\_t bufferMode  
*Buffer mode selection.*

### 35.4 Macro Definition Documentation

#### 35.4.1 #define FSL\_VREF\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 0))

### 35.5 Enumeration Type Documentation

#### 35.5.1 enum vref\_buffer\_mode\_t

Enumerator

**kVREF\_ModeBandgapOnly** Bandgap on only, for stabilization and startup.

**kVREF\_ModeHighPowerBuffer** High-power buffer mode enabled.

**kVREF\_ModeLowPowerBuffer** Low-power buffer mode enabled.

### 35.6 Function Documentation

#### 35.6.1 void VREF\_Init ( VREF\_Type \* *base*, const vref\_config\_t \* *config* )

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up **vref\_config\_t** parameters and how to call the VREF\_Init function by passing in these parameters. This is an example.

```

*   vref_config_t vrefConfig;
*   vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
*   vrefConfig.enableExternalVoltRef = false;
*   vrefConfig.enableLowRef = false;
*   VREF_Init(VREF, &vrefConfig);
*

```

## Parameters

<i>base</i>	VREF peripheral address.
<i>config</i>	Pointer to the configuration structure.

**35.6.2 void VREF\_Deinit ( VREF\_Type \* *base* )**

This function should be called to shut down the module. This is an example.

```

*   vref_config_t vrefUserConfig;
*   VREF_Init(VREF);
*   VREF_GetDefaultConfig(&vrefUserConfig);
*   ...
*   VREF_Deinit(VREF);
*

```

## Parameters

<i>base</i>	VREF peripheral address.
-------------	--------------------------

**35.6.3 void VREF\_GetDefaultConfig ( vref\_config\_t \* *config* )**

This function initializes the VREF configuration structure to default values. This is an example.

```

*   vrefConfig->bufferMode = kVREF_ModeHighPowerBuffer;
*   vrefConfig->enableExternalVoltRef = false;
*   vrefConfig->enableLowRef = false;
*

```

## Parameters

<i>config</i>	Pointer to the initialization structure.
---------------	--

**35.6.4 void VREF\_SetTrimVal ( VREF\_Type \* *base*, uint8\_t *trimValue* )**

This function sets a TRIM value for the reference voltage. Note that the TRIM value maximum is 0x3F.

## Function Documentation

Parameters

<i>base</i>	VREF peripheral address.
<i>trimValue</i>	Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)).

### 35.6.5 static uint8\_t VREF\_GetTrimVal ( VREF\_Type \* *base* ) [inline], [static]

This function gets the TRIM value from the TRM register.

Parameters

<i>base</i>	VREF peripheral address.
-------------	--------------------------

Returns

Six-bit value of trim setting.

# Chapter 36

## WDOG: Watchdog Timer Driver

### 36.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Watchdog module (WDOG) of MCUXpresso SDK devices.

### 36.2 Typical use case

```
wdog_config_t config;
WDOG_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
config.enableWindowMode = true;
config.windowValue = 0x1ffU;
WDOG_Init(wdog_base, &config);
```

## Data Structures

- struct `wdog_work_mode_t`  
*Defines WDOG work mode. [More...](#)*
- struct `wdog_config_t`  
*Describes WDOG configuration structure. [More...](#)*
- struct `wdog_test_config_t`  
*Describes WDOG test mode configuration structure. [More...](#)*

## Enumerations

- enum `wdog_clock_source_t` {  
  `kWDOG_LpoClockSource` = 0U,  
  `kWDOG_AlternateClockSource` = 1U }  
*Describes WDOG clock source.*
- enum `wdog_clock_prescaler_t` {  
  `kWDOG_ClockPrescalerDivide1` = 0x0U,  
  `kWDOG_ClockPrescalerDivide2` = 0x1U,  
  `kWDOG_ClockPrescalerDivide3` = 0x2U,  
  `kWDOG_ClockPrescalerDivide4` = 0x3U,  
  `kWDOG_ClockPrescalerDivide5` = 0x4U,  
  `kWDOG_ClockPrescalerDivide6` = 0x5U,  
  `kWDOG_ClockPrescalerDivide7` = 0x6U,  
  `kWDOG_ClockPrescalerDivide8` = 0x7U }  
*Describes the selection of the clock prescaler.*
- enum `wdog_test_mode_t` {  
  `kWDOG_QuickTest` = 0U,  
  `kWDOG_ByeTest` = 1U }  
*Describes WDOG test mode.*

## Typical use case

- enum `wdog_tested_byte_t` {  
  `kWDOG_TestByte0` = 0U,  
  `kWDOG_TestByte1` = 1U,  
  `kWDOG_TestByte2` = 2U,  
  `kWDOG_TestByte3` = 3U }  
    *Describes WDOG tested byte selection in byte test mode.*
- enum `_wdog_interrupt_enable_t` { `kWDOG_InterruptEnable` = `WDOG_STCTRLH_IRQRSTEN_MASK` }  
    *WDOG interrupt configuration structure, default settings all disabled.*
- enum `_wdog_status_flags_t` {  
  `kWDOG_RunningFlag` = `WDOG_STCTRLH_WDOGEN_MASK`,  
  `kWDOG_TimeoutFlag` = `WDOG_STCTRLL_INTFLG_MASK` }  
    *WDOG status flags.*

## Driver version

- #define `FSL_WDOG_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)  
    *Defines WDOG driver version 2.0.0.*

## Unlock sequence

- #define `WDOG_FIRST_WORD_OF_UNLOCK` (0xC520U)  
    *First word of unlock sequence.*
- #define `WDOG_SECOND_WORD_OF_UNLOCK` (0xD928U)  
    *Second word of unlock sequence.*

## Refresh sequence

- #define `WDOG_FIRST_WORD_OF_REFRESH` (0xA602U)  
    *First word of refresh sequence.*
- #define `WDOG_SECOND_WORD_OF_REFRESH` (0xB480U)  
    *Second word of refresh sequence.*

## WDOG Initialization and De-initialization

- void `WDOG_GetDefaultConfig` (`wdog_config_t` \*config)  
    *Initializes the WDOG configuration structure.*
- void `WDOG_Init` (`WDOG_Type` \*base, const `wdog_config_t` \*config)  
    *Initializes the WDOG.*
- void `WDOG_Deinit` (`WDOG_Type` \*base)  
    *Shuts down the WDOG.*
- void `WDOG_SetTestModeConfig` (`WDOG_Type` \*base, `wdog_test_config_t` \*config)  
    *Configures the WDOG functional test.*

## WDOG Functional Operation

- static void `WDOG_Enable` (`WDOG_Type` \*base)  
    *Enables the WDOG module.*
- static void `WDOG_Disable` (`WDOG_Type` \*base)

- static void **WDOG\_EnableInterrupts** (WDOG\_Type \*base, uint32\_t mask)
 

*Disables the WDOG module.*
- static void **WDOG\_DisableInterrupts** (WDOG\_Type \*base, uint32\_t mask)
 

*Enables the WDOG interrupt.*
- uint32\_t **WDOG\_GetStatusFlags** (WDOG\_Type \*base)
 

*Gets the WDOG all status flags.*
- void **WDOG\_ClearStatusFlags** (WDOG\_Type \*base, uint32\_t mask)
 

*Clears the WDOG flag.*
- static void **WDOG\_SetTimeoutValue** (WDOG\_Type \*base, uint32\_t timeoutCount)
 

*Sets the WDOG timeout value.*
- static void **WDOG\_SetWindowValue** (WDOG\_Type \*base, uint32\_t windowValue)
 

*Sets the WDOG window value.*
- static void **WDOG\_Unlock** (WDOG\_Type \*base)
 

*Unlocks the WDOG register written.*
- void **WDOG\_Refresh** (WDOG\_Type \*base)
 

*Refreshes the WDOG timer.*
- static uint16\_t **WDOG\_GetResetCount** (WDOG\_Type \*base)
 

*Gets the WDOG reset count.*
- static void **WDOG\_ClearResetCount** (WDOG\_Type \*base)
 

*Clears the WDOG reset count.*

## 36.3 Data Structure Documentation

### 36.3.1 struct wdog\_work\_mode\_t

#### Data Fields

- bool **enableWait**

*Enables or disables WDOG in wait mode.*
- bool **enableStop**

*Enables or disables WDOG in stop mode.*
- bool **enableDebug**

*Enables or disables WDOG in debug mode.*

### 36.3.2 struct wdog\_config\_t

#### Data Fields

- bool **enableWdog**

*Enables or disables WDOG.*
- **wdog\_clock\_source\_t clockSource**

*Clock source select.*
- **wdog\_clock\_prescaler\_t prescaler**

*Clock prescaler value.*
- **wdog\_work\_mode\_t workMode**

*Configures WDOG work mode in debug stop and wait mode.*
- bool **enableUpdate**

## Enumeration Type Documentation

- `bool enableInterrupt`  
*Enables or disables WDOG interrupt.*
- `bool enableWindowMode`  
*Enables or disables WDOG window mode.*
- `uint32_t windowValue`  
*Window value.*
- `uint32_t timeoutValue`  
*Timeout value.*

### 36.3.3 `struct wdog_test_config_t`

#### Data Fields

- `wdog_test_mode_t testMode`  
*Selects test mode.*
- `wdog_tested_byte_t testedByte`  
*Selects tested byte in byte test mode.*
- `uint32_t timeoutValue`  
*Timeout value.*

## 36.4 Macro Definition Documentation

### 36.4.1 `#define FSL_WDOG_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

## 36.5 Enumeration Type Documentation

### 36.5.1 `enum wdog_clock_source_t`

Enumerator

*kWDOG\_LpoClockSource* WDOG clock sourced from LPO.

*kWDOG\_AlternateClockSource* WDOG clock sourced from alternate clock source.

### 36.5.2 `enum wdog_clock_prescaler_t`

Enumerator

*kWDOG\_ClockPrescalerDivide1* Divided by 1.  
*kWDOG\_ClockPrescalerDivide2* Divided by 2.  
*kWDOG\_ClockPrescalerDivide3* Divided by 3.  
*kWDOG\_ClockPrescalerDivide4* Divided by 4.  
*kWDOG\_ClockPrescalerDivide5* Divided by 5.  
*kWDOG\_ClockPrescalerDivide6* Divided by 6.  
*kWDOG\_ClockPrescalerDivide7* Divided by 7.

*kWDOG\_ClockPrescalerDivide8* Divided by 8.

### 36.5.3 enum wdog\_test\_mode\_t

Enumerator

*kWDOG\_QuickTest* Selects quick test.

*kWDOG\_Bytetest* Selects byte test.

### 36.5.4 enum wdog\_tested\_byte\_t

Enumerator

*kWDOG\_TestByte0* Byte 0 selected in byte test mode.

*kWDOG\_TestByte1* Byte 1 selected in byte test mode.

*kWDOG\_TestByte2* Byte 2 selected in byte test mode.

*kWDOG\_TestByte3* Byte 3 selected in byte test mode.

### 36.5.5 enum \_wdog\_interrupt\_enable\_t

This structure contains the settings for all of the WDOG interrupt configurations.

Enumerator

*kWDOG\_InterruptEnable* WDOG timeout generates an interrupt before reset.

### 36.5.6 enum \_wdog\_status\_flags\_t

This structure contains the WDOG status flags for use in the WDOG functions.

Enumerator

*kWDOG\_RunningFlag* Running flag, set when WDOG is enabled.

*kWDOG\_TimeoutFlag* Interrupt flag, set when an exception occurs.

## 36.6 Function Documentation

### 36.6.1 void WDOG\_GetDefaultConfig ( wdog\_config\_t \* config )

This function initializes the WDOG configuration structure to default values. The default values are as follows.

## Function Documentation

```
* wdogConfig->enableWdog = true;
* wdogConfig->clockSource = KWDOG_IpoClockSource;
* wdogConfig->prescaler = kWDOG_ClockPrescalerDivide1;
* wdogConfig->workMode.enableWait = true;
* wdogConfig->workMode.enableStop = false;
* wdogConfig->workMode.enableDebug = false;
* wdogConfig->enableUpdate = true;
* wdogConfig->enableInterrupt = false;
* wdogConfig->enableWindowMode = false;
* wdogConfig->windowValue = 0;
* wdogConfig->timeoutValue = 0xFFFFU;
*
```

### Parameters

<i>config</i>	Pointer to the WDOG configuration structure.
---------------	--

### See Also

[wdog\\_config\\_t](#)

### 36.6.2 void WDOG\_Init ( **WDOG\_Type** \* *base*, **const wdog\_config\_t** \* *config* )

This function initializes the WDOG. When called, the WDOG runs according to the configuration. To reconfigure WDOG without forcing a reset first, enableUpdate must be set to true in the configuration.

This is an example.

```
* wdog_config_t config;
* WDOG_GetDefaultConfig(&config);
* config.timeoutValue = 0x7ffU;
* config.enableUpdate = true;
* WDOG_Init(wdog_base,&config);
*
```

### Parameters

<i>base</i>	WDOG peripheral base address
<i>config</i>	The configuration of WDOG

### 36.6.3 void WDOG\_Deinit ( **WDOG\_Type** \* *base* )

This function shuts down the WDOG. Ensure that the WDOG\_STCTRLH.ALLOWUPDATE is 1 which indicates that the register update is enabled.

### 36.6.4 void WDOG\_SetTestModeConfig ( WDOG\_Type \* *base*, wdog\_test\_config\_t \* *config* )

This function is used to configure the WDOG functional test. When called, the WDOG goes into test mode and runs according to the configuration. Ensure that the WDOG\_STCTRLH.ALLOWUPDATE is 1 which means that the register update is enabled.

This is an example.

```
*     wdog_test_config_t test_config;
*     test_config.testMode = kWDOG_QuickTest;
*     test_config.timeoutValue = 0xffffffff;
*     WDOG_SetTestModeConfig(wdog_base, &test_config);
*
```

Parameters

<i>base</i>	WDOG peripheral base address
<i>config</i>	The functional test configuration of WDOG

### 36.6.5 static void WDOG\_Enable ( WDOG\_Type \* *base* ) [inline], [static]

This function write value into WDOG\_STCTRLH register to enable the WDOG, it is a write-once register, make sure that the WCT window is still open and this register has not been written in this WCT while this function is called.

Parameters

<i>base</i>	WDOG peripheral base address
-------------	------------------------------

### 36.6.6 static void WDOG\_Disable ( WDOG\_Type \* *base* ) [inline], [static]

This function writes a value into the WDOG\_STCTRLH register to disable the WDOG. It is a write-once register. Ensure that the WCT window is still open and that register has not been written to in this WCT while the function is called.

Parameters

## Function Documentation

<i>base</i>	WDOG peripheral base address
-------------	------------------------------

### 36.6.7 static void WDOG\_EnableInterrupts ( **WDOG\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

This function writes a value into the WDOG\_STCTRLH register to enable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

<i>base</i>	WDOG peripheral base address
<i>mask</i>	The interrupts to enable The parameter can be combination of the following source if defined. <ul style="list-style-type: none"><li>• kWDOG_InterruptEnable</li></ul>

### 36.6.8 static void WDOG\_DisableInterrupts ( **WDOG\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

This function writes a value into the WDOG\_STCTRLH register to disable the WDOG interrupt. It is a write-once register. Ensure that the WCT window is still open and the register has not been written to in this WCT while the function is called.

Parameters

<i>base</i>	WDOG peripheral base address
<i>mask</i>	The interrupts to disable The parameter can be combination of the following source if defined. <ul style="list-style-type: none"><li>• kWDOG_InterruptEnable</li></ul>

### 36.6.9 **uint32\_t** WDOG\_GetStatusFlags ( **WDOG\_Type** \* *base* )

This function gets all status flags.

This is an example for getting the Running Flag.

```
*     uint32_t status;
*     status = WDOG_GetStatusFlags (wdog_base) &
*             kWDOG_RunningFlag;
```

\*

## Parameters

<i>base</i>	WDOG peripheral base address
-------------	------------------------------

## Returns

State of the status flag: asserted (true) or not-asserted (false).

## See Also

[\\_wdog\\_status\\_flags\\_t](#)

- true: a related status flag has been set.
- false: a related status flag is not set.

**36.6.10 void WDOG\_ClearStatusFlags ( WDOG\_Type \* *base*, uint32\_t *mask* )**

This function clears the WDOG status flag.

This is an example for clearing the timeout (interrupt) flag.

```
*   WDOG_ClearStatusFlags (wdog_base, kWDOG_TimeoutFlag);
*
```

## Parameters

<i>base</i>	WDOG peripheral base address
<i>mask</i>	The status flags to clear. The parameter could be any combination of the following values. kWDOG_TimeoutFlag

**36.6.11 static void WDOG\_SetTimeoutValue ( WDOG\_Type \* *base*, uint32\_t *timeoutCount* ) [inline], [static]**

This function sets the timeout value. It should be ensured that the time-out value for the WDOG is always greater than 2xWCT time + 20 bus clock cycles. This function writes a value into WDOG\_TOVALH and WDOG\_TOVALL registers which are write-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

## Function Documentation

Parameters

<i>base</i>	WDOG peripheral base address
<i>timeoutCount</i>	WDOG timeout value; count of WDOG clock tick.

### 36.6.12 static void WDOG\_SetWindowValue ( WDOG\_Type \* *base*, uint32\_t *windowValue* ) [inline], [static]

This function sets the WDOG window value. This function writes a value into WDOG\_WINH and WDOG\_WINL registers which are write-once. Ensure the WCT window is still open and the two registers have not been written to in this WCT while the function is called.

Parameters

<i>base</i>	WDOG peripheral base address
<i>windowValue</i>	WDOG window value.

### 36.6.13 static void WDOG\_Unlock ( WDOG\_Type \* *base* ) [inline], [static]

This function unlocks the WDOG register written. Before starting the unlock sequence and following configuration, disable the global interrupts. Otherwise, an interrupt may invalidate the unlocking sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

<i>base</i>	WDOG peripheral base address
-------------	------------------------------

### 36.6.14 void WDOG\_Refresh ( WDOG\_Type \* *base* )

This function feeds the WDOG. This function should be called before the WDOG timer is in timeout. Otherwise, a reset is asserted.

Parameters

<i>base</i>	WDOG peripheral base address
-------------	------------------------------

**36.6.15 static uint16\_t WDOG\_GetResetCount( WDOG\_Type \* *base* ) [inline],  
[static]**

This function gets the WDOG reset count value.

## Function Documentation

Parameters

<i>base</i>	WDOG peripheral base address
-------------	------------------------------

Returns

WDOG reset count value.

### 36.6.16 static void WDOG\_ClearResetCount( WDOG\_Type \* *base* ) [inline], [static]

This function clears the WDOG reset count value.

Parameters

<i>base</i>	WDOG peripheral base address
-------------	------------------------------

# Chapter 37

## Clock Driver

### 37.1 Overview

The MCUXpresso SDK provides APIs for MCUXpresso SDK devices' clock operation.

### 37.2 Get frequency

A centralized function `CLOCK_GetFreq` gets different clock type frequencies by passing a clock name. For example, pass a `kCLOCK_CoreSysClk` to get the core clock and pass a `kCLOCK_BusClk` to get the bus clock. Additionally, there are separate functions to get the frequency. For example, use `CLOCK_GetCoreSysClkFreq` to get the core clock frequency and `CLOCK_GetBusClkFreq` to get the bus clock frequency. Using these functions reduces the image size.

### 37.3 External clock frequency

The external clocks EXTAL0/EXTAL1/EXTAL32 are decided by the board level design. The Clock driver uses variables `g_xtal0Freq/g_xtal1Freq/g_xtal32Freq` to save clock frequencies. Likewise, the APIs `CLOCK_SetXtal0Freq`, `CLOCK_SetXtal1Freq`, and `CLOCK_SetXtal32Freq` are used to set these variables.

The upper layer must set these values correctly. For example, after `OSC0(SYSOSC)` is initialized using `CLOCK_InitOsc0` or `CLOCK_InitSysOsc`, the upper layer should call the `CLOCK_SetXtal0Freq`. Otherwise, the clock frequency get functions may not receive valid values. This is useful for multicore platforms where only one core calls `CLOCK_InitOsc0` to initialize `OSC0` and other cores call `CLOCK_SetXtal0Freq`.

## Modules

- Multipurpose Clock Generator (MCG)

## Files

- file `fsl_clock.h`

## Data Structures

- struct `sim_clock_config_t`  
*SIM configuration structure for clock setting.* [More...](#)
- struct `oscer_config_t`  
*OSC configuration for OSCERCLK.* [More...](#)
- struct `osc_config_t`  
*OSC Initialization Configuration Structure.* [More...](#)
- struct `mcg_pll_config_t`

## External clock frequency

*MCG PLL configuration.* [More...](#)

- struct **mcg\_config\_t**

*MCG mode change configuration structure.* [More...](#)

## Macros

- #define **MCG\_CONFIG\_CHECK\_PARAM\_0U**  
*Configures whether to check a parameter in a function.*
- #define **FSL\_SDK\_DISABLE\_DRIVER\_CLOCK\_CONTROL\_0**  
*Configure whether driver controls clock.*
- #define **DMAMUX\_CLOCKS**  
*Clock ip name array for DMAMUX.*
- #define **RTC\_CLOCKS**  
*Clock ip name array for RTC.*
- #define **SAI\_CLOCKS**  
*Clock ip name array for SAI.*
- #define **PORT\_CLOCKS**  
*Clock ip name array for PORT.*
- #define **FLEXBUS\_CLOCKS**  
*Clock ip name array for FLEXBUS.*
- #define **EWM\_CLOCKS**  
*Clock ip name array for EWM.*
- #define **PIT\_CLOCKS**  
*Clock ip name array for PIT.*
- #define **DSPI\_CLOCKS**  
*Clock ip name array for DSPI.*
- #define **LPTMR\_CLOCKS**  
*Clock ip name array for LPTMR.*
- #define **SDHC\_CLOCKS**  
*Clock ip name array for SDHC.*
- #define **FTM\_CLOCKS**  
*Clock ip name array for FTM.*
- #define **EDMA\_CLOCKS**  
*Clock ip name array for EDMA.*
- #define **FLEXCAN\_CLOCKS**  
*Clock ip name array for FLEXCAN.*
- #define **DAC\_CLOCKS**  
*Clock ip name array for DAC.*
- #define **ADC16\_CLOCKS**  
*Clock ip name array for ADC16.*
- #define **SYSMPU\_CLOCKS**  
*Clock ip name array for MPU.*
- #define **VREF\_CLOCKS**  
*Clock ip name array for VREF.*
- #define **CMT\_CLOCKS**  
*Clock ip name array for CMT.*
- #define **UART\_CLOCKS**  
*Clock ip name array for UART.*
- #define **RNGA\_CLOCKS**  
*Clock ip name array for RNGA.*
- #define **CRC\_CLOCKS**

- *Clock ip name array for CRC.*  
• #define **I2C\_CLOCKS**  
    *Clock ip name array for I2C.*
- #define **PDB\_CLOCKS**  
    *Clock ip name array for PDB.*
- #define **FTF\_CLOCKS**  
    *Clock ip name array for FTF.*
- #define **CMP\_CLOCKS**  
    *Clock ip name array for CMP.*
- #define **LPO\_CLK\_FREQ** 1000U  
    *LPO clock frequency.*
- #define **SYS\_CLK** kCLOCK\_CoreSysClk  
    *Peripherals clock source definition.*

## Enumerations

- enum **clock\_name\_t** {
 **kCLOCK\_CoreSysClk**,  
**kCLOCK\_PlatClk**,  
**kCLOCK\_BusClk**,  
**kCLOCK\_FlexBusClk**,  
**kCLOCK\_FlashClk**,  
**kCLOCK\_FastPeriphClk**,  
**kCLOCK\_PlIPlSelClk**,  
**kCLOCK\_Er32kClk**,  
**kCLOCK\_Osc0ErClk**,  
**kCLOCK\_Osc1ErClk**,  
**kCLOCK\_Osc0ErClkUndiv**,  
**kCLOCK\_McgFixedFreqClk**,  
**kCLOCK\_McgInternalRefClk**,  
**kCLOCK\_McgFllClk**,  
**kCLOCK\_McgPll0Clk**,  
**kCLOCK\_McgPll1Clk**,  
**kCLOCK\_McgExtPllClk**,  
**kCLOCK\_McgPeriphClk**,  
**kCLOCK\_McgIrc48MClk**,  
**kCLOCK\_LpoClk** }
- Clock name used to get clock frequency.*
- enum **clock\_usb\_src\_t** {
 **kCLOCK\_UsbSrcPll0** = SIM\_SOPT2\_USBSRC(1U) | SIM\_SOPT2\_PLLFLLSEL(1U),  
**kCLOCK\_UsbSrcExt** = SIM\_SOPT2\_USBSRC(0U) }  
    *USB clock source definition.*
- enum **clock\_ip\_name\_t**  
    *Clock gate name used for CLOCK\_EnableClock/CLOCK\_DisableClock.*
- enum **osc\_mode\_t** {
 **kOSC\_ModeExt** = 0U,  
**kOSC\_ModeOscLowPower** = MCG\_C2\_EREFS0\_MASK,  
**kOSC\_ModeOscHighGain** }

## External clock frequency

- *OSC work mode.*
  - enum `_osc_cap_load` {  
    `kOSC_Cap2P` = `OSC_CR_SC2P_MASK`,  
    `kOSC_Cap4P` = `OSC_CR_SC4P_MASK`,  
    `kOSC_Cap8P` = `OSC_CR_SC8P_MASK`,  
    `kOSC_Cap16P` = `OSC_CR_SC16P_MASK` }
- *Oscillator capacitor load setting.*
  - enum `_oscer_enable_mode` {  
    `kOSC_ErClkEnable` = `OSC_CR_ERCLKEN_MASK`,  
    `kOSC_ErClkEnableInStop` = `OSC_CR_EREFSTEN_MASK` }
- *OSCERCLK enable mode.*
  - enum `mcg_fll_src_t` {  
    `kMCG_FllSrcExternal`,  
    `kMCG_FllSrcInternal` }
- *MCG FLL reference clock source select.*
  - enum `mcg_irc_mode_t` {  
    `kMCG_IrcSlow`,  
    `kMCG_IrcFast` }
- *MCG internal reference clock select.*
  - enum `mcg_dmx32_t` {  
    `kMCG_Dmx32Default`,  
    `kMCG_Dmx32Fine` }
- *MCG DCO Maximum Frequency with 32.768 kHz Reference.*
  - enum `mcg_drs_t` {  
    `kMCG_DrsLow`,  
    `kMCG_DrsMid`,  
    `kMCG_DrsMidHigh`,  
    `kMCG_DrsHigh` }
- *MCG DCO range select.*
  - enum `mcg_pll_ref_src_t` {  
    `kMCG_PllRefOsc0`,  
    `kMCG_PllRefOsc1` }
- *MCG PLL reference clock select.*
  - enum `mcg_clkout_src_t` {  
    `kMCG_ClkOutSrcOut`,  
    `kMCG_ClkOutSrcInternal`,  
    `kMCG_ClkOutSrcExternal` }
- *MCGOUT clock source.*
  - enum `mcg_atm_select_t` {  
    `kMCG_AtmSel32k`,  
    `kMCG_AtmSel4m` }
- *MCG Automatic Trim Machine Select.*
  - enum `mcg_oscsel_t` {  
    `kMCG_OscselOsc`,  
    `kMCG_OscselRtc` }
- *MCG OSC Clock Select.*
  - enum `mcg_pll_clk_select_t` { `kMCG_PllClkSelPll0` }

- enum `mcg_monitor_mode_t` {
   
    `kMCG_MonitorNone`,
   
    `kMCG_MonitorInt`,
   
    `kMCG_MonitorReset` }
- MCG clock monitor mode.*
- enum `_mcg_status` {
   
    `kStatus_MCG_ModeUnreachable` = MAKE\_STATUS(kStatusGroup\_MCG, 0),
   
    `kStatus_MCG_ModeInvalid` = MAKE\_STATUS(kStatusGroup\_MCG, 1),
   
    `kStatus_MCG_AtmBusClockInvalid` = MAKE\_STATUS(kStatusGroup\_MCG, 2),
   
    `kStatus_MCG_AtmDesiredFreqInvalid` = MAKE\_STATUS(kStatusGroup\_MCG, 3),
   
    `kStatus_MCG_AtmIrcUsed` = MAKE\_STATUS(kStatusGroup\_MCG, 4),
   
    `kStatus_MCG_AtmHardwareFail` = MAKE\_STATUS(kStatusGroup\_MCG, 5),
   
    `kStatus_MCG_SourceUsed` = MAKE\_STATUS(kStatusGroup\_MCG, 6) }
- MCG status.*
- enum `_mcg_status_flags_t` {
   
    `kMCG_Osc0LostFlag` = (1U << 0U),
   
    `kMCG_Osc0InitFlag` = (1U << 1U),
   
    `kMCG_RtcOscLostFlag` = (1U << 4U),
   
    `kMCG_Pll0LostFlag` = (1U << 5U),
   
    `kMCG_Pll0LockFlag` = (1U << 6U) }
- MCG status flags.*
- enum `_mcg_irclk_enable_mode` {
   
    `kMCG_IrclkEnable` = MCG\_C1\_IRCLKEN\_MASK,
   
    `kMCG_IrclkEnableInStop` = MCG\_C1\_IREFSTEN\_MASK } *MCG internal reference clock (MCGIRCLK) enable mode definition.*
  - enum `_mcg_pll_enable_mode` {
   
    `kMCG_PllEnableIndependent` = MCG\_C5\_PLLCLKEN0\_MASK,
   
    `kMCG_PllEnableInStop` = MCG\_C5\_PLLSTEN0\_MASK } *MCG PLL clock enable mode definition.*
  - enum `mcg_mode_t` {
   
    `kMCG_ModeFEI` = 0U,
   
    `kMCG_ModeFBI`,
   
    `kMCG_ModeBLPI`,
   
    `kMCG_ModeFEE`,
   
    `kMCG_ModeFBE`,
   
    `kMCG_ModeBLPE`,
   
    `kMCG_ModePBE`,
   
    `kMCG_ModePEE`,
   
    `kMCG_ModeError` }
- MCG mode definitions.*

## Functions

- static void `CLOCK_EnableClock` (`clock_ip_name_t` name)
   
        *Enable the clock for specific IP.*
- static void `CLOCK_DisableClock` (`clock_ip_name_t` name)

## External clock frequency

*Disable the clock for specific IP.*

- static void **CLOCK\_SetSdhc0Clock** (uint32\_t src)  
*Set SDHC0 clock source.*
- static void **CLOCK\_SetEr32kClock** (uint32\_t src)  
*Set ERCLK32K source.*
- static void **CLOCK\_SetTraceClock** (uint32\_t src)  
*Set debug trace clock source.*
- static void **CLOCK\_SetPllFllSelClock** (uint32\_t src)  
*Set PLLFLLSEL clock source.*
- static void **CLOCK\_SetClkOutClock** (uint32\_t src)  
*Set CLKOUT source.*
- static void **CLOCK\_SetRtcClkOutClock** (uint32\_t src)  
*Set RTC\_CLKOUT source.*
- bool **CLOCK\_EnableUsbfs0Clock** (clock\_usb\_src\_t src, uint32\_t freq)  
*Enable USB FS clock.*
- static void **CLOCK\_DisableUsbfs0Clock** (void)  
*Disable USB FS clock.*
- static void **CLOCK\_SetOutDiv** (uint32\_t outdiv1, uint32\_t outdiv2, uint32\_t outdiv3, uint32\_t outdiv4)  
*System clock divider.*
- uint32\_t **CLOCK\_GetFreq** (clock\_name\_t clockName)  
*Gets the clock frequency for a specific clock name.*
- uint32\_t **CLOCK\_GetCoreSysClkFreq** (void)  
*Get the core clock or system clock frequency.*
- uint32\_t **CLOCK\_GetPlatClkFreq** (void)  
*Get the platform clock frequency.*
- uint32\_t **CLOCK\_GetBusClkFreq** (void)  
*Get the bus clock frequency.*
- uint32\_t **CLOCK\_GetFlexBusClkFreq** (void)  
*Get the flexbus clock frequency.*
- uint32\_t **CLOCK\_GetFlashClkFreq** (void)  
*Get the flash clock frequency.*
- uint32\_t **CLOCK\_GetPllFllSelClkFreq** (void)  
*Get the output clock frequency selected by SIM[PLLFLLSEL].*
- uint32\_t **CLOCK\_GetEr32kClkFreq** (void)  
*Get the external reference 32K clock frequency (ERCLK32K).*
- uint32\_t **CLOCK\_GetOsc0ErClkFreq** (void)  
*Get the OSC0 external reference clock frequency (OSC0ERCLK).*
- void **CLOCK\_SetSimConfig** (sim\_clock\_config\_t const \*config)  
*Set the clock configure in SIM module.*
- static void **CLOCK\_SetSimSafeDivs** (void)  
*Set the system clock dividers in SIM to safe value.*

## Variables

- uint32\_t **g\_xtal0Freq**  
*External XTAL0 (OSC0) clock frequency.*
- uint32\_t **g\_xtal32Freq**  
*External XTAL32/EXTAL32/RTC\_CLKIN clock frequency.*

## Driver version

- #define **FSL\_CLOCK\_DRIVER\_VERSION** (**MAKE\_VERSION**(2, 2, 1))  
*CLOCK driver version 2.2.1.*

## MCG frequency functions.

- **uint32\_t CLOCK\_GetOutClkFreq (void)**  
*Gets the MCG output clock (MCGOUTCLK) frequency.*
- **uint32\_t CLOCK\_GetFllFreq (void)**  
*Gets the MCG FLL clock (MCGFLLCLK) frequency.*
- **uint32\_t CLOCK\_GetInternalRefClkFreq (void)**  
*Gets the MCG internal reference clock (MCGIRCLK) frequency.*
- **uint32\_t CLOCK\_GetFixedFreqClkFreq (void)**  
*Gets the MCG fixed frequency clock (MCGFFCLK) frequency.*
- **uint32\_t CLOCK\_GetPll0Freq (void)**  
*Gets the MCG PLL0 clock (MCGPLL0CLK) frequency.*

## MCG clock configuration.

- **static void CLOCK\_SetLowPowerEnable (bool enable)**  
*Enables or disables the MCG low power.*
- **status\_t CLOCK\_SetInternalRefClkConfig (uint8\_t enableMode, mcg\_irc\_mode\_t ircs, uint8\_t fcrDiv)**  
*Configures the Internal Reference clock (MCGIRCLK).*
- **status\_t CLOCK\_SetExternalRefClkConfig (mcg\_oscsel\_t oscsel)**  
*Selects the MCG external reference clock.*
- **static void CLOCK\_SetFllExtRefDiv (uint8\_t frdiv)**  
*Set the FLL external reference clock divider value.*
- **void CLOCK\_EnablePll0 (mcg\_pll\_config\_t const \*config)**  
*Enables the PLL0 in FLL mode.*
- **static void CLOCK\_DisablePll0 (void)**  
*Disables the PLL0 in FLL mode.*
- **uint32\_t CLOCK\_CalcPllDiv (uint32\_t refFreq, uint32\_t desireFreq, uint8\_t \*prdiv, uint8\_t \*vdiv)**  
*Calculates the PLL divider setting for a desired output frequency.*

## MCG clock lock monitor functions.

- **void CLOCK\_SetOsc0MonitorMode (mcg\_monitor\_mode\_t mode)**  
*Sets the OSC0 clock monitor mode.*
- **void CLOCK\_SetRtcOscMonitorMode (mcg\_monitor\_mode\_t mode)**  
*Sets the RTC OSC clock monitor mode.*
- **void CLOCK\_SetPll0MonitorMode (mcg\_monitor\_mode\_t mode)**  
*Sets the PLL0 clock monitor mode.*
- **uint32\_t CLOCK\_GetStatusFlags (void)**  
*Gets the MCG status flags.*
- **void CLOCK\_ClearStatusFlags (uint32\_t mask)**  
*Clears the MCG status flags.*

## OSC configuration

- **static void OSC\_SetExtRefClkConfig (OSC\_Type \*base, oscer\_config\_t const \*config)**

## External clock frequency

*Configures the OSC external reference clock (OSCERCLK).*

- static void **OSC\_SetCapLoad** (OSC\_Type \*base, uint8\_t capLoad)  
*Sets the capacitor load configuration for the oscillator.*
- void **CLOCK\_InitOsc0** (osc\_config\_t const \*config)  
*Initializes the OSC0.*
- void **CLOCK\_DeinitOsc0** (void)  
*Deinitializes the OSC0.*

## External clock frequency

- static void **CLOCK\_SetXtal0Freq** (uint32\_t freq)  
*Sets the XTAL0 frequency based on board settings.*
- static void **CLOCK\_SetXtal32Freq** (uint32\_t freq)  
*Sets the XTAL32/RTC\_CLKIN frequency based on board settings.*

## MCG auto-trim machine.

- status\_t **CLOCK\_TrimInternalRefClk** (uint32\_t extFreq, uint32\_t desireFreq, uint32\_t \*actualFreq, mcg\_atm\_select\_t atms)  
*Auto trims the internal reference clock.*

## MCG mode functions.

- mcg\_mode\_t **CLOCK\_GetMode** (void)  
*Gets the current MCG mode.*
- status\_t **CLOCK\_SetFeiMode** (mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FEI mode.*
- status\_t **CLOCK\_SetFeeMode** (uint8\_t frdiv, mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FEE mode.*
- status\_t **CLOCK\_SetFbiMode** (mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FBI mode.*
- status\_t **CLOCK\_SetFbeMode** (uint8\_t frdiv, mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FBE mode.*
- status\_t **CLOCK\_SetBlpiMode** (void)  
*Sets the MCG to BLPI mode.*
- status\_t **CLOCK\_SetBlpeMode** (void)  
*Sets the MCG to BLPE mode.*
- status\_t **CLOCK\_SetPbeMode** (mcg\_pll\_clk\_select\_t pllcs, mcg\_pll\_config\_t const \*config)  
*Sets the MCG to PBE mode.*
- status\_t **CLOCK\_SetPeeMode** (void)  
*Sets the MCG to PEE mode.*
- status\_t **CLOCK\_ExternalModeToFbeModeQuick** (void)  
*Switches the MCG to FBE mode from the external mode.*
- status\_t **CLOCK\_InternalModeToFbiModeQuick** (void)  
*Switches the MCG to FBI mode from internal modes.*
- status\_t **CLOCK\_BootToFeiMode** (mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*fllStableDelay)(void))  
*Sets the MCG to FEI mode during system boot up.*

- status\_t **CLOCK\_BootToFeeMode** ([mcg\\_oscsel\\_t](#) oscsel, [uint8\\_t](#) frdiv, [mcg\\_dmx32\\_t](#) dmx32, [mcg\\_drs\\_t](#) drs, void(\*flStableDelay)(void))  
*Sets the MCG to FEE mode during system bootup.*
- status\_t **CLOCK\_BootToBlpiMode** ([uint8\\_t](#) fcrdiv, [mcg\\_irc\\_mode\\_t](#) ircs, [uint8\\_t](#) ircEnableMode)  
*Sets the MCG to BLPI mode during system boot up.*
- status\_t **CLOCK\_BootToBlpeMode** ([mcg\\_oscsel\\_t](#) oscsel)  
*Sets the MCG to BLPE mode during system boot up.*
- status\_t **CLOCK\_BootToPeeMode** ([mcg\\_oscsel\\_t](#) oscsel, [mcg\\_pll\\_clk\\_select\\_t](#) pllcs, [mcg\\_pll\\_config\\_t](#) const \*config)  
*Sets the MCG to PEE mode during system boot up.*
- status\_t **CLOCK\_SetMcgConfig** ([mcg\\_config\\_t](#) const \*config)  
*Sets the MCG to a target mode.*

## 37.4 Data Structure Documentation

### 37.4.1 struct sim\_clock\_config\_t

#### Data Fields

- [uint8\\_t](#) **pllFllSel**  
*PLL/FLL/IRC48M selection.*
- [uint8\\_t](#) **er32kSrc**  
*ERCLK32K source selection.*
- [uint32\\_t](#) **clkdiv1**  
*SIM\_CLKDIV1.*

#### 37.4.1.0.0.79 Field Documentation

##### 37.4.1.0.0.79.1 [uint8\\_t sim\\_clock\\_config\\_t::pllFllSel](#)

##### 37.4.1.0.0.79.2 [uint8\\_t sim\\_clock\\_config\\_t::er32kSrc](#)

##### 37.4.1.0.0.79.3 [uint32\\_t sim\\_clock\\_config\\_t::clkdiv1](#)

### 37.4.2 struct oscer\_config\_t

#### Data Fields

- [uint8\\_t](#) **enableMode**  
*OSCERCLK enable mode.*

#### 37.4.2.0.0.80 Field Documentation

##### 37.4.2.0.0.80.1 [uint8\\_t oscer\\_config\\_t::enableMode](#)

OR'ed value of [\\_oscer\\_enable\\_mode](#).

## Data Structure Documentation

### 37.4.3 `struct osc_config_t`

Defines the configuration data structure to initialize the OSC. When porting to a new board, set the following members according to the board setting:

1. freq: The external frequency.
2. workMode: The OSC module mode.

#### Data Fields

- `uint32_t freq`  
*External clock frequency.*
- `uint8_t capLoad`  
*Capacitor load setting.*
- `osc_mode_t workMode`  
*OSC work mode setting.*
- `oscer_config_t oscerConfig`  
*Configuration for OSCERCLK.*

#### 37.4.3.0.0.81 Field Documentation

##### 37.4.3.0.0.81.1 `uint32_t osc_config_t::freq`

##### 37.4.3.0.0.81.2 `uint8_t osc_config_t::capLoad`

##### 37.4.3.0.0.81.3 `osc_mode_t osc_config_t::workMode`

##### 37.4.3.0.0.81.4 `oscer_config_t osc_config_t::oscerConfig`

### 37.4.4 `struct mcg_pll_config_t`

#### Data Fields

- `uint8_t enableMode`  
*Enable mode.*
- `uint8_t prdiv`  
*Reference divider PRDIV.*
- `uint8_t vdiv`  
*VCO divider VDIV.*

#### 37.4.4.0.0.82 Field Documentation

##### 37.4.4.0.0.82.1 `uint8_t mcg_pll_config_t::enableMode`

OR'ed value of `_mcg_pll_enable_mode`.

**37.4.4.0.0.82.2 uint8\_t mcg\_pll\_config\_t::prdiv**

**37.4.4.0.0.82.3 uint8\_t mcg\_pll\_config\_t::vdiv**

### 37.4.5 struct mcg\_config\_t

When porting to a new board, set the following members according to the board setting:

1. frdiv: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by frdiv is in the 31.25 kHz to 39.0625 kHz range.
2. The PLL reference clock divider PRDIV: PLL reference clock frequency after PRDIV should be in the FSL\_FEATURE\_MCG\_PLL\_REF\_MIN to FSL\_FEATURE\_MCG\_PLL\_REF\_MAX range.

### Data Fields

- **mcg\_mode\_t mcgMode**  
*MCG mode.*
- **uint8\_t irclkEnableMode**  
*MCGIRCLK enable mode.*
- **mcg\_irc\_mode\_t ircs**  
*Source, MCG\_C2[IRCS].*
- **uint8\_t fcrdiv**  
*Divider, MCG\_SC[FCRDIV].*
- **uint8\_t frdiv**  
*Divider MCG\_C1[FRDIV].*
- **mcg\_drs\_t drs**  
*DCO range MCG\_C4[DRST\_DRS].*
- **mcg\_dmx32\_t dmx32**  
*MCG\_C4[DMX32].*
- **mcg\_oscsel\_t oscsel**  
*OSC select MCG\_C7[OSCSEL].*
- **mcg\_pll\_config\_t pll0Config**  
*MCGPLL0CLK configuration.*

## Macro Definition Documentation

### 37.4.5.0.0.83 Field Documentation

37.4.5.0.0.83.1 `mcg_mode_t mcg_config_t::mcgMode`

37.4.5.0.0.83.2 `uint8_t mcg_config_t::irclkEnableMode`

37.4.5.0.0.83.3 `mcg_irc_mode_t mcg_config_t::ircs`

37.4.5.0.0.83.4 `uint8_t mcg_config_t::fcrdiv`

37.4.5.0.0.83.5 `uint8_t mcg_config_t::frdiv`

37.4.5.0.0.83.6 `mcg_drs_t mcg_config_t::drs`

37.4.5.0.0.83.7 `mcg_dmx32_t mcg_config_t::dmx32`

37.4.5.0.0.83.8 `mcg_oscsel_t mcg_config_t::oscsel`

37.4.5.0.0.83.9 `mcg_pll_config_t mcg_config_t::pll0Config`

## 37.5 Macro Definition Documentation

### 37.5.1 #define MCG\_CONFIG\_CHECK\_PARAM 0U

Some MCG settings must be changed with conditions, for example:

1. MCGIRCLK settings, such as the source, divider, and the trim value should not change when MC-GIRCLK is used as a system clock source.
2. MCG\_C7[OSCSEL] should not be changed when the external reference clock is used as a system clock source. For example, in FBE/BLPE/PBE modes.
3. The users should only switch between the supported clock modes.

MCG functions check the parameter and MCG status before setting, if not allowed to change, the functions return error. The parameter checking increases code size, if code size is a critical requirement, change [MCG\\_CONFIG\\_CHECK\\_PARAM](#) to 0 to disable parameter checking.

### 37.5.2 #define FSL\_SDK\_DISABLE\_DRIVER\_CLOCK\_CONTROL 0

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note

All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

**37.5.3 #define FSL\_CLOCK\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 1))****37.5.4 #define DMAMUX\_CLOCKS****Value:**{  
 kCLOCK\_Dmamux0 \  
}**37.5.5 #define RTC\_CLOCKS****Value:**{  
 kCLOCK\_Rtc0 \  
}**37.5.6 #define SAI\_CLOCKS****Value:**{  
 kCLOCK\_Sai0 \  
}**37.5.7 #define PORT\_CLOCKS****Value:**{  
 kCLOCK\_PortA, kCLOCK\_PortB, kCLOCK\_PortC, kCLOCK\_PortD, kCLOCK\_PortE \  
}**37.5.8 #define FLEXBUS\_CLOCKS****Value:**{  
 kCLOCK\_Flexbus0 \  
}

## Macro Definition Documentation

### 37.5.9 #define EWM\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Ewm0 \
}
```

### 37.5.10 #define PIT\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Pit0 \
}
```

### 37.5.11 #define DSPI\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Spi0, kCLOCK_Spi1, kCLOCK_Spi2 \
}
```

### 37.5.12 #define LPTMR\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Lptmr0 \
}
```

### 37.5.13 #define SDHC\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Sdhc0 \
}
```

### 37.5.14 #define FTM\_CLOCKS

**Value:**

```
{  
    kCLOCK_Ftm0, kCLOCK_Ftm1, kCLOCK_Ftm2, kCLOCK_Ftm3 \  
}
```

### 37.5.15 #define EDMA\_CLOCKS

**Value:**

```
{  
    kCLOCK_Dma0 \  
}
```

### 37.5.16 #define FLEXCAN\_CLOCKS

**Value:**

```
{  
    kCLOCK_Flexcan0 \  
}
```

### 37.5.17 #define DAC\_CLOCKS

**Value:**

```
{  
    kCLOCK_Dac0, kCLOCK_Dac1 \  
}
```

### 37.5.18 #define ADC16\_CLOCKS

**Value:**

```
{  
    kCLOCK_Adc0, kCLOCK_Adc1 \  
}
```

## Macro Definition Documentation

### 37.5.19 #define SYSMPU\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Sysmpu0 \
}
```

### 37.5.20 #define VREF\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Vref0 \
}
```

### 37.5.21 #define CMT\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Cmt0 \
}
```

### 37.5.22 #define UART\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Uart0, kCLOCK_Uart1, kCLOCK_Uart2, kCLOCK_Uart3, kCLOCK_Uart4, kCLOCK_Uart5 \
}
```

### 37.5.23 #define RNGA\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Rnga0 \
}
```

### 37.5.24 #define CRC\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Crc0 \
}
```

### 37.5.25 #define I2C\_CLOCKS

**Value:**

```
{           \
    kCLOCK_I2c0, kCLOCK_I2c1, kCLOCK_I2c2 \
}
```

### 37.5.26 #define PDB\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Pdb0 \
}
```

### 37.5.27 #define FTF\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Ftf0 \
}
```

### 37.5.28 #define CMP\_CLOCKS

**Value:**

```
{           \
    kCLOCK_Cmp0, kCLOCK_Cmp1, kCLOCK_Cmp2 \
}
```

## Enumeration Type Documentation

### 37.5.29 #define SYS\_CLK kCLOCK\_CoreSysClk

## 37.6 Enumeration Type Documentation

### 37.6.1 enum clock\_name\_t

Enumerator

*kCLOCK\_CoreSysClk* Core/system clock.  
*kCLOCK\_PlatClk* Platform clock.  
*kCLOCK\_BusClk* Bus clock.  
*kCLOCK\_FlexBusClk* FlexBus clock.  
*kCLOCK\_FlashClk* Flash clock.  
*kCLOCK\_FastPeriphClk* Fast peripheral clock.  
*kCLOCK\_PllFllSelClk* The clock after SIM[PLLFLSEL].  
*kCLOCK\_Er32kClk* External reference 32K clock (ERCLK32K)  
*kCLOCK\_Osc0ErClk* OSC0 external reference clock (OSC0ERCLK)  
*kCLOCK\_Osc1ErClk* OSC1 external reference clock (OSC1ERCLK)  
*kCLOCK\_Osc0ErClkUndiv* OSC0 external reference undivided clock(OSC0ERCLK\_UNDIV).  
*kCLOCK\_McgFixedFreqClk* MCG fixed frequency clock (MCGFFCLK)  
*kCLOCK\_McgInternalRefClk* MCG internal reference clock (MCGIRCLK)  
*kCLOCK\_McgFllClk* MCGFLLCLK.  
*kCLOCK\_McgPll0Clk* MCGPLL0CLK.  
*kCLOCK\_McgPll1Clk* MCGPLL1CLK.  
*kCLOCK\_McgExtPllClk* EXT\_PLLCLK.  
*kCLOCK\_McgPeriphClk* MCG peripheral clock (MCGPCLK)  
*kCLOCK\_McgIrc48MClk* MCG IRC48M clock.  
*kCLOCK\_LpoClk* LPO clock.

### 37.6.2 enum clock\_usb\_src\_t

Enumerator

*kCLOCK\_UsbSrcPll0* Use PLL0.  
*kCLOCK\_UsbSrcExt* Use USB\_CLKIN.

### 37.6.3 enum clock\_ip\_name\_t

### 37.6.4 enum osc\_mode\_t

Enumerator

*kOSC\_ModeExt* Use an external clock.

***kOSC\_ModeOscLowPower*** Oscillator low power.

***kOSC\_ModeOscHighGain*** Oscillator high gain.

### 37.6.5 enum \_osc\_cap\_load

Enumerator

***kOSC\_Cap2P*** 2 pF capacitor load

***kOSC\_Cap4P*** 4 pF capacitor load

***kOSC\_Cap8P*** 8 pF capacitor load

***kOSC\_Cap16P*** 16 pF capacitor load

### 37.6.6 enum \_oscer\_enable\_mode

Enumerator

***kOSC\_ErClkEnable*** Enable.

***kOSC\_ErClkEnableInStop*** Enable in stop mode.

### 37.6.7 enum mcg\_fll\_src\_t

Enumerator

***kMCG\_FllSrcExternal*** External reference clock is selected.

***kMCG\_FllSrcInternal*** The slow internal reference clock is selected.

### 37.6.8 enum mcg\_irc\_mode\_t

Enumerator

***kMCG\_IrcSlow*** Slow internal reference clock selected.

***kMCG\_IrcFast*** Fast internal reference clock selected.

### 37.6.9 enum mcg\_dmx32\_t

Enumerator

***kMCG\_Dmx32Default*** DCO has a default range of 25%.

***kMCG\_Dmx32Fine*** DCO is fine-tuned for maximum frequency with 32.768 kHz reference.

## Enumeration Type Documentation

### 37.6.10 enum mcg\_drs\_t

Enumerator

*kMCG\_DrsLow* Low frequency range.

*kMCG\_DrsMid* Mid frequency range.

*kMCG\_DrsMidHigh* Mid-High frequency range.

*kMCG\_DrsHigh* High frequency range.

### 37.6.11 enum mcg\_pll\_ref\_src\_t

Enumerator

*kMCG\_PllRefOsc0* Selects OSC0 as PLL reference clock.

*kMCG\_PllRefOsc1* Selects OSC1 as PLL reference clock.

### 37.6.12 enum mcg\_clkout\_src\_t

Enumerator

*kMCG\_ClkOutSrcOut* Output of the FLL is selected (reset default)

*kMCG\_ClkOutSrcInternal* Internal reference clock is selected.

*kMCG\_ClkOutSrcExternal* External reference clock is selected.

### 37.6.13 enum mcg\_atm\_select\_t

Enumerator

*kMCG\_AtmSel32k* 32 kHz Internal Reference Clock selected

*kMCG\_AtmSel4m* 4 MHz Internal Reference Clock selected

### 37.6.14 enum mcg\_oscsel\_t

Enumerator

*kMCG\_OscselOsc* Selects System Oscillator (OSCCLK)

*kMCG\_OscselRtc* Selects 32 kHz RTC Oscillator.

**37.6.15 enum mcg\_pll\_clk\_select\_t**

Enumerator

*kMCG\_PllClkSelPll0* PLL0 output clock is selected.

**37.6.16 enum mcg\_monitor\_mode\_t**

Enumerator

*kMCG\_MonitorNone* Clock monitor is disabled.

*kMCG\_MonitorInt* Trigger interrupt when clock lost.

*kMCG\_MonitorReset* System reset when clock lost.

**37.6.17 enum \_mcg\_status**

Enumerator

*kStatus\_MCG\_ModeUnreachable* Can't switch to target mode.

*kStatus\_MCG\_ModeInvalid* Current mode invalid for the specific function.

*kStatus\_MCG\_AtmBusClockInvalid* Invalid bus clock for ATM.

*kStatus\_MCG\_AtmDesiredFreqInvalid* Invalid desired frequency for ATM.

*kStatus\_MCG\_AtmIrcUsed* IRC is used when using ATM.

*kStatus\_MCG\_AtmHardwareFail* Hardware fail occurs during ATM.

*kStatus\_MCG\_SourceUsed* Can't change the clock source because it is in use.

**37.6.18 enum \_mcg\_status\_flags\_t**

Enumerator

*kMCG\_Osc0LostFlag* OSC0 lost.

*kMCG\_Osc0InitFlag* OSC0 crystal initialized.

*kMCG\_RtcOscLostFlag* RTC OSC lost.

*kMCG\_Pll0LostFlag* PLL0 lost.

*kMCG\_Pll0LockFlag* PLL0 locked.

**37.6.19 enum \_mcg\_irclk\_enable\_mode**

Enumerator

*kMCG\_IrclkEnable* MCGIRCLK enable.

*kMCG\_IrclkEnableInStop* MCGIRCLK enable in stop mode.

## Function Documentation

### 37.6.20 enum \_mcg\_pll\_enable\_mode

Enumerator

*kMCG\_PlEnableIndependent* MCGPLLCLK enable independent of the MCG clock mode. Generally, the PLL is disabled in FLL modes (FEI/FBI/FEE/FBE). Setting the PLL clock enable independent, enables the PLL in the FLL modes.

*kMCG\_PlEnableInStop* MCGPLLCLK enable in STOP mode.

### 37.6.21 enum mcg\_mode\_t

Enumerator

*kMCG\_ModeFEI* FEI - FLL Engaged Internal.

*kMCG\_ModeFBI* FBI - FLL Bypassed Internal.

*kMCG\_ModeBLPI* BLPI - Bypassed Low Power Internal.

*kMCG\_ModeFEE* FEE - FLL Engaged External.

*kMCG\_ModeFBE* FBE - FLL Bypassed External.

*kMCG\_ModeBLPE* BLPE - Bypassed Low Power External.

*kMCG\_ModePBE* PBE - PLL Bypassed External.

*kMCG\_ModePEE* PEE - PLL Engaged External.

*kMCG\_ModeError* Unknown mode.

## 37.7 Function Documentation

### 37.7.1 static void CLOCK\_EnableClock ( *clock\_ip\_name\_t name* ) [inline], [static]

Parameters

<i>name</i>	Which clock to enable, see <a href="#">clock_ip_name_t</a> .
-------------	--

### 37.7.2 static void CLOCK\_DisableClock ( *clock\_ip\_name\_t name* ) [inline], [static]

Parameters

<i>name</i>	Which clock to disable, see <a href="#">clock_ip_name_t</a> .
-------------	---

### 37.7.3 static void CLOCK\_SetSdhc0Clock ( uint32\_t *src* ) [inline], [static]

Parameters

<i>src</i>	The value to set SDHC0 clock source.
------------	--------------------------------------

### 37.7.4 static void CLOCK\_SetEr32kClock ( uint32\_t *src* ) [inline], [static]

Parameters

<i>src</i>	The value to set ERCLK32K clock source.
------------	---

### 37.7.5 static void CLOCK\_SetTraceClock ( uint32\_t *src* ) [inline], [static]

Parameters

<i>src</i>	The value to set debug trace clock source.
------------	--

### 37.7.6 static void CLOCK\_SetPllfllSelClock ( uint32\_t *src* ) [inline], [static]

Parameters

<i>src</i>	The value to set PLLFLLSEL clock source.
------------	--

### 37.7.7 static void CLOCK\_SetClkOutClock ( uint32\_t *src* ) [inline], [static]

Parameters

## Function Documentation

<i>src</i>	The value to set CLKOUT source.
------------	---------------------------------

### 37.7.8 static void CLOCK\_SetRtcClkOutClock ( uint32\_t *src* ) [inline], [static]

Parameters

<i>src</i>	The value to set RTC_CLKOUT source.
------------	-------------------------------------

### 37.7.9 bool CLOCK\_EnableUsbfs0Clock ( clock\_usb\_src\_t *src*, uint32\_t *freq* )

Parameters

<i>src</i>	USB FS clock source.
<i>freq</i>	The frequency specified by src.

Return values

<i>true</i>	The clock is set successfully.
<i>false</i>	The clock source is invalid to get proper USB FS clock.

### 37.7.10 static void CLOCK\_DisableUsbfs0Clock ( void ) [inline], [static]

Disable USB FS clock.

### 37.7.11 static void CLOCK\_SetOutDiv ( uint32\_t *outdiv1*, uint32\_t *outdiv2*, uint32\_t *outdiv3*, uint32\_t *outdiv4* ) [inline], [static]

Set the SIM\_CLKDIV1[OUTDIV1], SIM\_CLKDIV1[OUTDIV2], SIM\_CLKDIV1[OUTDIV3], SIM\_CLKDIV1[OUTDIV4].

Parameters

<i>outdiv1</i>	Clock 1 output divider value.
<i>outdiv2</i>	Clock 2 output divider value.
<i>outdiv3</i>	Clock 3 output divider value.
<i>outdiv4</i>	Clock 4 output divider value.

**37.7.12 uint32\_t CLOCK\_GetFreq ( clock\_name\_t *clockName* )**

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in `clock_name_t`. The MCG must be properly configured before using this function.

Parameters

<i>clockName</i>	Clock names defined in <code>clock_name_t</code>
------------------	--

Returns

Clock frequency value in Hertz

**37.7.13 uint32\_t CLOCK\_GetCoreSysClkFreq ( void )**

Returns

Clock frequency in Hz.

**37.7.14 uint32\_t CLOCK\_GetPlatClkFreq ( void )**

Returns

Clock frequency in Hz.

**37.7.15 uint32\_t CLOCK\_GetBusClkFreq ( void )**

Returns

Clock frequency in Hz.

## Function Documentation

### 37.7.16 `uint32_t CLOCK_GetFlexBusClkFreq( void )`

Returns

Clock frequency in Hz.

### 37.7.17 `uint32_t CLOCK_GetFlashClkFreq( void )`

Returns

Clock frequency in Hz.

### 37.7.18 `uint32_t CLOCK_GetPllFllSelClkFreq( void )`

Returns

Clock frequency in Hz.

### 37.7.19 `uint32_t CLOCK_GetEr32kClkFreq( void )`

Returns

Clock frequency in Hz.

### 37.7.20 `uint32_t CLOCK_GetOsc0ErClkFreq( void )`

Returns

Clock frequency in Hz.

### 37.7.21 `void CLOCK_SetSimConfig( sim_clock_config_t const * config )`

This function sets system layer clock settings in SIM module.

Parameters

<i>config</i>	Pointer to the configure structure.
---------------	-------------------------------------

### 37.7.22 static void CLOCK\_SetSimSafeDivs ( void ) [inline], [static]

The system level clocks (core clock, bus clock, flexbus clock and flash clock) must be in allowed ranges. During MCG clock mode switch, the MCG output clock changes then the system level clocks may be out of range. This function could be used before MCG mode change, to make sure system level clocks are in allowed range.

Parameters

<i>config</i>	Pointer to the configure structure.
---------------	-------------------------------------

### 37.7.23 uint32\_t CLOCK\_GetOutClkFreq ( void )

This function gets the MCG output clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGOUTCLK.

### 37.7.24 uint32\_t CLOCK\_GetFllFreq ( void )

This function gets the MCG FLL clock frequency in Hz based on the current MCG register value. The FLL is enabled in FEI/FBI/FEE/FBE mode and disabled in low power state in other modes.

Returns

The frequency of MCGFLLCLK.

### 37.7.25 uint32\_t CLOCK\_GetInternalRefClkFreq ( void )

This function gets the MCG internal reference clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGIRCLK.

## Function Documentation

### 37.7.26 `uint32_t CLOCK_GetFixedFreqClkFreq ( void )`

This function gets the MCG fixed frequency clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGFFCLK.

### 37.7.27 `uint32_t CLOCK_GetPll0Freq ( void )`

This function gets the MCG PLL0 clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGPLL0CLK.

### 37.7.28 `static void CLOCK_SetLowPowerEnable ( bool enable ) [inline], [static]`

Enabling the MCG low power disables the PLL and FLL in bypass modes. In other words, in FBE and PBE modes, enabling low power sets the MCG to BLPE mode. In FBI and PBI modes, enabling low power sets the MCG to BLPI mode. When disabling the MCG low power, the PLL or FLL are enabled based on MCG settings.

Parameters

<i>enable</i>	True to enable MCG low power, false to disable MCG low power.
---------------	---

### 37.7.29 `status_t CLOCK_SetInternalRefClkConfig ( uint8_t enableMode, mcg_irc_mode_t ircs, uint8_t fcrdiv )`

This function sets the MCGIRCLK base on parameters. It also selects the IRC source. If the fast IRC is used, this function sets the fast IRC divider. This function also sets whether the MCGIRCLK is enabled in stop mode. Calling this function in FBI/PBI/BLPI modes may change the system clock. As a result, using the function in these modes it is not allowed.

Parameters

<i>enableMode</i>	MCGIRCLK enable mode, OR'ed value of <a href="#">_mcg_irclk_enable_mode</a> .
<i>ircs</i>	MCGIRCLK clock source, choose fast or slow.
<i>fcrdiv</i>	Fast IRC divider setting (FCRDIV).

Return values

<i>kStatus_MCG_SourceUsed</i>	Because the internal reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.
<i>kStatus_Success</i>	MCGIRCLK configuration finished successfully.

### 37.7.30 **status\_t CLOCK\_SetExternalRefClkConfig ( mcg\_oscsel\_t oscsel )**

Selects the MCG external reference clock source, changes the MCG\_C7[OSCSEL], and waits for the clock source to be stable. Because the external reference clock should not be changed in FEE/FBE/BLP-E/PBE/PEE modes, do not call this function in these modes.

Parameters

<i>oscsel</i>	MCG external reference clock source, MCG_C7[OSCSEL].
---------------	--

Return values

<i>kStatus_MCG_SourceUsed</i>	Because the external reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.
<i>kStatus_Success</i>	External reference clock set successfully.

### 37.7.31 **static void CLOCK\_SetFllExtRefDiv ( uint8\_t frdiv ) [inline], [static]**

Sets the FLL external reference clock divider value, the register MCG\_C1[FRDIV].

Parameters

<i>frdiv</i>	The FLL external reference clock divider value, MCG_C1[FRDIV].
--------------	--

## Function Documentation

### 37.7.32 void CLOCK\_EnablePll0 ( mcg\_pll\_config\_t const \* config )

This function sets up the PLL0 in FLL mode and reconfigures the PLL0. Ensure that the PLL reference clock is enabled before calling this function and that the PLL0 is not used as a clock source. The function CLOCK\_CalcPllDiv gets the correct PLL divider values.

Parameters

<i>config</i>	Pointer to the configuration structure.
---------------	---

### 37.7.33 static void CLOCK\_DisablePll0 ( void ) [inline], [static]

This function disables the PLL0 in FLL mode. It should be used together with the [CLOCK\\_EnablePll0](#).

### 37.7.34 uint32\_t CLOCK\_CalcPllDiv ( uint32\_t refFreq, uint32\_t desireFreq, uint8\_t \* prdiv, uint8\_t \* vdiv )

This function calculates the correct reference clock divider (PRDIV) and VCO divider (VDIV) to generate a desired PLL output frequency. It returns the closest frequency match with the corresponding PRDIV/-VDIV returned from parameters. If a desired frequency is not valid, this function returns 0.

Parameters

<i>refFreq</i>	PLL reference clock frequency.
<i>desireFreq</i>	Desired PLL output frequency.
<i>prdiv</i>	PRDIV value to generate desired PLL frequency.
<i>vdiv</i>	VDIV value to generate desired PLL frequency.

Returns

Closest frequency match that the PLL was able to generate.

### 37.7.35 void CLOCK\_SetOsc0MonitorMode ( mcg\_monitor\_mode\_t mode )

This function sets the OSC0 clock monitor mode. See [mcg\\_monitor\\_mode\\_t](#) for details.

Parameters

<i>mode</i>	Monitor mode to set.
-------------	----------------------

### 37.7.36 void CLOCK\_SetRtcOscMonitorMode ( mcg\_monitor\_mode\_t mode )

This function sets the RTC OSC clock monitor mode. See [mcg\\_monitor\\_mode\\_t](#) for details.

Parameters

<i>mode</i>	Monitor mode to set.
-------------	----------------------

### 37.7.37 void CLOCK\_SetPll0MonitorMode ( mcg\_monitor\_mode\_t mode )

This function sets the PLL0 clock monitor mode. See [mcg\\_monitor\\_mode\\_t](#) for details.

Parameters

<i>mode</i>	Monitor mode to set.
-------------	----------------------

### 37.7.38 uint32\_t CLOCK\_GetStatusFlags ( void )

This function gets the MCG clock status flags. All status flags are returned as a logical OR of the enumeration [\\_mcg\\_status\\_flags\\_t](#). To check a specific flag, compare the return value with the flag.

Example:

```
// To check the clock lost lock status of OSC0 and PLL0.
uint32_t mcgFlags;

mcgFlags = CLOCK_GetStatusFlags();

if (mcgFlags & kMCG_Osc0LostFlag)
{
    // OSC0 clock lock lost. Do something.
}
if (mcgFlags & kMCG_Pll0LostFlag)
{
    // PLL0 clock lock lost. Do something.
}
```

Returns

Logical OR value of the [\\_mcg\\_status\\_flags\\_t](#).

## Function Documentation

### 37.7.39 void CLOCK\_ClearStatusFlags ( uint32\_t mask )

This function clears the MCG clock lock lost status. The parameter is a logical OR value of the flags to clear. See [\\_mcg\\_status\\_flags\\_t](#).

Example:

```
// To clear the clock lock lost status flags of OSC0 and PLL0.  
CLOCK_ClearStatusFlags(kMCG_Osc0LostFlag | kMCG_Pll0LostFlag);
```

Parameters

<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration <a href="#">_mcg_status_flags_t</a> .
-------------	---

### 37.7.40 static void OSC\_SetExtRefClkConfig ( OSC\_Type \* *base*, oscer\_config\_t const \* *config* ) [inline], [static]

This function configures the OSC external reference clock (OSCERCLK). This is an example to enable the OSCERCLK in normal and stop modes and also set the output divider to 1:

```
oscer_config_t config =  
{  
    .enableMode = kOSC_ErClkEnable |  
        kOSC_ErClkEnableInStop,  
    .erclkDiv   = 1U,  
};  
  
OSC_SetExtRefClkConfig(OSC, &config);
```

Parameters

<i>base</i>	OSC peripheral address.
<i>config</i>	Pointer to the configuration structure.

### 37.7.41 static void OSC\_SetCapLoad ( OSC\_Type \* *base*, uint8\_t *capLoad* ) [inline], [static]

This function sets the specified capacitors configuration for the oscillator. This should be done in the early system level initialization function call based on the system configuration.

Parameters

<i>base</i>	OSC peripheral address.
<i>capLoad</i>	OR'ed value for the capacitor load option, see <a href="#">_osc_cap_load</a> .

Example:

```
// To enable only 2 pF and 8 pF capacitor load, please use like this.
OSC_SetCapLoad(OSC, kOSC_Cap2P | kOSC_Cap8P);
```

### 37.7.42 void CLOCK\_InitOsc0 ( osc\_config\_t const \* config )

This function initializes the OSC0 according to the board configuration.

Parameters

<i>config</i>	Pointer to the OSC0 configuration structure.
---------------	--

### 37.7.43 void CLOCK\_DeinitOsc0 ( void )

This function deinitializes the OSC0.

### 37.7.44 static void CLOCK\_SetXtal0Freq ( uint32\_t freq ) [inline], [static]

Parameters

<i>freq</i>	The XTAL0/EXTAL0 input clock frequency in Hz.
-------------	---

### 37.7.45 static void CLOCK\_SetXtal32Freq ( uint32\_t freq ) [inline], [static]

Parameters

<i>freq</i>	The XTAL32/EXTAL32/RTC_CLKIN input clock frequency in Hz.
-------------	---

### 37.7.46 status\_t CLOCK\_TrimInternalRefClk ( uint32\_t extFreq, uint32\_t desireFreq, uint32\_t \* actualFreq, mcg\_atm\_select\_t atms )

This function trims the internal reference clock by using the external clock. If successful, it returns the kStatus\_Success and the frequency after trimming is received in the parameter actualFreq. If an error

## Function Documentation

occurs, the error code is returned.

Parameters

<i>extFreq</i>	External clock frequency, which should be a bus clock.
<i>desireFreq</i>	Frequency to trim to.
<i>actualFreq</i>	Actual frequency after trimming.
<i>atms</i>	Trim fast or slow internal reference clock.

Return values

<i>kStatus_Success</i>	ATM success.
<i>kStatus_MCG_AtmBus-ClockInvalid</i>	The bus clock is not in allowed range for the ATM.
<i>kStatus_MCG_Atm-DesiredFreqInvalid</i>	MCGIRCLK could not be trimmed to the desired frequency.
<i>kStatus_MCG_AtmIrc-Used</i>	Could not trim because MCGIRCLK is used as a bus clock source.
<i>kStatus_MCG_Atm-HardwareFail</i>	Hardware fails while trimming.

### 37.7.47 **mcg\_mode\_t CLOCK\_GetMode( void )**

This function checks the MCG registers and determines the current MCG mode.

Returns

Current MCG mode or error code; See [mcg\\_mode\\_t](#).

### 37.7.48 **status\_t CLOCK\_SetFeiMode( mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*)(void) fllStableDelay )**

This function sets the MCG to FEI mode. If setting to FEI mode fails from the current mode, this function returns an error.

Parameters

## Function Documentation

<i>dmx32</i>	DMX32 in FEI mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to ensure that the FLL is stable. Passing NULL does not cause a delay.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

Note

If *dmx32* is set to kMCG\_Dmx32Fine, the slow IRC must not be trimmed to a frequency above 32768 Hz.

### 37.7.49 **status\_t CLOCK\_SetFeeMode ( uint8\_t *frdiv*, mcg\_dmx32\_t *dmx32*, mcg\_drs\_t *drs*, void(\*)(void) *fllStableDelay* )**

This function sets the MCG to FEE mode. If setting to FEE mode fails from the current mode, this function returns an error.

Parameters

<i>frdiv</i>	FLL reference clock divider setting, FRDIV.
<i>dmx32</i>	DMX32 in FEE mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to make sure FLL is stable. Passing NULL does not cause a delay.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

### 37.7.50 **status\_t CLOCK\_SetFbiMode ( mcg\_dmx32\_t *dmx32*, mcg\_drs\_t *drs*, void(\*)(void) *fllStableDelay* )**

This function sets the MCG to FBI mode. If setting to FBI mode fails from the current mode, this function returns an error.

## Parameters

<i>dmx32</i>	DMX32 in FBI mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to make sure FLL is stable. If the FLL is not used in FBI mode, this parameter can be NULL. Passing NULL does not cause a delay.

## Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

## Note

If *dmx32* is set to kMCG\_Dmx32Fine, the slow IRC must not be trimmed to frequency above 32768 Hz.

### 37.7.51 **status\_t CLOCK\_SetFbeMode ( uint8\_t *frdiv*, mcg\_dmx32\_t *dmx32*, mcg\_drs\_t *drs*, void(\*)(void) *fllStableDelay* )**

This function sets the MCG to FBE mode. If setting to FBE mode fails from the current mode, this function returns an error.

## Parameters

<i>frdiv</i>	FLL reference clock divider setting, FRDIV.
<i>dmx32</i>	DMX32 in FBE mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to make sure FLL is stable. If the FLL is not used in FBE mode, this parameter can be NULL. Passing NULL does not cause a delay.

## Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
-------------------------------------	--------------------------------------

## Function Documentation

<i>kStatus_Success</i>	Switched to the target mode successfully.
------------------------	---

### 37.7.52 **status\_t CLOCK\_SetBlpiMode ( void )**

This function sets the MCG to BLPI mode. If setting to BLPI mode fails from the current mode, this function returns an error.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

### 37.7.53 **status\_t CLOCK\_SetBlpeMode ( void )**

This function sets the MCG to BLPE mode. If setting to BLPE mode fails from the current mode, this function returns an error.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

### 37.7.54 **status\_t CLOCK\_SetPbeMode ( mcg\_pll\_clk\_select\_t *pllcs*, mcg\_pll\_config\_t const \* *config* )**

This function sets the MCG to PBE mode. If setting to PBE mode fails from the current mode, this function returns an error.

Parameters

<i>pllcs</i>	The PLL selection, PLLCS.
<i>config</i>	Pointer to the PLL configuration.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

Note

1. The parameter `pllcs` selects the PLL. For platforms with only one PLL, the parameter `pllc`s is kept for interface compatibility.
2. The parameter `config` is the PLL configuration structure. On some platforms, it is possible to choose the external PLL directly, which renders the configuration structure not necessary. In this case, pass in NULL. For example: `CLOCK_SetPbeMode(kMCG_OscselOsc, kMCG_Pl-ClkSelExtPll, NULL);`

### 37.7.55 status\_t CLOCK\_SetPeeMode( void )

This function sets the MCG to PEE mode.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

Note

This function only changes the CLKS to use the PLL/FLL output. If the PRDIV/VDIV are different than in the PBE mode, set them up in PBE mode and wait. When the clock is stable, switch to PEE mode.

### 37.7.56 status\_t CLOCK\_ExternalModeToFbeModeQuick( void )

This function switches the MCG from external modes (PEE/PBE/BLPE/FEE) to the FBE mode quickly. The external clock is used as the system clock source and PLL is disabled. However, the FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEE mode to FEI mode:

```
* CLOCK_ExternalModeToFbeModeQuick();
* CLOCK_SetFeiMode(...);
*
```

## Function Documentation

Return values

<i>kStatus_Success</i>	Switched successfully.
<i>kStatus_MCG_Mode-Invalid</i>	If the current mode is not an external mode, do not call this function.

### 37.7.57 status\_t CLOCK\_InternalModeToFbiModeQuick ( void )

This function switches the MCG from internal modes (PEI/PBI/BLPI/FEI) to the FBI mode quickly. The MCGIRCLK is used as the system clock source and PLL is disabled. However, FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEI mode to FEE mode:

```
* CLOCK_InternalModeToFbiModeQuick();
* CLOCK_SetFeeMode(...);
*
```

Return values

<i>kStatus_Success</i>	Switched successfully.
<i>kStatus_MCG_Mode-Invalid</i>	If the current mode is not an internal mode, do not call this function.

### 37.7.58 status\_t CLOCK\_BootToFeiMode ( mcg\_dmx32\_t *dmx32*, mcg\_drs\_t *drs*, void(\*)(void) *fllStableDelay* )

This function sets the MCG to FEI mode from the reset mode. It can also be used to set up MCG during system boot up.

Parameters

<i>dmx32</i>	DMX32 in FEI mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to ensure that the FLL is stable.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

## Note

If dmx32 is set to kMCG\_Dmx32Fine, the slow IRC must not be trimmed to frequency above 32768 Hz.

### 37.7.59 **status\_t CLOCK\_BootToFeeMode ( mcg\_oscsel\_t oscsel, uint8\_t frdiv, mcg\_dmx32\_t dmx32, mcg\_drs\_t drs, void(\*)(void) fllStableDelay )**

This function sets MCG to FEE mode from the reset mode. It can also be used to set up the MCG during system boot up.

## Parameters

<i>oscsel</i>	OSC clock select, OSCSEL.
<i>frdiv</i>	FLL reference clock divider setting, FRDIV.
<i>dmx32</i>	DMX32 in FEE mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to ensure that the FLL is stable.

## Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

### 37.7.60 **status\_t CLOCK\_BootToBlpiMode ( uint8\_t fcrdiv, mcg\_irc\_mode\_t ircs, uint8\_t ircEnableMode )**

This function sets the MCG to BLPI mode from the reset mode. It can also be used to set up the MCG during system boot up.

## Function Documentation

Parameters

<i>fcrdiv</i>	Fast IRC divider, FCRDIV.
<i>ircs</i>	The internal reference clock to select, IRCS.
<i>ircEnableMode</i>	The MCGIRCLK enable mode, OR'ed value of <a href="#"><u>_mcg_irclk_enable_mode</u></a> .

Return values

<i>kStatus_MCG_SourceUsed</i>	Could not change MCGIRCLK setting.
<i>kStatus_Success</i>	Switched to the target mode successfully.

### 37.7.61 **status\_t CLOCK\_BootToBlpeMode ( mcg\_oscsel\_t *oscsel* )**

This function sets the MCG to BLPE mode from the reset mode. It can also be used to set up the MCG during system boot up.

Parameters

<i>oscsel</i>	OSC clock select, MCG_C7[OSCSEL].
---------------	-----------------------------------

Return values

<i>kStatus_MCG_ModeUnreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

### 37.7.62 **status\_t CLOCK\_BootToPeeMode ( mcg\_oscsel\_t *oscsel*, mcg\_pll\_clk\_select\_t *pllcs*, mcg\_pll\_config\_t const \* *config* )**

This function sets the MCG to PEE mode from reset mode. It can also be used to set up the MCG during system boot up.

Parameters

<i>oscsel</i>	OSC clock select, MCG_C7[OSCSEL].
---------------	-----------------------------------

<i>pllcs</i>	The PLL selection, PLLCS.
<i>config</i>	Pointer to the PLL configuration.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

### 37.7.63 **status\_t CLOCK\_SetMcgConfig ( mcg\_config\_t const \* config )**

This function sets MCG to a target mode defined by the configuration structure. If switching to the target mode fails, this function chooses the correct path.

Parameters

<i>config</i>	Pointer to the target MCG mode configuration structure.
---------------	---

Returns

Return *kStatus\_Success* if switched successfully; Otherwise, it returns an error code [\\_mcg\\_status](#).

Note

If the external clock is used in the target mode, ensure that it is enabled. For example, if the OSC0 is used, set up OSC0 correctly before calling this function.

## 37.8 Variable Documentation

### 37.8.1 uint32\_t g\_xtal0Freq

The XTAL0/EXTAL0 (OSC0) clock frequency in Hz. When the clock is set up, use the function CLOCK\_SetXtal0Freq to set the value in the clock driver. For example, if XTAL0 is 8 MHz:

```
* CLOCK_InitOsc0(...); // Set up the OSC0
* CLOCK_SetXtal0Freq(80000000); // Set the XTAL0 value to the clock driver.
*
```

This is important for the multicore platforms where only one core needs to set up the OSC0 using the CLOCK\_InitOsc0. All other cores need to call the CLOCK\_SetXtal0Freq to get a valid clock frequency.

## Variable Documentation

### 37.8.2 `uint32_t g_xtal32Freq`

The XTAL32/EXTAL32/RTC\_CLKIN clock frequency in Hz. When the clock is set up, use the function CLOCK\_SetXtal32Freq to set the value in the clock driver.

This is important for the multicore platforms where only one core needs to set up the clock. All other cores need to call the CLOCK\_SetXtal32Freq to get a valid clock frequency.

## 37.9 Multipurpose Clock Generator (MCG)

The MCUXpresso SDK provides a peripheral driver for the module of MCUXpresso SDK devices.

### 37.9.1 Function description

MCG driver provides these functions:

- Functions to get the MCG clock frequency.
- Functions to configure the MCG clock, such as PLLCLK and MCGIRCLK.
- Functions for the MCG clock lock lost monitor.
- Functions for the OSC configuration.
- Functions for the MCG auto-trim machine.
- Functions for the MCG mode.

#### 37.9.1.1 MCG frequency functions

MCG module provides clocks, such as MCGOUTCLK, MCGIRCLK, MCGFFCLK, MCGFLLCLK and MCGPLLCLK. The MCG driver provides functions to get the frequency of these clocks, such as [CLOCK\\_GetOutClkFreq\(\)](#), [CLOCK\\_GetInternalRefClkFreq\(\)](#), [CLOCK\\_GetFixedFreqClkFreq\(\)](#), [CLOCK\\_GetFllFreq\(\)](#), [CLOCK\\_GetPll0Freq\(\)](#), [CLOCK\\_GetPll1Freq\(\)](#), and [CLOCK\\_GetExtPllFreq\(\)](#). These functions get the clock frequency based on the current MCG registers.

#### 37.9.1.2 MCG clock configuration

The MCG driver provides functions to configure the internal reference clock (MCGIRCLK), the external reference clock, and MCGPLLCLK.

The function [CLOCK\\_SetInternalRefClkConfig\(\)](#) configures the MCGIRCLK, including the source and the driver. Do not change MCGIRCLK when the MCG mode is BLPI/FBI/PBI because the MCGIRCLK is used as a system clock in these modes and changing settings makes the system clock unstable.

The function [CLOCK\\_SetExternalRefClkConfig\(\)](#) configures the external reference clock source (MCG\_C7[OSCSEL]). Do not call this function when the MCG mode is BLPE/FBE/PBE/FEE/PEE because the external reference clock is used as a clock source in these modes. Changing the external reference clock source requires at least a 50 microseconds wait. The function [CLOCK\\_SetExternalRefClkConfig\(\)](#) implements a for loop delay internally. The for loop delay assumes that the system clock is 96 MHz, which ensures at least 50 micro seconds delay. However, when the system clock is slow, the delay time may significantly increase. This for loop count can be optimized for better performance for specific cases.

The MCGPLLCLK is disabled in FBE/FEE/FBI/FEI modes by default. Applications can enable the MCGPLLCLK in these modes using the functions [CLOCK\\_EnablePll0\(\)](#) and [CLOCK\\_EnablePll1\(\)](#). To enable the MCGPLLCLK, the PLL reference clock divider(PRDIV) and the PLL VCO divider(VDIV) must be set to a proper value. The function [CLOCK\\_CalcPllDiv\(\)](#) helps to get the PRDIV/VDIV.

## Multipurpose Clock Generator (MCG)

### 37.9.1.3 MCG clock lock monitor functions

The MCG module monitors the OSC and the PLL clock lock status. The MCG driver provides the functions to set the clock monitor mode, check the clock lost status, and clear the clock lost status.

### 37.9.1.4 OSC configuration

The MCG is needed together with the OSC module to enable the OSC clock. The function [CLOCK\\_InitOsc0\(\)](#) `CLOCK_InitOsc1` uses the MCG and OSC to initialize the OSC. The OSC should be configured based on the board design.

### 37.9.1.5 MCG auto-trim machine

The MCG provides an auto-trim machine to trim the MCG internal reference clock based on the external reference clock (BUS clock). During clock trimming, the MCG must not work in FEI/FBI/BLPI/PBI/PEI modes. The function [CLOCK\\_TrimInternalRefClk\(\)](#) is used for the auto clock trimming.

### 37.9.1.6 MCG mode functions

The function `CLOCK_GetMcgMode` returns the current MCG mode. The MCG can only switch between the neighbouring modes. If the target mode is not current mode's neighbouring mode, the application must choose the proper switch path. For example, to switch to PEE mode from FEI mode, use FEI -> FBE -> PBE -> PEE.

For the MCG modes, the MCG driver provides three kinds of functions:

The first type of functions involve functions `CLOCK_SetXxxMode`, such as [CLOCK\\_SetFeiMode\(\)](#). These functions only set the MCG mode from neighbouring modes. If switching to the target mode directly from current mode is not possible, the functions return an error.

The second type of functions are the functions `CLOCK_BootToXxxMode`, such as [CLOCK\\_BootToFeiMode\(\)](#). These functions set the MCG to specific modes from reset mode. Because the source mode and target mode are specific, these functions choose the best switch path. The functions are also useful to set up the system clock during boot up.

The third type of functions is the [CLOCK\\_SetMcgConfig\(\)](#). This function chooses the right path to switch to the target mode. It is easy to use, but introduces a large code size.

Whenever the FLL settings change, there should be a 1 millisecond delay to ensure that the FLL is stable. The function [CLOCK\\_SetMcgConfig\(\)](#) implements a for loop delay internally to ensure that the FLL is stable. The for loop delay assumes that the system clock is 96 MHz, which ensures at least 1 millisecond delay. However, when the system clock is slow, the delay time may increase significantly. The for loop count can be optimized for better performance according to a specific use case.

### 37.9.2 Typical use case

The function `CLOCK_SetMcgConfig` is used to switch between any modes. However, this heavy-light function introduces a large code size. This section shows how to use the mode function to implement a quick and light-weight switch between typical specific modes. Note that the step to enable the external clock is not included in the following steps. Enable the corresponding clock before using it as a clock source.

#### 37.9.2.1 Switch between BLPI and FEI

Use case	Steps	Functions
BLPI -> FEI	BLPI -> FBI	<code>CLOCK_InternalModeToFbiModeQuick(...)</code>
	FBI -> FEI	<code>CLOCK_SetFeiMode(...)</code>
	Configure MCGIRCLK if need	<code>CLOCK_SetInternalRefClkConfig(...)</code>
FEI -> BLPI	Configure MCGIRCLK if need	<code>CLOCK_SetInternalRefClkConfig(...)</code>
	FEI -> FBI	<code>CLOCK_SetFbiMode(...)</code> with <code>fllStableDelay=NULL</code>
	FBI -> BLPI	<code>CLOCK_SetLowPowerEnable(true)</code>

#### 37.9.2.2 Switch between BLPI and FEE

Use case	Steps	Functions
BLPI -> FEE	BLPI -> FBI	<code>CLOCK_InternalModeToFbiModeQuick(...)</code>
	Change external clock source if need	<code>CLOCK_SetExternalRefClkConfig(...)</code>
	FBI -> FEE	<code>CLOCK_SetFeeMode(...)</code>
FEE -> BLPI	Configure MCGIRCLK if need	<code>CLOCK_SetInternalRefClkConfig(...)</code>
	FEE -> FBI	<code>CLOCK_SetFbiMode(...)</code> with <code>fllStableDelay=NULL</code>
	FBI -> BLPI	<code>CLOCK_SetLowPowerEnable(true)</code>

## Multipurpose Clock Generator (MCG)

### 37.9.2.3 Switch between BLPI and PEE

Use case	Steps	Functions
BLPI -> PEE	BLPI -> FBI	CLOCK_InternalModeToFbi-ModeQuick(...)
	Change external clock source if need	CLOCK_SetExternalRefClk-Config(...)
	FBI -> FBE	CLOCK_SetFbeMode(...) // f1l-StableDelay=NULL
	FBE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPI	PEE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config(...)
	FBE -> FBI	CLOCK_SetFbiMode(...) with f1lStableDelay=NULL
	FBI -> BLPI	CLOCK_SetLowPower-Enable(true)

### 37.9.2.4 Switch between BLPE and PEE

This table applies when using the same external clock source (MCG\_C7[OSCSEL]) in BLPE mode and PEE mode.

Use case	Steps	Functions
BLPE -> PEE	BLPE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPE	PEE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)
	FBE -> BLPE	CLOCK_SetLowPower-Enable(true)

If using different external clock sources (MCG\_C7[OSCSEL]) in BLPE mode and PEE mode, call the [CLOCK\\_SetExternalRefClkConfig\(\)](#) in FBI or FEI mode to change the external reference clock.

Use case	Steps	Functions
BLPE -> PEE	BLPE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)

	FBE -> FBI	CLOCK_SetFbiMode(...) with flStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	FBI -> FBE	CLOCK_SetFbeMode(...) with flStableDelay=NULL
	FBE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPE	PEE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)
	FBE -> FBI	CLOCK_SetFbiMode(...) with flStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with flStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPowerEnable(true)

### 37.9.2.5 Switch between BLPE and FEE

This table applies when using the same external clock source (MCG\_C7[OSCSEL]) in BLPE mode and FEE mode.

Use case	Steps	Functions
BLPE -> FEE	BLPE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)
	FBE -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	PEE -> FBE	CLOCK_SetPbeMode(...)
	FBE -> BLPE	CLOCK_SetLowPowerEnable(true)

If using different external clock sources (MCG\_C7[OSCSEL]) in BLPE mode and FEE mode, call the [CLOCK\\_SetExternalRefClkConfig\(\)](#) in FBI or FEI mode to change the external reference clock.

Use case	Steps	Functions
BLPE -> FEE	BLPE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)

## Multipurpose Clock Generator (MCG)

	FBE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	FBI -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	FEE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with fllStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPowerEnable(true)

### 37.9.2.6 Switch between BLPI and PEI

Use case	Steps	Functions
BLPI -> PEI	BLPI -> PBI	CLOCK_SetPbiMode(...)
	PBI -> PEI	CLOCK_SetPeiMode(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClkConfig(...)
PEI -> BLPI	Configure MCGIRCLK if need	CLOCK_SetInternalRefClkConfig
	PEI -> FBI	CLOCK_InternalModeToFbiModeQuick(...)
	FBI -> BLPI	CLOCK_SetLowPowerEnable(true)

### 37.9.3 Code Configuration Option

#### 37.9.3.1 MCG\_USER\_CONFIG\_FLL\_STABLE\_DELAY\_EN

When switching to use FLL with function `CLOCK_SetFeeMode()` and `CLOCK_SetFeeMode()`, there is an internal function `CLOCK_FllStableDelay()`. It is used to delay a few ms so that to wait the FLL to be stable enough. By default, it is implemented in driver code like:

```
#ifndef MCG_USER_CONFIG_FLL_STABLE_DELAY_EN
void CLOCK_FllStableDelay(void)
{
    /*
```

```
Should wait at least 1ms. Because in these modes, the core clock is 100MHz
at most, so this function could obtain the 1ms delay.

*/
volatile uint32_t i = 30000U;
while (i--)
{
    __NOP();
}
#endif /* MCG_USER_CONFIG_FLL_STABLE_DELAY_EN */
```

Once user is willing to create his own delay function, just assert the macro MCG\_USER\_CONFIG\_FLL\_STABLE\_DELAY\_EN, and then define function CLOCK\_FllStableDelay in the application code.



# Chapter 38

## DMA Manager

### 38.1 Overview

DMA Manager provides a series of functions to manage the DMAMUX instances and channels.

### 38.2 Function groups

#### 38.2.1 DMAMGR Initialization and De-initialization

This function group initializes and deinitializes the DMA Manager.

#### 38.2.2 DMAMGR Operation

This function group requests/releases the DMAMUX channel and configures the channel request source.

### 38.3 Typical use case

#### 38.3.1 DMAMGR static channel allocattion

```
uint8_t channel;
dmamanager_handle_t dmamanager_handle;

/* Initialize DMAMGR */
DMAMGR_Init(&dmamanager_handle, EXAMPLE_DMA_BASEADDR, DMA_CHANNEL_NUMBER, startChannel);
/* Request a DMAMUX channel by static allocate mechanism */
channel = kDMAMGR_STATIC_ALLOCATE;
DMAMGR_RequestChannel(&dmamanager_handle, kDmaRequestMux0AlwaysOn63, channel, &handle)
;
```

#### 38.3.2 DMAMGR dynamic channel allocation

```
uint8_t channel;
dmamanager_handle_t dmamanager_handle;

/* Initialize DMAMGR */
DMAMGR_Init(&dmamanager_handle, EXAMPLE_DMA_BASEADDR, DMA_CHANNEL_NUMBER, startChannel);
/* Request a DMAMUX channel by Dynamic allocate mechanism */
channel = DMAMGR_DYNAMIC_ALLOCATE;
DMAMGR_RequestChannel(&dmamanager_handle, kDmaRequestMux0AlwaysOn63, channel, &handle)
;
```

## Data Structures

- struct `dmamanager_handle_t`  
*dmamanager handle typedef. [More...](#)*

## Data Structure Documentation

### Macros

- #define **DMAMGR\_DYNAMIC\_ALLOCATE** 0xFFU  
*Dynamic channel allocation mechanism.*

### Enumerations

- enum **\_dma\_manager\_status** {  
    kStatus\_DMAMGR\_ChannelOccupied = MAKE\_STATUS(kStatusGroup\_DMAMGR, 0),  
    kStatus\_DMAMGR\_ChannelNotUsed = MAKE\_STATUS(kStatusGroup\_DMAMGR, 1),  
    kStatus\_DMAMGR\_NoFreeChannel = MAKE\_STATUS(kStatusGroup\_DMAMGR, 2) }  
*DMA manager status.*

## DMAMGR Initialization and De-initialization

- void **DMAMGR\_Init** (**dmamanager\_handle\_t** \*dmamanager\_handle, **DMA\_Type** \*dma\_base, **uint32\_t** channelNum, **uint32\_t** startChannel)  
*Initializes the DMA manager.*
- void **DMAMGR\_Deinit** (**dmamanager\_handle\_t** \*dmamanager\_handle)  
*Deinitializes the DMA manager.*

## DMAMGR Operation

- **status\_t DMAMGR\_RequestChannel** (**dmamanager\_handle\_t** \*dmamanager\_handle, **uint32\_t** requestSource, **uint32\_t** channel, **void** \*handle)  
*Requests a DMA channel.*
- **status\_t DMAMGR\_ReleaseChannel** (**dmamanager\_handle\_t** \*dmamanager\_handle, **void** \*handle)  
*Releases a DMA channel.*
- **bool DMAMGR\_IsChannelOccupied** (**dmamanager\_handle\_t** \*dmamanager\_handle, **uint32\_t** channel)  
*Get a DMA channel status.*

## 38.4 Data Structure Documentation

### 38.4.1 struct dmamanager\_handle\_t

Note

The contents of this structure are private and subject to change.

This dma manager handle structure is used to store the parameters transferred by users. And users shall not free the memory before calling DMAMGR\_Deinit, also shall not modify the contents of the memory.

### Data Fields

- **void \*dma\_base**  
*Peripheral DMA instance.*
- **uint32\_t channelNum**

- *Channel numbers for the DMA instance which need to be managed by dma manager.*

- `uint32_t startChannel`  
*The start channel that can be managed by dma manager; users need to transfer it with a certain number or NULL.*

- `bool s_DMAMGR_Channels [64]`  
*The s\_DMAMGR\_Channels is used to store dma manager state.*

- `uint32_t DmamuxInstanceStart`  
*The DmamuxInstance is used to calculate the DMAMUX Instance according to the DMA Instance.*

- `uint32_t multiple`  
*The multiple is used to calculate the multiple between DMAMUX count and DMA count.*

#### 38.4.1.0.0.84 Field Documentation

38.4.1.0.0.84.1 `void* dmamanager_handle_t::dma_base`

38.4.1.0.0.84.2 `uint32_t dmamanager_handle_t::channelNum`

38.4.1.0.0.84.3 `uint32_t dmamanager_handle_t::startChannel`

38.4.1.0.0.84.4 `bool dmamanager_handle_t::s_DMAMGR_Channels[64]`

38.4.1.0.0.84.5 `uint32_t dmamanager_handle_t::DmamuxInstanceStart`

38.4.1.0.0.84.6 `uint32_t dmamanager_handle_t::multiple`

#### 38.5 Macro Definition Documentation

38.5.1 `#define DMAMGR_DYNAMIC_ALLOCATE 0xFFU`

#### 38.6 Enumeration Type Documentation

38.6.1 `enum _dma_manager_status`

Enumerator

`kStatus_DMAMGR_ChannelOccupied` Channel has been occupied.

`kStatus_DMAMGR_ChannelNotUsed` Channel has not been used.

`kStatus_DMAMGR_NoFreeChannel` All channels have been occupied.

#### 38.7 Function Documentation

38.7.1 `void DMAMGR_Init ( dmamanager_handle_t * dmamanager_handle, DMA_Type * dma_base, uint32_t channelNum, uint32_t startChannel )`

This function initializes the DMA manager, ungates the DMAMUX clocks, and initializes the eDMA or DMA peripherals.

## Function Documentation

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>dma_base</i>	Peripheral DMA instance base pointer.
<i>dmamux_base</i>	Peripheral DMAMUX instance base pointer.
<i>channelNum</i>	Channel numbers for the DMA instance which need to be managed by dma manager.
<i>startChannel</i>	The start channel that can be managed by dma manager.

### 38.7.2 void DMAMGR\_Deinit ( *dmamanager\_handle\_t \* dmamanager\_handle* )

This function deinitializes the DMA manager, disables the DMAMUX channels, gates the DMAMUX clocks, and deinitializes the eDMA or DMA peripherals.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
--------------------------	---

### 38.7.3 status\_t DMAMGR\_RequestChannel ( *dmamanager\_handle\_t \* dmamanager\_handle, uint32\_t requestSource, uint32\_t channel, void \* handle* )

This function requests a DMA channel which is not occupied. The two channels to allocate the mechanism are dynamic and static channels. For the dynamic allocation mechanism (channe = DMAMGR\_DYNAMIC\_ALLOCATE), DMAMGR allocates a DMA channel according to the given request source and start-Channel and then configures it. For static allocation mechanism, DMAMGR configures the given channel according to the given request source and channel number.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>requestSource</i>	DMA channel request source number. See the soc.h, see the enum <code>dma_request_source_t</code>
<i>channel</i>	The channel number users want to occupy. If using the dynamic channel allocate mechanism, set the channel equal to <code>DMAMGR_DYNAMIC_ALLOCATE</code> .
<i>handle</i>	DMA or eDMA handle pointer.

Return values

<i>kStatus_Success</i>	In a dynamic/static channel allocation mechanism, allocate the DMAMUX channel successfully.
<i>kStatus_DMAMGR_No_FreeChannel</i>	In a dynamic channel allocation mechanism, all DMAMUX channels are occupied.
<i>kStatus_DMAMGR_ChannelOccupied</i>	In a static channel allocation mechanism, the given channel is occupied.

### 38.7.4 **status\_t DMAMGR\_ReleaseChannel ( dmamanager\_handle\_t \* dmamanager\_handle, void \* handle )**

This function releases an occupied DMA channel.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>handle</i>	DMA or eDMA handle pointer.

Return values

## Function Documentation

<i>kStatus_Success</i>	Releases the given channel successfully.
<i>kStatus_DMAMGR_-ChannelNotUsed</i>	The given channel to be released had not been used before.

### 38.7.5 **bool DMAMGR\_IsChannelOccupied ( dmamanager\_handle\_t \* dmamanager\_handle, uint32\_t channel )**

This function get a DMA channel status. Return 0 indicates the channel has not been used, return 1 indicates the channel has been occupied.

#### Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal,users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>channel</i>	The channel number that users want get its status.

# **Chapter 39**

## **Memory-Mapped Cryptographic Acceleration Unit (MMCAU)**

### **39.1 Overview**

The mmCAU software library uses the mmCAU co-processor that is connected to the ARM Cortex-M4-/M0+ Private Peripheral Bus (PPB). In this chapter, CAU refers to both CAU and mmCAU unless explicitly noted.

### **39.2 Purpose**

The following chapter describes how to use the mmCAU software library in any application to integrate a cryptographic algorithm or hashing function supported by the software library. NXP products supported by the software library are MCU/MPUs. Check the specific Freescale product for CAU availability.

### **39.3 Library Features**

The library is as compact and generic as possible to simplify the integration with existing cryptographic software. The library has a standard header file with ANSI C prototypes for all functions: "cau\_api.h". This software library is thread safe only if CAU registers are saved on a context switch. The mmCAU software library is also compatible to ARM C compiler conventions (EABI). All pointers passed to mmCAU API functions (input and output data blocks, keys, key schedules, and so on) are aligned to 0-modulo-4 addresses.

For applications that don't need to deal with the aligned addresses, a simple wrapper layer is provided. The wrapper layer consists of the "fsl\_mmcau.h" header file and "fsl\_mmcau.c" source code file. The only function of the wrapper layer is that it supports unaligned addresses

. The CAU library supports the following encryption/decryption algorithms and hashing functions:

- AES128
- AES192
- AES256
- DES
- MD5
- SHA1
- SHA256

**Note:** 3DES crypto algorithms are supported by calling the corresponding DES crypto function three times. Hardware support for SHA256 is only present in the CAU version 2. See the appropriate MC-U/MPU reference manual for details about availability. Additionally, the [cau\\_sha256\\_initialize\\_output\(\)](#) function checks the hardware revision and returns a (-1) value if the CAU lacks SHA256 support.

### **39.4 CAU and mmCAU software library overview**

Table 1 shows the crypto algorithms and hashing functions included in the software library:

## mmCAU software library usage

Crypto Algorithms	AES128 AES192 AES256	cau_aes_set_key
		cau_aes_encrypt
		cau_aes_decrypt
	DES/3DES	cau_des_chk_parity
		cau_des_encrypt
		cau_des_decrypt
Hashing Functions	MD5	cau_md5_initialize_output
		cau_md5_hash_n
		cau_md5_update
		cau_md5_hash
	SHA1	cau_sha1_initialize_output
		cau_sha1_hash_n
		cau_sha1_update
		cau_sha1_hash
	SHA256	cau_sha256_initialize_output
		cau_sha256_hash_n
		cau_sha256_update
		cau_sha256_hash

**Table 1: Library Overview**

### 39.5 mmCAU software library usage

The software library contains the following files:

File	Description
cau_api.h	CAU and mmCAU header file
lib_mmcau.a	mmCAU library

**Table 2: File Description**

The header file and lib\_mmcau.a must always be included in the project.

### Functions

- void `cau_aes_set_key` (const unsigned char \*key, const int key\_size, unsigned char \*key\_sch)  
*AES: Performs an AES key expansion.*
- void `cau_aes_encrypt` (const unsigned char \*in, const unsigned char \*key\_sch, const int nr, unsigned char \*out)

- void **cau\_aes\_decrypt** (const unsigned char \*in, const unsigned char \*key\_sch, const int nr, unsigned char \*out)
 

*AES: Encrypts a single 16 byte block.*
- int **cau\_des\_chk\_parity** (const unsigned char \*key)
 

*DES: Checks key parity.*
- void **cau\_des\_encrypt** (const unsigned char \*in, const unsigned char \*key, unsigned char \*out)
 

*DES: Encrypts a single 8-byte block.*
- void **cau\_des\_decrypt** (const unsigned char \*in, const unsigned char \*key, unsigned char \*out)
 

*DES: Decrypts a single 8-byte block.*
- void **cau\_md5\_initialize\_output** (const unsigned char \*md5\_state)
 

*MD5: Initializes the MD5 state variables.*
- void **cau\_md5\_hash\_n** (const unsigned char \*msg\_data, const int num\_blks, unsigned char \*md5\_state)
 

*MD5: Updates MD5 state variables with n message blocks.*
- void **cau\_md5\_update** (const unsigned char \*msg\_data, const int num\_blks, unsigned char \*md5\_state)
 

*MD5: Updates MD5 state variables.*
- void **cau\_md5\_hash** (const unsigned char \*msg\_data, unsigned char \*md5\_state)
 

*MD5: Updates MD5 state variables with one message block.*
- void **cau\_sha1\_initialize\_output** (const unsigned int \*sha1\_state)
 

*SHA1: Initializes the SHA1 state variables.*
- void **cau\_sha1\_hash\_n** (const unsigned char \*msg\_data, const int num\_blks, unsigned int \*sha1\_state)
 

*SHA1: Updates SHA1 state variables with n message blocks.*
- void **cau\_sha1\_update** (const unsigned char \*msg\_data, const int num\_blks, unsigned int \*sha1\_state)
 

*SHA1: Updates SHA1 state variables.*
- void **cau\_sha1\_hash** (const unsigned char \*msg\_data, unsigned int \*sha1\_state)
 

*SHA1: Updates SHA1 state variables with one message block.*
- int **cau\_sha256\_initialize\_output** (const unsigned int \*output)
 

*SHA256: Initializes the SHA256 state variables.*
- void **cau\_sha256\_hash\_n** (const unsigned char \*input, const int num\_blks, unsigned int \*output)
 

*SHA256: Updates SHA256 state variables with n message blocks.*
- void **cau\_sha256\_update** (const unsigned char \*input, const int num\_blks, unsigned int \*output)
 

*SHA256: Updates SHA256 state variables.*
- void **cau\_sha256\_hash** (const unsigned char \*input, unsigned int \*output)
 

*SHA256: Updates SHA256 state variables with one message block.*
- status\_t **MMCAU\_AES\_SetKey** (const uint8\_t \*key, const size\_t keySize, uint8\_t \*keySch)
 

*AES: Performs an AES key expansion.*
- status\_t **MMCAU\_AES\_EncryptEcb** (const uint8\_t \*in, const uint8\_t \*keySch, uint32\_t aesRounds, uint8\_t \*out)
 

*AES: Encrypts a single 16 byte block.*
- status\_t **MMCAU\_AES\_DecryptEcb** (const uint8\_t \*in, const uint8\_t \*keySch, uint32\_t aesRounds, uint8\_t \*out)
 

*AES: Decrypts a single 16-byte block.*
- status\_t **MMCAU\_DES\_ChkParity** (const uint8\_t \*key)
 

*DES: Checks the key parity.*
- status\_t **MMCAU\_DES\_EncryptEcb** (const uint8\_t \*in, const uint8\_t \*key, uint8\_t \*out)
 

*DES: Encrypts a single 8-byte block.*

## Function Documentation

- status\_t [MMCAU\\_DES\\_DecryptEcb](#) (const uint8\_t \*in, const uint8\_t \*key, uint8\_t \*out)  
*DES: Decrypts a single 8-byte block.*
- status\_t [MMCAU\\_MD5\\_InitializeOutput](#) (uint32\_t \*md5State)  
*MD5: Initializes the MD5 state variables.*
- status\_t [MMCAU\\_MD5\\_HashN](#) (const uint8\_t \*msgData, uint32\_t numBlocks, uint32\_t \*md5State)  
*MD5: Updates the MD5 state variables with n message blocks.*
- status\_t [MMCAU\\_MD5\\_Update](#) (const uint8\_t \*msgData, uint32\_t numBlocks, uint32\_t \*md5State)  
*MD5: Updates the MD5 state variables.*
- status\_t [MMCAU\\_SHA1\\_InitializeOutput](#) (uint32\_t \*sha1State)  
*SHA1: Initializes the SHA1 state variables.*
- status\_t [MMCAU\\_SHA1\\_HashN](#) (const uint8\_t \*msgData, uint32\_t numBlocks, uint32\_t \*sha1State)  
*SHA1: Updates the SHA1 state variables with n message blocks.*
- status\_t [MMCAU\\_SHA1\\_Update](#) (const uint8\_t \*msgData, uint32\_t numBlocks, uint32\_t \*sha1State)  
*SHA1: Updates the SHA1 state variables.*
- status\_t [MMCAU\\_SHA256\\_InitializeOutput](#) (uint32\_t \*sha256State)  
*SHA256: Initializes the SHA256 state variables.*
- status\_t [MMCAU\\_SHA256\\_HashN](#) (const uint8\_t \*input, uint32\_t numBlocks, uint32\_t \*sha256State)  
*SHA256: Updates the SHA256 state variables with n message blocks.*
- status\_t [MMCAU\\_SHA256\\_Update](#) (const uint8\_t \*input, uint32\_t numBlocks, uint32\_t \*sha256State)  
*SHA256: Updates SHA256 state variables.*

## 39.6 Function Documentation

### 39.6.1 void cau\_aes\_set\_key ( const unsigned char \* key, const int key\_size, unsigned char \* key\_sch )

This function performs an AES key expansion

Parameters

	<i>key</i>	Pointer to input key (128, 192, 256 bits in length).
	<i>key_size</i>	Key size in bits (128, 192, 256)
out	<i>key_sch</i>	Pointer to key schedule output (44, 52, 60 longwords)

Note

All pointers must have word (4 bytes) alignment

Table below shows the requirements for the [cau\\_aes\\_set\\_key\(\)](#) function when using AES128, AES192 or AES256.

[in] Key Size (bits)   [out] Key Schedule Size (32 bit data values)
:-----:   :-----:

128	44
192	52
256	60

### 39.6.2 void cau\_aes\_encrypt( const unsigned char \* *in*, const unsigned char \* *key\_sch*, const int *nr*, unsigned char \* *out* )

This function encrypts a single 16-byte block for AES128, AES192 and AES256

Parameters

	<i>in</i>	Pointer to 16-byte block of input plaintext
	<i>key_sch</i>	Pointer to key schedule (44, 52, 60 longwords)
	<i>nr</i>	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	<i>out</i>	Pointer to 16-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

Table below shows the requirements for the [cau\\_aes\\_encrypt\(\)](#)/[cau\\_aes\\_decrypt\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
AES128	44	10
AES192	52	12
AES256	60	14

### 39.6.3 void cau\_aes\_decrypt( const unsigned char \* *in*, const unsigned char \* *key\_sch*, const int *nr*, unsigned char \* *out* )

This function decrypts a single 16-byte block for AES128, AES192 and AES256

Parameters

## Function Documentation

	<i>in</i>	Pointer to 16-byte block of input ciphertext
	<i>key_sch</i>	Pointer to key schedule (44, 52, 60 longwords)
	<i>nr</i>	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	<i>out</i>	Pointer to 16-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

Table below shows the requirements for the [cau\\_aes\\_encrypt\(\)](#)/[cau\\_aes\\_decrypt\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
AES128	44	10
AES192	52	12
AES256	60	14

### 39.6.4 int cau\_des\_chk\_parity ( const unsigned char \* *key* )

This function checks the parity of a DES key

Parameters

<i>key</i>	64-bit DES key with parity bits. Must have word (4 bytes) alignment.
------------	--

Returns

0 no error

-1 parity error

### 39.6.5 void cau\_des\_encrypt ( const unsigned char \* *in*, const unsigned char \* *key*, unsigned char \* *out* )

This function encrypts a single 8-byte block with DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input plaintext
	<i>key</i>	Pointer to 64-bit DES key with parity bits
out	<i>out</i>	Pointer to 8-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

### **39.6.6 void cau\_des\_decrypt ( const unsigned char \* *in*, const unsigned char \* *key*, unsigned char \* *out* )**

This function decrypts a single 8-byte block with DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input ciphertext
	<i>key</i>	Pointer to 64-bit DES key with parity bits
out	<i>out</i>	Pointer to 8-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

### **39.6.7 void cau\_md5\_initialize\_output ( const unsigned char \* *md5\_state* )**

This function initializes the MD5 state variables. The output can be used as input to [cau\\_md5\\_hash\(\)](#) and [cau\\_md5\\_hash\\_n\(\)](#).

Parameters

out	<i>md5_state</i>	Pointer to 128-bit block of md5 state variables: a,b,c,d
-----	------------------	--

Note

All pointers must have word (4 bytes) alignment

## Function Documentation

**39.6.8 void cau\_md5\_hash\_n ( const unsigned char \* *msg\_data*, const int *num\_blk*s, unsigned char \* *md5\_state* )**

This function updates MD5 state variables for one or more input message blocks

## Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
in, out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

## Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_md5\\_initialize\\_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

### 39.6.9 void cau\_md5\_update ( const unsigned char \* *msg\_data*, const int *num\_blk*s, unsigned char \* *md5\_state* )

This function updates MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau\\_md5\\_initialize\\_output\(\)](#) first.

## Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

## Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_md5\\_initialize\\_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

### 39.6.10 void cau\_md5\_hash ( const unsigned char \* *msg\_data*, unsigned char \* *md5\_state* )

This function updates MD5 state variables for one input message block

## Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
in,out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_md5\\_initialize\\_output\(\)](#) function must be called when starting a new hash.

### 39.6.11 void cau\_sha1\_initialize\_output ( const unsigned int \* *sha1\_state* )

This function initializes the SHA1 state variables. The output can be used as input to [cau\\_sha1\\_hash\(\)](#) and [cau\\_sha1\\_hash\\_n\(\)](#).

Parameters

out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e
-----	-------------------	---

Note

All pointers must have word (4 bytes) alignment

### 39.6.12 void cau\_sha1\_hash\_n ( const unsigned char \* *msg\_data*, const int *num\_blk*s, unsigned int \* *sha1\_state* )

This function updates SHA1 state variables for one or more input message blocks

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
in,out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_sha1\\_initialize\\_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

**39.6.13 void cau\_sha1\_update ( const unsigned char \* *msg\_data*, const int *num\_blk*s, unsigned int \* *sha1\_state* )**

This function updates SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau\\_sha1\\_initialize\\_output\(\)](#) first.

## Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_sha1\\_initialize\\_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

### **39.6.14 void cau\_sha1\_hash ( const unsigned char \* *msg\_data*, unsigned int \* *sha1\_state* )**

This function updates SHA1 state variables for one input message block

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
in, out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_sha1\\_initialize\\_output\(\)](#) function must be called when starting a new hash.

### **39.6.15 int cau\_sha256\_initialize\_output ( const unsigned int \* *output* )**

This function initializes the SHA256 state variables. The output can be used as input to [cau\\_sha256\\_hash\(\)](#) and [cau\\_sha256\\_hash\\_n\(\)](#).

Parameters

out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables a,b,c,d,e,f,g,h
-----	---------------------	--

## Note

All pointers must have word (4 bytes) alignment

## Returns

- 0 No error. CAU hardware support for SHA256 is present.
- 1 Error. CAU hardware support for SHA256 is not present.

### **39.6.16 void cau\_sha256\_hash\_n ( const unsigned char \* *input*, const int *num\_blk*s, unsigned int \* *output* )**

This function updates SHA256 state variables for one or more input message blocks

## Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
in,out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

## Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_sha256\\_initialize\\_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

### **39.6.17 void cau\_sha256\_update ( const unsigned char \* *input*, const int *num\_blk*s, unsigned int \* *output* )**

This function updates SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau\\_sha256\\_initialize\\_output\(\)](#) first.

## Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blk</i> s	Number of 512-bit blocks to process
out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_sha256\\_initialize\\_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

### **39.6.18 void cau\_sha256\_hash ( const unsigned char \* *input*, unsigned int \* *output* )**

This function updates SHA256 state variables for one input message block

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
in, out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau\\_sha256\\_initialize\\_output\(\)](#) function must be called when starting a new hash.

### **39.6.19 status\_t MMCAU\_AES\_SetKey ( const uint8\_t \* *key*, const size\_t *keySize*, uint8\_t \* *keySch* )**

This function performs an AES key expansion.

Parameters

	<i>key</i>	Pointer to input key (128, 192, 256 bits in length).
	<i>keySize</i>	Key size in bytes (16, 24, 32)
out	<i>keySch</i>	Pointer to key schedule output (44, 52, 60 longwords)

## Note

Table below shows the requirements for the [MMCAU\\_AES\\_SetKey\(\)](#) function when using AES128, AES192, or AES256.

[in] Key Size (bits)	[out] Key Schedule Size (32 bit data values)
-----	-----
128	44
192	52
256	60

## Returns

Status of the operation. (kStatus\_Success, kStatus\_InvalidArgument, kStatus\_Fail)

### 39.6.20 status\_t MMCAU\_AES\_EncryptEcb ( const uint8\_t \* *in*, const uint8\_t \* *keySch*, uint32\_t *aesRounds*, uint8\_t \* *out* )

This function encrypts a single 16-byte block for AES128, AES192, and AES256.

## Parameters

	<i>in</i>	Pointer to 16-byte block of input plaintext.
	<i>keySch</i>	Pointer to key schedule (44, 52, 60 longwords).
	<i>aesRounds</i>	Number of AES rounds (10, 12, 14 = f(key_schedule)).
out	<i>out</i>	Pointer to 16-byte block of output ciphertext.

## Note

Input and output blocks may overlap.

Table below shows the requirements for the [MMCAU\\_AES\\_EncryptEcb\(\)](#)/[MMCAU\\_AES\\_DecryptEcb\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
-----	-----	-----
AES128	44	10
AES192	52	12
AES256	60	14

## Function Documentation

Returns

Status of the operation. (kStatus\_Success, kStatus\_InvalidArgument, kStatus\_Fail)

### 39.6.21 **status\_t MMCAU\_AES\_DecryptEcb ( const uint8\_t \* *in*, const uint8\_t \* *keySch*, uint32\_t *aesRounds*, uint8\_t \* *out* )**

This function decrypts a single 16-byte block for AES128, AES192, and AES256.

Parameters

	<i>in</i>	Pointer to 16-byte block of input ciphertext.
	<i>keySch</i>	Pointer to key schedule (44, 52, 60 longwords).
	<i>aesRounds</i>	Number of AES rounds (10, 12, 14 = f(key_schedule)).
out	<i>out</i>	Pointer to 16-byte block of output plaintext.

Note

Input and output blocks may overlap.

Table below shows the requirements for the [cau\\_aes\\_encrypt\(\)](#)/[cau\\_aes\\_decrypt\(\)](#). function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
:-----:   :-----:   :-----:		
AES128   44   10		
AES192   52   12		
AES256   60   14		

Returns

Status of the operation. (kStatus\_Success, kStatus\_InvalidArgument, kStatus\_Fail)

### 39.6.22 **status\_t MMCAU\_DES\_ChkParity ( const uint8\_t \* *key* )**

This function checks the parity of a DES key.

Parameters

<i>key</i>	64-bit DES key with parity bits.
------------	----------------------------------

Returns

kStatus\_Success No error.  
kStatus\_Fail Parity error.  
kStatus\_InvalidArgument Key argument is NULL.

### 39.6.23 status\_t MMCAU\_DES\_EncryptEcb ( const uint8\_t \* *in*, const uint8\_t \* *key*, uint8\_t \* *out* )

This function encrypts a single 8-byte block with the DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input plaintext.
	<i>key</i>	Pointer to 64-bit DES key with parity bits.
<i>out</i>	<i>out</i>	Pointer to 8-byte block of output ciphertext.

Note

Input and output blocks may overlap.

Returns

Status of the operation. (kStatus\_Success, kStatus\_InvalidArgument, kStatus\_Fail)

### 39.6.24 status\_t MMCAU\_DES\_DecryptEcb ( const uint8\_t \* *in*, const uint8\_t \* *key*, uint8\_t \* *out* )

This function decrypts a single 8-byte block with the DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input ciphertext.
--	-----------	--

## Function Documentation

	<i>key</i>	Pointer to 64-bit DES key with parity bits.
out	<i>out</i>	Pointer to 8-byte block of output plaintext.

### Note

Input and output blocks may overlap.

### Returns

Status of the operation. (kStatus\_Success, kStatus\_InvalidArgument, kStatus\_Fail)

### 39.6.25 status\_t MMCAU\_MD5\_InitializeOutput ( *uint32\_t \* md5State* )

This function initializes the MD5 state variables. The output can be used as input to [MMCAU\\_MD5\\_HashN\(\)](#).

#### Parameters

out	<i>md5State</i>	Pointer to 128-bit block of md5 state variables: a,b,c,d
-----	-----------------	--

### 39.6.26 status\_t MMCAU\_MD5\_HashN ( *const uint8\_t \* msgData, uint32\_t numBlocks, uint32\_t \* md5State* )

This function updates the MD5 state variables for one or more input message blocks.

#### Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
in,out	<i>md5State</i>	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

### Note

Input message and digest output blocks must not overlap. The [MMCAU\\_MD5\\_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

**39.6.27 status\_t MMCAU\_MD5\_Update ( const uint8\_t \* *msgData*, uint32\_t *numBlocks*, uint32\_t \* *md5State* )**

This function updates the MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls [MMCAU\\_MD5\\_InitializeOutput\(\)](#) first.

## Function Documentation

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
out	<i>md5State</i>	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

Note

Input message and digest output blocks must not overlap. The [MMCAU\\_MD5\\_InitializeOutput\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

### 39.6.28 status\_t MMCAU\_SHA1\_InitializeOutput ( *uint32\_t \* sha1State* )

This function initializes the SHA1 state variables. The output can be used as input to [MMCAU\\_SHA1\\_HashN\(\)](#).

Parameters

out	<i>sha1State</i>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.
-----	------------------	--

### 39.6.29 status\_t MMCAU\_SHA1\_HashN ( *const uint8\_t \* msgData, uint32\_t numBlocks, uint32\_t \* sha1State* )

This function updates the SHA1 state variables for one or more input message blocks.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
in, out	<i>sha1State</i>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The [MMCAU\\_SHA1\\_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

**39.6.30 status\_t MMCAU\_SHA1\_Update ( const uint8\_t \* *msgData*, uint32\_t *numBlocks*, uint32\_t \* *sha1State* )**

This function updates the SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls [MMCAU\\_SHA1\\_InitializeOutput\(\)](#) first.

## Function Documentation

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
out	<i>sha1State</i>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The [MMCAU\\_SHA1\\_InitializeOutput\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

### 39.6.31 status\_t MMCAU\_SHA256\_InitializeOutput ( *uint32\_t \* sha256State* )

This function initializes the SHA256 state variables. The output can be used as input to [MMCAU\\_SHA256\\_HashN\(\)](#).

Parameters

out	<i>sha256State</i>	Pointer to 256-bit block of SHA2 state variables a, b, c, d, e, f, g, h.
-----	--------------------	--

Returns

kStatus\_Success No error. CAU hardware support for SHA256 is present.

kStatus\_Fail Error. CAU hardware support for SHA256 is not present.

kStatus\_InvalidArgument Error. sha256State is NULL.

### 39.6.32 status\_t MMCAU\_SHA256\_HashN ( *const uint8\_t \* input*, *uint32\_t numBlocks*, *uint32\_t \* sha256State* )

This function updates SHA256 state variables for one or more input message blocks.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
--	----------------	---

	<i>numBlocks</i>	Number of 512-bit blocks to process.
in, out	<i>sha256State</i>	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

## Note

Input message and digest output blocks must not overlap. The [MMCAU\\_SHA256\\_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

### 39.6.33 status\_t MMCAU\_SHA256\_Update ( const uint8\_t \* *input*, uint32\_t *numBlocks*, uint32\_t \* *sha256State* )

This function updates the SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau\\_sha256\\_initialize\\_output\(\)](#) first.

## Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
out	<i>sha256State</i>	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

## Note

Input message and digest output blocks must not overlap. The [MMCAU\\_SHA256\\_InitializeOutput\(\)](#) function is not required to be called. as it is called internally to start a new hash. All input message blocks must be contiguous.

## Function Documentation

# Chapter 40

## Secure Digital Card/Embedded MultiMedia Card (CARD)

### 40.1 Overview

The MCUXpresso SDK provides a driver to access the Secure Digital Card and Embedded MultiMedia Card based on the SDHC driver.

### Function groups

This function group implements the SD card functional API.

This function group implements the MMC card functional API.

### Typical use case

```
/* Initialize SDHC. */
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Save host information. */
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->host.transfer = SDHC_TransferFunction;

/* Init card. */
if (SD_Init(card))
{
    PRINTF("\r\nSD card init failed.\r\n");
}

while (true)
{
    if (kStatus_Success != SD_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != SD_ReadBlocks(card, g_dataRead, DATA_BLOCK_START, DATA_BLOCK_COUNT))
    {
        PRINTF("Read multiple data blocks failed.\r\n");
    }

    if (kStatus_Success != SD_EraseBlocks(card, DATA_BLOCK_START, DATA_BLOCK_COUNT))
    {
        PRINTF("Erase multiple data blocks failed.\r\n");
    }
}

SD_Deinit(card);

/* Initialize SDHC. */
```

## Overview

```
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Save host information. */
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->host.transfer = SDHC_TransferFunction;

/* Init card. */
if (MMC_Init(card))
{
    PRINTF("\n MMC card init failed \n");
}

while (true)
{
    if (kStatus_Success != MMC_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != MMC_ReadBlocks(card, g_dataRead, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF("Read multiple data blocks failed.\r\n");
    }
}
MMC_Deinit(card);
```

## Data Structures

- struct `sd_card_t`  
*SD card state.* [More...](#)
- struct `sdio_card_t`  
*SDIO card state.* [More...](#)
- struct `mmc_card_t`  
*SD card state.* [More...](#)
- struct `mmc_boot_config_t`  
*MMC card boot configuration definition.* [More...](#)

## Macros

- #define `FSL_SDMMC_DRIVER_VERSION` (`MAKE_VERSION(2U, 1U, 2U)`) /\*2.1.2\*/
*Driver version.*
- #define `FSL_SDMMC_DEFAULT_BLOCK_SIZE` (512U)
*Default block size.*
- #define `HOST_NOT_SUPPORT` 0U
*use this define to indicate the host not support feature*
- #define `HOST_SUPPORT` 1U
*use this define to indicate the host support feature*
- #define `KHOST_DATABUSWIDTH1BIT` `kSDHC_DataBusWidth1Bit`
*1-bit mode*
- #define `KHOST_DATABUSWIDTH4BIT` `kSDHC_DataBusWidth4Bit`
*4-bit mode*

- #define **KHOST\_DATABUSWIDTH8BIT** kSDHC\_DataBusWidth8Bit  
*8-bit mode*
- #define **HOST\_STANDARD\_TUNING\_START** (0U)  
*standard tuning start point*
- #define **HOST\_TUNIG\_STEP** (1U)  
*standard tuning step*
- #define **HOST\_RETUNING\_TIMER\_COUNT** (4U)  
*Re-tuning timer.*

## Enumerations

- enum **\_sdmmc\_status** {
   
kStatus\_SDMMC\_NotSupportYet = MAKE\_STATUS(kStatusGroup\_SDMMC, 0U),
   
kStatus\_SDMMC\_TransferFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 1U),
   
kStatus\_SDMMC\_SetCardBlockSizeFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 2U),
   
kStatus\_SDMMC\_HostNotSupport = MAKE\_STATUS(kStatusGroup\_SDMMC, 3U),
   
kStatus\_SDMMC\_CardNotSupport = MAKE\_STATUS(kStatusGroup\_SDMMC, 4U),
   
kStatus\_SDMMC\_AllSendCidFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 5U),
   
kStatus\_SDMMC\_SendRelativeAddressFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 6U),
   
kStatus\_SDMMC\_SendCsdFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 7U),
   
kStatus\_SDMMC\_SelectCardFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 8U),
   
kStatus\_SDMMC\_SendScrFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 9U),
   
kStatus\_SDMMC\_SetDataBusWidthFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 10U),
   
kStatus\_SDMMC\_GoIdleFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 11U),
   
kStatus\_SDMMC\_HandShakeOperationConditionFailed,
   
kStatus\_SDMMC\_SendApplicationCommandFailed,
   
kStatus\_SDMMC\_SwitchFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 14U),
   
kStatus\_SDMMC\_StopTransmissionFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 15U),
   
kStatus\_SDMMC\_WaitWriteCompleteFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 16U),
   
kStatus\_SDMMC\_SetBlockCountFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 17U),
   
kStatus\_SDMMC\_SetRelativeAddressFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 18U),
   
kStatus\_SDMMC\_SwitchBusTimingFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 19U),
   
kStatus\_SDMMC\_SendExtendedCsdFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 20U),
   
kStatus\_SDMMC\_ConfigureBootFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 21U),
   
kStatus\_SDMMC\_ConfigureExtendedCsdFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 22-U),
   
kStatus\_SDMMC\_EnableHighCapacityEraseFailed,
   
kStatus\_SDMMC\_SendTestPatternFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 24U),
   
kStatus\_SDMMC\_ReceiveTestPatternFailed = MAKE\_STATUS(kStatusGroup\_SDMMC, 25U),
   
kStatus\_SDMMC\_SDIO\_ResponseError = MAKE\_STATUS(kStatusGroup\_SDMMC, 26U),
   
kStatus\_SDMMC\_SDIO\_InvalidArgument,
   
kStatus\_SDMMC\_SDIO\_SendOperationConditionFail,
   
kStatus\_SDMMC\_InvalidVoltage = MAKE\_STATUS(kStatusGroup\_SDMMC, 29U),
   
kStatus\_SDMMC\_SDIO\_SwitchHighSpeedFail = MAKE\_STATUS(kStatusGroup\_SDMMC, 30-U)

## Overview

U),  
kStatus\_SDMMC\_SDIO\_ReadCISFail = MAKE\_STATUS(kStatusGroup\_SDMMC, 31U),  
kStatus\_SDMMC\_SDIO\_InvalidCard = MAKE\_STATUS(kStatusGroup\_SDMMC, 32U),  
kStatus\_SDMMC\_TuningFail = MAKE\_STATUS(kStatusGroup\_SDMMC, 33U),  
kStatus\_SDMMC\_SwitchVoltageFail = MAKE\_STATUS(kStatusGroup\_SDMMC, 34U),  
kStatus\_SDMMC\_ReTuningRequest = MAKE\_STATUS(kStatusGroup\_SDMMC, 35U),  
kStatus\_SDMMC\_SetDriverStrengthFail = MAKE\_STATUS(kStatusGroup\_SDMMC, 36U),  
kStatus\_SDMMC\_SetPowerClassFail = MAKE\_STATUS(kStatusGroup\_SDMMC, 37U) }

*SD/MMC card API's running status.*

- enum `_sd_card_flag` {  
    kSD\_SupportHighCapacityFlag = (1U << 1U),  
    kSD\_Support4BitWidthFlag = (1U << 2U),  
    kSD\_SupportSdhcFlag = (1U << 3U),  
    kSD\_SupportSdxcFlag = (1U << 4U),  
    kSD\_SupportVoltage180v = (1U << 5U),  
    kSD\_SupportSetBlockCountCmd = (1U << 6U),  
    kSD\_SupportSpeedClassControlCmd = (1U << 7U) }

*SD card flags.*

- enum `_mmc_card_flag` {  
    kMMC\_SupportHighSpeed26MHZFlag = (1U << 0U),  
    kMMC\_SupportHighSpeed52MHZFlag = (1U << 1U),  
    kMMC\_SupportHighSpeedDDR52MHZ180V300VFlag = (1 << 2U),  
    kMMC\_SupportHighSpeedDDR52MHZ120VFlag = (1 << 3U),  
    kMMC\_SupportHS200200MHZ180VFlag = (1 << 4U),  
    kMMC\_SupportHS200200MHZ120VFlag = (1 << 5U),  
    kMMC\_SupportHS400DDR200MHZ180VFlag = (1 << 6U),  
    kMMC\_SupportHS400DDR200MHZ120VFlag = (1 << 7U),  
    kMMC\_SupportHighCapacityFlag = (1U << 8U),  
    kMMC\_SupportAlternateBootFlag = (1U << 9U),  
    kMMC\_SupportDDRBootFlag = (1U << 10U),  
    kMMC\_SupportHighSpeedBootFlag = (1U << 11U),  
    kMMC\_DataBusWidth4BitFlag = (1U << 12U),  
    kMMC\_DataBusWidth8BitFlag = (1U << 13U),  
    kMMC\_DataBusWidth1BitFlag = (1U << 14U) }

*MMC card flags.*

- enum `card_operation_voltage_t` {  
    kCARD\_OperationVoltageNone = 0U,  
    kCARD\_OperationVoltage330V = 1U,  
    kCARD\_OperationVoltage300V = 2U,  
    kCARD\_OperationVoltage180V = 3U }  
*card operation voltage*

*SDHC host capability.*

- enum `_host_endian_mode` {  
    kHOST\_EndianModeBig = 0U,  
    kHOST\_EndianModeHalfWordBig = 1U,

```
kHOST_EndianModeLittle = 2U }
```

*host Endian mode corresponding to driver define*

## SDCARD Function

- status\_t **SD\_Init** (sd\_card\_t \*card)  
*Initializes the card on a specific host controller.*
- void **SD\_Deinit** (sd\_card\_t \*card)  
*Deinitializes the card.*
- bool **SD\_CheckReadOnly** (sd\_card\_t \*card)  
*Checks whether the card is write-protected.*
- status\_t **SD\_ReadBlocks** (sd\_card\_t \*card, uint8\_t \*buffer, uint32\_t startBlock, uint32\_t blockCount)  
*Reads blocks from the specific card.*
- status\_t **SD\_WriteBlocks** (sd\_card\_t \*card, const uint8\_t \*buffer, uint32\_t startBlock, uint32\_t blockCount)  
*Writes blocks of data to the specific card.*
- status\_t **SD\_EraseBlocks** (sd\_card\_t \*card, uint32\_t startBlock, uint32\_t blockCount)  
*Erases blocks of the specific card.*

## MMCCARD Function

- status\_t **MMC\_Init** (mmc\_card\_t \*card)  
*Initializes the MMC card.*
- void **MMC\_Deinit** (mmc\_card\_t \*card)  
*Deinitializes the card.*
- bool **MMC\_CheckReadOnly** (mmc\_card\_t \*card)  
*Checks if the card is read-only.*
- status\_t **MMC\_ReadBlocks** (mmc\_card\_t \*card, uint8\_t \*buffer, uint32\_t startBlock, uint32\_t blockCount)  
*Reads data blocks from the card.*
- status\_t **MMC\_WriteBlocks** (mmc\_card\_t \*card, const uint8\_t \*buffer, uint32\_t startBlock, uint32\_t blockCount)  
*Writes data blocks to the card.*
- status\_t **MMC\_EraseGroups** (mmc\_card\_t \*card, uint32\_t startGroup, uint32\_t endGroup)  
*Erases groups of the card.*
- status\_t **MMC\_SelectPartition** (mmc\_card\_t \*card, mmc\_access\_partition\_t partitionNumber)  
*Selects the partition to access.*
- status\_t **MMC\_SetBootConfig** (mmc\_card\_t \*card, const mmc\_boot\_config\_t \*config)  
*Configures the boot activity of the card.*
- status\_t **SDIO\_CardInActive** (sdio\_card\_t \*card)  
*set SDIO card to inactive state*
- status\_t **SDIO\_IO\_Write\_Direct** (sdio\_card\_t \*card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \*data, bool raw)  
*IO direct write transfer function.*
- status\_t **SDIO\_IO\_Read\_Direct** (sdio\_card\_t \*card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \*data)  
*IO direct read transfer function.*
- status\_t **SDIO\_IO\_Write\_Extended** (sdio\_card\_t \*card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \*buffer, uint32\_t count, uint32\_t flags)

## Data Structure Documentation

- status\_t **SDIO\_IO\_Read\_Extended** (sdio\_card\_t \*card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \*buffer, uint32\_t count, uint32\_t flags)  
*IO extended write transfer function.*
- status\_t **SDIO\_GetCardCapability** (sdio\_card\_t \*card, sdio\_func\_num\_t func)  
*get SDIO card capability*
- status\_t **SDIO\_SetBlockSize** (sdio\_card\_t \*card, sdio\_func\_num\_t func, uint32\_t blockSize)  
*set SDIO card block size*
- status\_t **SDIO\_CardReset** (sdio\_card\_t \*card)  
*set SDIO card reset*
- status\_t **SDIO\_SetDataBusWidth** (sdio\_card\_t \*card, sdio\_bus\_width\_t busWidth)  
*set SDIO card data bus width*
- status\_t **SDIO\_SwitchToHighSpeed** (sdio\_card\_t \*card)  
*switch the card to high speed*
- status\_t **SDIO\_ReadCIS** (sdio\_card\_t \*card, sdio\_func\_num\_t func, const uint32\_t \*tupleList, uint32\_t tupleNum)  
*read SDIO card CIS for each function*
- status\_t **SDIO\_Init** (sdio\_card\_t \*card)  
*SDIO card init function.*
- status\_t **SDIO\_EnableIOInterrupt** (sdio\_card\_t \*card, sdio\_func\_num\_t func, bool enable)  
*enable IO interrupt*
- status\_t **SDIO\_EnableIO** (sdio\_card\_t \*card, sdio\_func\_num\_t func, bool enable)  
*enable IO and wait IO ready*
- status\_t **SDIO\_SelectIO** (sdio\_card\_t \*card, sdio\_func\_num\_t func)  
*select IO*
- status\_t **SDIO\_AbortIO** (sdio\_card\_t \*card, sdio\_func\_num\_t func)  
*Abort IO transfer.*
- void **SDIO\_DeInit** (sdio\_card\_t \*card)  
*SDIO card\_deinit.*

## adaptor function

- static status\_t **HOST\_NotSupport** (void \*parameter)  
*host not support function, this function is used for host not support feature*
- status\_t **CardInsertDetect** (HOST\_TYPE \*hostBase)  
*Detect card insert, only need for SD cases.*
- status\_t **HOST\_Init** (void \*host)  
*Init host controller.*
- void **HOST\_Deinit** (void \*host)  
*Deinit host controller.*

## 40.2 Data Structure Documentation

### 40.2.1 struct sd\_card\_t

Define the card structure including the necessary fields to identify and describe the card.

## Data Fields

- **HOST\_CONFIG host**  
*Host information.*
- **bool isHostReady**  
*use this flag to indicate if need host re-init or not*
- **uint32\_t busClock\_Hz**  
*SD bus clock frequency unitized in Hz.*
- **uint32\_t relativeAddress**  
*Relative address of the card.*
- **uint32\_t version**  
*Card version.*
- **uint32\_t flags**  
*Flags in \_sd\_card\_flag.*
- **uint32\_t rawCid [4U]**  
*Raw CID content.*
- **uint32\_t rawCsd [4U]**  
*Raw CSD content.*
- **uint32\_t rawScr [2U]**  
*Raw CSD content.*
- **uint32\_t ocr**  
*Raw OCR content.*
- **sd\_cid\_t cid**  
*CID.*
- **sd\_csd\_t csd**  
*CSD.*
- **sd\_scr\_t scr**  
*SCR.*
- **uint32\_t blockCount**  
*Card total block number.*
- **uint32\_t blockSize**  
*Card block size.*
- **sd\_timing\_mode\_t currentTiming**  
*current timing mode*
- **sd\_driver\_strength\_t driverStrength**  
*driver strength*
- **sd\_max\_current\_t maxCurrent**  
*card current limit*
- **card\_operation\_voltage\_t operationVoltage**  
*card operation voltage*

### 40.2.2 struct sdio\_card\_t

Define the card structure including the necessary fields to identify and describe the card.

## Data Fields

- **HOST\_CONFIG host**

## Data Structure Documentation

- *Host information.*
  - bool **isHostReady**  
*use this flag to indicate if need host re-init or not*
  - bool **memPresentFlag**  
*indicate if memory present*
  - uint32\_t **busClock\_Hz**  
*SD bus clock frequency united in Hz.*
  - uint32\_t **relativeAddress**  
*Relative address of the card.*
  - uint8\_t **sdVersion**  
*SD version.*
  - uint8\_t **sdioVersion**  
*SDIO version.*
  - uint8\_t **cccrVersioin**  
*CCCR version.*
  - uint8\_t **ioTotalNumber**  
*total number of IO function*
  - uint32\_t **ccerflags**  
*Flags in \_sd\_card\_flag.*
  - uint32\_t **io0blockSize**  
*record the io0 block size*
  - uint32\_t **ocr**  
*Raw OCR content, only 24bit available for SDIO card.*
  - uint32\_t **commonCISPointer**  
*point to common CIS*
  - sdio\_fbr\_t **ioFBR** [7U]  
*FBR table.*
  - sdio\_common\_cis\_t **commonCIS**  
*CIS table.*
  - sdio\_func\_cis\_t **funcCIS** [7U]  
*function CIS table*

### 40.2.3 struct mmc\_card\_t

Define the card structure including the necessary fields to identify and describe the card.

#### Data Fields

- HOST\_CONFIG **host**  
*Host information.*
- bool **isHostReady**  
*use this flag to indicate if need host re-init or not*
- uint32\_t **busClock\_Hz**  
*MMC bus clock united in Hz.*
- uint32\_t **relativeAddress**  
*Relative address of the card.*
- bool **enablePreDefinedBlockCount**  
*Enable PRE-DEFINED block count when read/write.*

- `uint32_t flags`  
*Capability flag in \_mmc\_card\_flag.*
- `uint32_t rawCid [4U]`  
*Raw CID content.*
- `uint32_t rawCsd [4U]`  
*Raw CSD content.*
- `uint32_t rawExtendedCsd [MMC_EXTENDED_CSD_BYTES/4U]`  
*Raw MMC Extended CSD content.*
- `uint32_t ocr`  
*Raw OCR content.*
- `mmc_cid_t cid`  
*CID.*
- `mmc_csd_t csd`  
*CSD.*
- `mmc_extended_csd_t extendedCsd`  
*Extended CSD.*
- `uint32_t blockSize`  
*Card block size.*
- `uint32_t userPartitionBlocks`  
*Card total block number in user partition.*
- `uint32_t bootPartitionBlocks`  
*Boot partition size united as block size.*
- `uint32_t eraseGroupBlocks`  
*Erase group size united as block size.*
- `mmc_access_partition_t currentPartition`  
*Current access partition.*
- `mmc_voltage_window_t hostVoltageWindow`  
*Host voltage window.*
- `mmc_high_speed_timing_t currentTiming`  
*indicate the current host timing mode*

#### 40.2.4 struct mmc\_boot\_config\_t

##### Data Fields

- `bool enableBootAck`  
*Enable boot ACK.*
- `mmc_boot_partition_enable_t bootPartition`  
*Boot partition.*
- `bool retainBootBusWidth`  
*If retain boot bus width.*
- `mmc_data_bus_width_t bootDataBusWidth`  
*Boot data bus width.*

#### 40.3 Macro Definition Documentation

##### 40.3.1 #define FSL\_SDMMC\_DRIVER\_VERSION (MAKE\_VERSION(2U, 1U, 2U)) /\*2.1.2\*/

## Enumeration Type Documentation

### 40.4 Enumeration Type Documentation

#### 40.4.1 enum \_sdmmc\_status

Enumerator

*kStatus\_SDMMC\_NotSupportYet* Haven't supported.  
*kStatus\_SDMMC\_TransferFailed* Send command failed.  
*kStatus\_SDMMC\_SetCardBlockSizeFailed* Set block size failed.  
*kStatus\_SDMMC\_HostNotSupport* Host doesn't support.  
*kStatus\_SDMMC\_CardNotSupport* Card doesn't support.  
*kStatus\_SDMMC\_AllSendCidFailed* Send CID failed.  
*kStatus\_SDMMC\_SendRelativeAddressFailed* Send relative address failed.  
*kStatus\_SDMMC\_SendCsdFailed* Send CSD failed.  
*kStatus\_SDMMC\_SelectCardFailed* Select card failed.  
*kStatus\_SDMMC\_SendScrFailed* Send SCR failed.  
*kStatus\_SDMMC\_SetDataBusWidthFailed* Set bus width failed.  
*kStatus\_SDMMC\_GoIdleFailed* Go idle failed.  
*kStatus\_SDMMC\_HandShakeOperationConditionFailed* Send Operation Condition failed.  
*kStatus\_SDMMC\_SendApplicationCommandFailed* Send application command failed.  
*kStatus\_SDMMC\_SwitchFailed* Switch command failed.  
*kStatus\_SDMMC\_StopTransmissionFailed* Stop transmission failed.  
*kStatus\_SDMMC\_WaitWriteCompleteFailed* Wait write complete failed.  
*kStatus\_SDMMC\_SetBlockCountFailed* Set block count failed.  
*kStatus\_SDMMC\_SetRelativeAddressFailed* Set relative address failed.  
*kStatus\_SDMMC\_SwitchBusTimingFailed* Switch high speed failed.  
*kStatus\_SDMMC\_SendExtendedCsdFailed* Send EXT\_CSD failed.  
*kStatus\_SDMMC\_ConfigureBootFailed* Configure boot failed.  
*kStatus\_SDMMC\_ConfigureExtendedCsdFailed* Configure EXT\_CSD failed.  
*kStatus\_SDMMC\_EnableHighCapacityEraseFailed* Enable high capacity erase failed.  
*kStatus\_SDMMC\_SendTestPatternFailed* Send test pattern failed.  
*kStatus\_SDMMC\_ReceiveTestPatternFailed* Receive test pattern failed.  
*kStatus\_SDMMC\_SDIO\_ResponseError* sdio response error  
*kStatus\_SDMMC\_SDIO\_InvalidArgument* sdio invalid argument response error  
*kStatus\_SDMMC\_SDIO\_SendOperationConditionFail* sdio send operation condition fail  
*kStatus\_SDMMC\_InvalidVoltage* invalld voltage  
*kStatus\_SDMMC\_SDIO\_SwitchHighSpeedFail* switch to high speed fail  
*kStatus\_SDMMC\_SDIO\_ReadCISFail* read CIS fail  
*kStatus\_SDMMC\_SDIO\_InvalidCard* invalld SDIO card  
*kStatus\_SDMMC\_TuningFail* tuning fail  
*kStatus\_SDMMC\_SwitchVoltageFail* switch voltage fail  
*kStatus\_SDMMC\_ReTuningRequest* retuning request  
*kStatus\_SDMMC\_SetDriverStrengthFail* set driver strength fail  
*kStatus\_SDMMC\_SetPowerClassFail* set power class fail

#### 40.4.2 enum \_sd\_card\_flag

Enumerator

*kSD\_SupportHighCapacityFlag* Support high capacity.  
*kSD\_Support4BitWidthFlag* Support 4-bit data width.  
*kSD\_SupportSdhcFlag* Card is SDHC.  
*kSD\_SupportSdxcFlag* Card is SDXC.  
*kSD\_SupportVoltage180v* card support 1.8v voltage  
*kSD\_SupportSetBlockCountCmd* card support cmd23 flag  
*kSD\_SupportSpeedClassControlCmd* card support speed class control flag

#### 40.4.3 enum \_mmc\_card\_flag

Enumerator

*kMMC\_SupportHighSpeed26MHZFlag* Support high speed 26MHZ.  
*kMMC\_SupportHighSpeed52MHZFlag* Support high speed 52MHZ.  
*kMMC\_SupportHighSpeedDDR52MHZ180V300VFlag* ddr 52MHZ 1.8V or 3.0V  
*kMMC\_SupportHighSpeedDDR52MHZ120VFlag* DDR 52MHZ 1.2V.  
*kMMC\_SupportHS200200MHZ180VFlag* HS200 ,200MHZ,1.8V.  
*kMMC\_SupportHS200200MHZ120VFlag* HS200, 200MHZ, 1.2V.  
*kMMC\_SupportHS400DDR200MHZ180VFlag* HS400, DDR, 200MHZ,1.8V.  
*kMMC\_SupportHS400DDR200MHZ120VFlag* HS400, DDR, 200MHZ,1.2V.  
*kMMC\_SupportHighCapacityFlag* Support high capacity.  
*kMMC\_SupportAlternateBootFlag* Support alternate boot.  
*kMMC\_SupportDDRBootFlag* support DDR boot flag  
*kMMC\_SupportHighSpeedBootFlag* support high speed boot flag  
*kMMC\_DataBusWidth4BitFlag* current data bus is 4 bit mode  
*kMMC\_DataBusWidth8BitFlag* current data bus is 8 bit mode  
*kMMC\_DataBusWidth1BitFlag* current data bus is 1 bit mode

#### 40.4.4 enum card\_operation\_voltage\_t

Enumerator

*kCARD\_OperationVoltageNone* indicate current voltage setting is not setting bu suser  
*kCARD\_OperationVoltage330V* card operation voltage around 3.3v  
*kCARD\_OperationVoltage300V* card operation voltage around 3.0v  
*kCARD\_OperationVoltage180V* card operation voltage around 31.8v

## Function Documentation

### 40.4.5 enum \_host\_endian\_mode

Enumerator

*kHOST\_EndianModeBig* Big endian mode.

*kHOST\_EndianModeHalfWordBig* Half word big endian mode.

*kHOST\_EndianModeLittle* Little endian mode.

## 40.5 Function Documentation

### 40.5.1 status\_t SD\_Init( sd\_card\_t \* card )

This function initializes the card on a specific host controller.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_Not-SupportYet</i>	Card not support.
<i>kStatus_SDMMC_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_SDMMC_All-SendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_Send-RelativeAddressFailed</i>	Send relative address failed.
<i>kStatus_SDMMC_Send-CsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_Select-CardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_Send-ScrFailed</i>	Send SCR failed.

<i>kStatus_SDMMC_SetBusWidthFailed</i>	Set bus width failed.
<i>kStatus_SDMMC_SwitchHighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_SetCardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

#### 40.5.2 void SD\_Deinit ( *sd\_card\_t* \* *card* )

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

#### 40.5.3 bool SD\_CheckReadOnly ( *sd\_card\_t* \* *card* )

This function checks if the card is write-protected via the CSD register.

Parameters

<i>card</i>	The specific card.
-------------	--------------------

Return values

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

#### 40.5.4 *status\_t* SD\_ReadBlocks ( *sd\_card\_t* \* *card*, *uint8\_t* \* *buffer*, *uint32\_t* *startBlock*, *uint32\_t* *blockCount* )

This function reads blocks from the specific card with default block size defined by the SDHC\_CARD\_DEFAULT\_BLOCK\_SIZE.

## Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save the data read from card.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to read.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

### 40.5.5 **status\_t SD\_WriteBlocks ( *sd\_card\_t \* card, const uint8\_t \* buffer, uint32\_t startBlock, uint32\_t blockCount* )**

This function writes blocks to the specific card with default block size 512 bytes.

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer holding the data to be written to the card.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to write.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_NotSupportYet</i>	Not support now.
<i>kStatus_SDMMC_CardNotSupport</i>	Card not support.
<i>kStatus_SDMMC_WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_StopTransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

#### 40.5.6 **status\_t SD\_EraseBlocks ( sd\_card\_t \* card, uint32\_t startBlock, uint32\_t blockCount )**

This function erases blocks of the specific card with default block size 512 bytes.

Parameters

<i>card</i>	Card descriptor.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to erase.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.

## Function Documentation

<i>kStatus_SDMMC_WaitWriteCompleteFailed</i>	Send status failed.
<i>kStatus_Success</i>	Operate successfully.

### 40.5.7 **status\_t MMC\_Init ( mmc\_card\_t \* *card* )**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_GoIdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_SendOperationConditionFailed</i>	Send operation condition failed.
<i>kStatus_SDMMC_AllSendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_SetRelativeAddressFailed</i>	Set relative address failed.
<i>kStatus_SDMMC_SendCsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_CardNotSupport</i>	Card not support.
<i>kStatus_SDMMC_SelectCardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_SendExtendedCsdFailed</i>	Send EXT_CSD failed.
<i>kStatus_SDMMC_SetBusWidthFailed</i>	Set bus width failed.

<i>kStatus_SDMMC_Switch-HighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_Set-CardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

#### 40.5.8 void MMC\_Deinit ( mmc\_card\_t \* *card* )

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

#### 40.5.9 bool MMC\_CheckReadOnly ( mmc\_card\_t \* *card* )

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

#### 40.5.10 status\_t MMC\_ReadBlocks ( mmc\_card\_t \* *card*, uint8\_t \* *buffer*, uint32\_t *startBlock*, uint32\_t *blockCount* )

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save data.
<i>startBlock</i>	The start block index.

## Function Documentation

<i>blockCount</i>	The number of blocks to read.
-------------------	-------------------------------

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Set-BlockCountFailed</i>	Set block count failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

### 40.5.11 **status\_t MMC\_WriteBlocks ( mmc\_card\_t \* *card*, const uint8\_t \* *buffer*, uint32\_t *startBlock*, uint32\_t *blockCount* )**

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save data blocks.
<i>startBlock</i>	Start block number to write.
<i>blockCount</i>	Block count.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Set-BlockCountFailed</i>	Set block count failed.

<i>kStatus_SDMMC_WaitWriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_StopTransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

#### 40.5.12 **status\_t MMC\_EraseGroups ( mmc\_card\_t \* *card*, uint32\_t *startGroup*, uint32\_t *endGroup* )**

Erase group is the smallest erase unit in MMC card. The erase range is [startGroup, endGroup].

Parameters

<i>card</i>	Card descriptor.
<i>startGroup</i>	Start group number.
<i>endGroup</i>	End group number.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_WaitWriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_TransferFailed</i>	Transfer failed.
<i>kStatus_Success</i>	Operate successfully.

#### 40.5.13 **status\_t MMC\_SelectPartition ( mmc\_card\_t \* *card*, mmc\_access\_partition\_t *partitionNumber* )**

Parameters

## Function Documentation

<i>card</i>	Card descriptor.
<i>partition-Number</i>	The partition number.

Return values

<i>kStatus_SDMMC_ConfigureExtendedCsdFailed</i>	Configure EXT_CSD failed.
<i>kStatus_Success</i>	Operate successfully.

### 40.5.14 **status\_t MMC\_SetBootConfig ( mmc\_card\_t \* *card*, const mmc\_boot\_config\_t \* *config* )**

Parameters

<i>card</i>	Card descriptor.
<i>config</i>	Boot configuration structure.

Return values

<i>kStatus_SDMMC_NotSupportYet</i>	Not support now.
<i>kStatus_SDMMC_ConfigureExtendedCsdFailed</i>	Configure EXT_CSD failed.
<i>kStatus_SDMMC_ConfigureBootFailed</i>	Configure boot failed.
<i>kStatus_Success</i>	Operate successfully.

### 40.5.15 **status\_t SDIO\_CardInActive ( sdio\_card\_t \* *card* )**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

#### 40.5.16 **status\_t SDIO\_IO\_Write\_Direct ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \* data, bool raw* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>register</i>	address
<i>the</i>	data pointer to write
<i>raw</i>	flag, indicate read after write or write only

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

#### 40.5.17 **status\_t SDIO\_IO\_Read\_Direct ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \* data* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

## Function Documentation

<i>register</i>	address
<i>data</i>	pointer to read

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

### 40.5.18 **status\_t SDIO\_IO\_Write\_Extended ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \* buffer, uint32\_t count, uint32\_t flags* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>register</i>	address
<i>data</i>	buffer to write
<i>data</i>	count
<i>write</i>	flags

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_SDMMC_SDIO_InvalidArgument</i>	
<i>kStatus_Success</i>	

### 40.5.19 **status\_t SDIO\_IO\_Read\_Extended ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, uint32\_t regAddr, uint8\_t \* buffer, uint32\_t count, uint32\_t flags* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>register</i>	address
<i>data</i>	buffer to read
<i>data</i>	count
<i>write</i>	flags

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_SDMMC_SDIO_InvalidArgument</i>	
<i>kStatus_Success</i>	

#### 40.5.20 **status\_t SDIO\_GetCardCapability ( *sdio\_card\_t \* card, sdio\_func\_num\_t func* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

#### 40.5.21 **status\_t SDIO\_SetBlockSize ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, uint32\_t blockSize* )**

## Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	io number
<i>block</i>	size

Return values

<i>kStatus_SDMMC_SetCardBlockSizeFailed</i>	
<i>kStatus_SDMMC_SDIO_InvalidArgument</i>	
<i>kStatus_Success</i>	

### 40.5.22 status\_t SDIO\_CardReset ( *sdio\_card\_t \* card* )

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

### 40.5.23 status\_t SDIO\_SetDataBusWidth ( *sdio\_card\_t \* card, sdio\_bus\_width\_t busWidth* )

Parameters

<i>card</i>	Card descriptor.
<i>data</i>	bus width

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

#### 40.5.24 **status\_t SDIO\_SwitchToHighSpeed ( *sdio\_card\_t \* card* )**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_SDMMC_SDIO_SwitchHighSpeedFail</i>	
<i>kStatus_Success</i>	

#### 40.5.25 **status\_t SDIO\_ReadCIS ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, const uint32\_t \* tupleList, uint32\_t tupleNum* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	io number
<i>tuple</i>	code list
<i>tuple</i>	code number

Return values

<i>kStatus_SDMMC_SDIO_ReadCISFail</i>	
---------------------------------------	--

## Function Documentation

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

### 40.5.26 **status\_t SDIO\_Init( sdio\_card\_t \* card )**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_GoIdleFailed</i>	
<i>kStatus_SDMMC_HandShakeOperationConditionFailed</i>	
<i>kStatus_SDMMC_SDIO_InvalidCard</i>	
<i>kStatus_SDMMC_SDIO_InvalidVoltage</i>	
<i>kStatus_SDMMC_SendRelativeAddressFailed</i>	
<i>kStatus_SDMMC_SelectCardFailed</i>	
<i>kStatus_SDMMC_SDIO_SwitchHighSpeedFail</i>	
<i>kStatus_SDMMC_SDIO_ReadCISFail</i>	
<i>kStatus_SDMMC_TransferFailed</i>	

<i>kStatus_Success</i>
------------------------

#### 40.5.27 **status\_t SDIO\_EnableIOInterrupt ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, bool enable* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>enable/disable</i>	flag

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

#### 40.5.28 **status\_t SDIO\_EnableIO ( *sdio\_card\_t \* card, sdio\_func\_num\_t func, bool enable* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>enable/disable</i>	flag

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

#### 40.5.29 **status\_t SDIO\_SelectIO ( *sdio\_card\_t \* card, sdio\_func\_num\_t func* )**

## Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

### 40.5.30 **status\_t SDIO\_AbortIO ( *sdio\_card\_t \* card, sdio\_func\_num\_t func* )**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

### 40.5.31 **void SDIO\_DelInit ( *sdio\_card\_t \* card* )**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

### 40.5.32 **static status\_t HOST\_NotSupport ( *void \* parameter* ) [inline], [static]**

Parameters

<i>void</i>	parameter ,used to avoid build warning
-------------	--

Return values

<i>kStatus_Fail,host</i>	do not support
--------------------------	----------------

#### 40.5.33 **status\_t CardInsertDetect ( HOST\_TYPE \* *hostBase* )**

Parameters

<i>hostBase</i>	the pointer to host base address
-----------------	----------------------------------

Return values

<i>kStatus_Success</i>	detect card insert
<i>kStatus_Fail</i>	card insert event fail

#### 40.5.34 **status\_t HOST\_Init ( void \* *host* )**

Parameters

<i>host</i>	the pointer to host structure in card structure.
-------------	--

Return values

<i>kStatus_Success</i>	host init success
<i>kStatus_Fail</i>	event fail

#### 40.5.35 **void HOST\_Deinit ( void \* *host* )**

Parameters

## Function Documentation

<i>host</i>	the pointer to host structure in card structure.
-------------	--

# Chapter 41

## SPI based Secure Digital Card (SDSPI)

### 41.1 Overview

The MCUXpresso SDK provides a driver to access the Secure Digital Card based on the SPI driver.

### Function groups

This function group implements the SD card functional API in the SPI mode.

### Typical use case

```
/* SPI_Init(). */

/* Register the SDSPI driver callback. */

/* Initializes card. */
if (kStatus_Success != SDSPI_Init(card))
{
    SDSPI_Deinit(card)
    return;
}

/* Read/Write card */
memset(g_testWriteBuffer, 0x17U, sizeof(g_testWriteBuffer));

while (true)
{
    memset(g_testReadBuffer, 0U, sizeof(g_testReadBuffer));

    SDSPI_WriteBlocks(card, g_testWriteBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    SDSPI_ReadBlocks(card, g_testReadBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    if (memcmp(g_testReadBuffer, g_testReadBuffer, sizeof(g_testWriteBuffer)))
    {
        break;
    }
}
```

### Data Structures

- struct `sdspi_command_t`  
*SDSPI command.* [More...](#)
- struct `sdspi_host_t`  
*SDSPI host state.* [More...](#)
- struct `sdspi_card_t`  
*SD Card Structure.* [More...](#)

## Overview

### Enumerations

- enum `_sdspi_status` {  
    `kStatus_SDSPI_SetFrequencyFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 0U),  
    `kStatus_SDSPI_ExchangeFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 1U),  
    `kStatus_SDSPI_WaitReadyFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 2U),  
    `kStatus_SDSPI_ResponseError` = MAKE\_STATUS(kStatusGroup\_SDSPI, 3U),  
    `kStatus_SDSPI_WriteProtected` = MAKE\_STATUS(kStatusGroup\_SDSPI, 4U),  
    `kStatus_SDSPI_GoIdleFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 5U),  
    `kStatus_SDSPI_SendCommandFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 6U),  
    `kStatus_SDSPI_ReadFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 7U),  
    `kStatus_SDSPI_WriteFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 8U),  
    `kStatus_SDSPI_SendInterfaceConditionFailed`,  
    `kStatus_SDSPI_SendOperationConditionFailed`,  
    `kStatus_SDSPI_ReadOcrFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 11U),  
    `kStatus_SDSPI_SetBlockSizeFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 12U),  
    `kStatus_SDSPI_SendCsdFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 13U),  
    `kStatus_SDSPI_SendCidFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 14U),  
    `kStatus_SDSPI_StopTransmissionFailed` = MAKE\_STATUS(kStatusGroup\_SDSPI, 15U),  
    `kStatus_SDSPI_SendApplicationCommandFailed` }

*SDSPI API status.*

- enum `_sdspi_card_flag` {  
    `kSDSPI_SupportHighCapacityFlag` = (1U << 0U),  
    `kSDSPI_SupportSdhcFlag` = (1U << 1U),  
    `kSDSPI_SupportSdxcFlag` = (1U << 2U),  
    `kSDSPI_SupportSdscFlag` = (1U << 3U) }

*SDSPI card flag.*

- enum `sdspi_response_type_t` {  
    `kSDSPI_ResponseTypeR1` = 0U,  
    `kSDSPI_ResponseTypeR1b` = 1U,  
    `kSDSPI_ResponseTypeR2` = 2U,  
    `kSDSPI_ResponseTypeR3` = 3U,  
    `kSDSPI_ResponseTypeR7` = 4U }

*SDSPI response type.*

### SDSPI Function

- `status_t SDSPI_Init (sdspi_card_t *card)`  
*Initializes the card on a specific SPI instance.*
- `void SDSPI_Deinit (sdspi_card_t *card)`  
*Deinitializes the card.*
- `bool SDSPI_CheckReadOnly (sdspi_card_t *card)`  
*Checks whether the card is write-protected.*
- `status_t SDSPI_ReadBlocks (sdspi_card_t *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`  
*Reads blocks from the specific card.*
- `status_t SDSPI_WriteBlocks (sdspi_card_t *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`

*Writes blocks of data to the specific card.*

## 41.2 Data Structure Documentation

### 41.2.1 struct sdspi\_command\_t

#### Data Fields

- `uint8_t index`  
*Command index.*
- `uint32_t argument`  
*Command argument.*
- `uint8_t responseType`  
*Response type.*
- `uint8_t response [5U]`  
*Response content.*

### 41.2.2 struct sdspi\_host\_t

#### Data Fields

- `uint32_t busBaudRate`  
*Bus baud rate.*
- `status_t(* setFrequency )(uint32_t frequency)`  
*Set frequency of SPI.*
- `status_t(* exchange )(uint8_t *in, uint8_t *out, uint32_t size)`  
*Exchange data over SPI.*
- `uint32_t(* getCurrentMilliseconds )(void)`  
*Get current time in milliseconds.*

### 41.2.3 struct sdspi\_card\_t

Define the card structure including the necessary fields to identify and describe the card.

#### Data Fields

- `sdspi_host_t * host`  
*Host state information.*
- `uint32_t relativeAddress`  
*Relative address of the card.*
- `uint32_t flags`  
*Flags defined in \_sdspi\_card\_flag.*
- `uint8_t rawCid [16U]`  
*Raw CID content.*
- `uint8_t rawCsd [16U]`

## Enumeration Type Documentation

- `uint8_t rawScr [8U]`  
*Raw CSD content.*
- `uint32_t ocr`  
*Raw OCR content.*
- `sd_cid_t cid`  
*CID.*
- `sd_csd_t csd`  
*CSD.*
- `sd_scr_t scr`  
*SCR.*
- `uint32_t blockCount`  
*Card total block number.*
- `uint32_t blockSize`  
*Card block size.*

### 41.2.3.0.0.85 Field Documentation

#### 41.2.3.0.0.85.1 `uint32_t sdspi_card_t::flags`

## 41.3 Enumeration Type Documentation

### 41.3.1 `enum _sdspi_status`

Enumerator

- `kStatus_SDSPI_SetFrequencyFailed` Set frequency failed.
- `kStatus_SDSPI_ExchangeFailed` Exchange data on SPI bus failed.
- `kStatus_SDSPI_WaitReadyFailed` Wait card ready failed.
- `kStatus_SDSPI_ResponseError` Response is error.
- `kStatus_SDSPI_WriteProtected` Write protected.
- `kStatus_SDSPI_GoIdleFailed` Go idle failed.
- `kStatus_SDSPI_SendCommandFailed` Send command failed.
- `kStatus_SDSPI_ReadFailed` Read data failed.
- `kStatus_SDSPI_WriteFailed` Write data failed.
- `kStatus_SDSPI_SendInterfaceConditionFailed` Send interface condition failed.
- `kStatus_SDSPI_SendOperationConditionFailed` Send operation condition failed.
- `kStatus_SDSPI_ReadOcrFailed` Read OCR failed.
- `kStatus_SDSPI_SetBlockSizeFailed` Set block size failed.
- `kStatus_SDSPI_SendCsdFailed` Send CSD failed.
- `kStatus_SDSPI_SendCidFailed` Send CID failed.
- `kStatus_SDSPI_StopTransmissionFailed` Stop transmission failed.
- `kStatus_SDSPI_SendApplicationCommandFailed` Send application command failed.

### 41.3.2 enum \_sdspi\_card\_flag

Enumerator

*kSDSPI\_SupportHighCapacityFlag* Card is high capacity.

*kSDSPI\_SupportSdhcFlag* Card is SDHC.

*kSDSPI\_SupportSdxcFlag* Card is SDXC.

*kSDSPI\_SupportSdscFlag* Card is SDSC.

### 41.3.3 enum sdspi\_response\_type\_t

Enumerator

*kSDSPI\_ResponseOfTypeR1* Response 1.

*kSDSPI\_ResponseOfTypeR1b* Response 1 with busy.

*kSDSPI\_ResponseOfTypeR2* Response 2.

*kSDSPI\_ResponseOfTypeR3* Response 3.

*kSDSPI\_ResponseOfTypeR7* Response 7.

## 41.4 Function Documentation

### 41.4.1 status\_t SDSPI\_Init ( *sdspi\_card\_t \* card* )

This function initializes the card on a specific SPI instance.

Parameters

<i>card</i>	Card descriptor
-------------	-----------------

Return values

<i>kStatus_SDSPi_SetFrequencyFailed</i>	Set frequency failed.
<i>kStatus_SDSPi_GoIdleFailed</i>	Go idle failed.
<i>kStatus_SDSPi_SendInterfaceConditionFailed</i>	Send interface condition failed.

## Function Documentation

<i>kStatus_SDSPI_SendOperationConditionFailed</i>	Send operation condition failed.
<i>kStatus_Timeout</i>	Send command timeout.
<i>kStatus_SDSPI_NotSupportYet</i>	Not support yet.
<i>kStatus_SDSPI_ReadOcrFailed</i>	Read OCR failed.
<i>kStatus_SDSPI_SetBlockSizeFailed</i>	Set block size failed.
<i>kStatus_SDSPI_SendCsdFailed</i>	Send CSD failed.
<i>kStatus_SDSPI_SendCidFailed</i>	Send CID failed.
<i>kStatus_Success</i>	Operate successfully.

### 41.4.2 void SDSPI\_Deinit ( *sdspi\_card\_t \* card* )

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor
-------------	-----------------

### 41.4.3 bool SDSPI\_CheckReadOnly ( *sdspi\_card\_t \* card* )

This function checks if the card is write-protected via CSD register.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

---

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

#### 41.4.4 **status\_t SDSPI\_ReadBlocks ( *sdspi\_card\_t \* card, uint8\_t \* buffer, uint32\_t startBlock, uint32\_t blockCount* )**

This function reads blocks from specific card.

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	the buffer to hold the data read from card
<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to read

Return values

<i>kStatus_SDSPI_Send-CommandFailed</i>	Send command failed.
<i>kStatus_SDSPI_Read-Failed</i>	Read data failed.
<i>kStatus_SDSPI_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

#### 41.4.5 **status\_t SDSPI\_WriteBlocks ( *sdspi\_card\_t \* card, uint8\_t \* buffer, uint32\_t startBlock, uint32\_t blockCount* )**

This function writes blocks to specific card

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	the buffer holding the data to be written to the card

## Function Documentation

<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to write

Return values

<i>kStatus_SDSPI_WriteProtected</i>	Card is write protected.
<i>kStatus_SDSPI_SendCommandFailed</i>	Send command failed.
<i>kStatus_SDSPI_ResponseError</i>	Response is error.
<i>kStatus_SDSPI_WriteFailed</i>	Write data failed.
<i>kStatus_SDSPI_ExchangeFailed</i>	Exchange data over SPI failed.
<i>kStatus_SDSPI_WaitReadyFailed</i>	Wait card to be ready status failed.
<i>kStatus_Success</i>	Operate successfully.

# Chapter 42

## Debug Console

### 42.1 Overview

This chapter describes the programming interface of the debug console driver.

The debug console enables debug log messages to be output via the specified peripheral with frequency of the peripheral source clock and base address at the specified baud rate. Additionally, it provides input and output functions to scan and print formatted data.

### 42.2 Function groups

#### 42.2.1 Initialization

To initialize the debug console, call the DbgConsole\_Init() function with these parameters. This function automatically enables the module and the clock.

```
/*
 * @brief Initializes the the peripheral used to debug messages.
 *
 * @param baseAddr      Indicates which address of the peripheral is used to send debug messages.
 * @param baudRate     The desired baud rate in bits per second.
 * @param device        Low level device type for the debug console, can be one of:
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_UART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPUART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPSCI,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_USBCDC.
 * @param clkSrcFreq   Frequency of peripheral source clock.
 *
 * @return             Whether initialization was successful or not.
 */
status_t DbgConsole_Init(uint32_t baseAddr, uint32_t baudRate, uint8_t device, uint32_t clkSrcFreq)
```

Selects the supported debug console hardware device type, such as

```
DEBUG_CONSOLE_DEVICE_TYPE_NONE
DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
DEBUG_CONSOLE_DEVICE_TYPE_UART
DEBUG_CONSOLE_DEVICE_TYPE_LPUART
DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
```

After the initialization is successful, stdout and stdin are connected to the selected peripheral. The debug console state is stored in the debug\_console\_state\_t structure, such as shown here.

```
typedef struct DebugConsoleState
{
    uint8_t                  type;
    void*                   base;
    debug_console_ops_t     ops;
} debug_console_state_t;
```

## Function groups

This example shows how to call the DbgConsole\_Init() given the user configuration structure.

```
uint32_t uartClkSrcFreq = CLOCK_GetFreq(BOARD_DEBUG_UART_CLKSRC);  
  
DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, BOARD_DEBUG_UART_BAUDRATE, DEBUG_CONSOLE_DEVICE_TYPE_UART,  
                 uartClkSrcFreq);
```

### 42.2.2 Advanced Feature

The debug console provides input and output functions to scan and print formatted data.

- Support a format specifier for PRINTF following this prototype " %[flags][width][.precision][length]specifier", which is explained below

flags	Description
-	Left-justified within the given field width. Right-justified is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is written, a blank space is inserted before the value.
#	Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E, and f specifiers this is the number of digits to be printed after the decimal point. For g and G specifiers This is the maximum number of significant digits to be printed. For s this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
Do not support	

specifier	Description
d or i	Signed decimal integer
f	Decimal floating point
F	Decimal floating point capital letters
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer capital letters
o	Signed octal
b	Binary value
p	Pointer address
u	Unsigned decimal integer
c	Character
s	String of characters
n	Nothing printed

## Function groups

- Support a format specifier for SCANF following this prototype " %[\*][width][length]specifier", which is explained below

*	Description
An optional starting asterisk indicates that the data is to be read from the stream but ignored. In other words, it is not stored in the corresponding argument.	

width	Description
This specifies the maximum number of characters to be read in the current reading operation.	

length	Description
hh	The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X).
h	The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X).
l	The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
ll	The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G).
j or z or t	Not supported

specifier	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *

specifier	Qualifying Input	Type of argument
i	Integer: : Number optionally preceded with a + or - sign	int *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
a, A, e, E, f, F, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab).	char *
u	Unsigned decimal integer.	unsigned int *

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file.

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(const char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the MCUXpresso SDK printf/scanf.

```
#if SDK_DEBUGCONSOLE      /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF            DbgConsole_Printf
#define SCANF              DbgConsole_Scanf
#define PUTCHAR            DbgConsole_Putchar
#define GETCHAR            DbgConsole_Getchar
#else                      /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF            printf
#define SCANF              scanf
#define PUTCHAR            putchar
#define GETCHAR            getchar
#endif /* SDK_DEBUGCONSOLE */
```

## 42.3 Typical use case

### Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

## Typical use case

### Some examples use the PRINTF function

Statement prints the string format.

```
PRINTF("%s %s\r\n", "Hello", "world!");
```

Statement prints the hexadecimal format/

```
PRINTF("0x%02X hexadecimal number equivalents 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\r\nTime: %u ticks %2.5f milliseconds\r\nDONE\r", "1 day", 86400, 86.4);
```

### Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

### Print out failure messages using KSDK \_\_assert\_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \"% %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file,
           line, func);
    for (;;) {}
}
```

### Note:

To use 'printf' and 'scanf' for GNUC Base, add file '**fsl\_sbrk.c**' in path: ..\{package}\devices\{subset}\utilities\fsl-sbrk.c to your project.

## Modules

- Semihosting

## 42.4 Semihosting

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism can be used, for example, to enable functions in the C library, such as printf() and scanf(), to use the screen and keyboard of the host rather than having a screen and keyboard on the target system.

### 42.4.1 Guide Semihosting for IAR

**NOTE:** After the setting both "printf" and "scanf" are available for debugging.

#### Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

#### Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7.
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

#### Step 3: Starting semihosting

1. Choose "Semihosting\_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Start the project by choosing Project>Download and Debug.
4. Choose View>Terminal I/O to display the output from the I/O operations.

### 42.4.2 Guide Semihosting for Keil µVision

**NOTE:** Keil supports Semihosting only for Cortex-M3/Cortex-M4 cores.

#### Step 1: Prepare code

Remove function fputc and fgetc is used to support KEIL in "fsl\_debug\_console.c" and add the following code to project.

```
#pragma import(__use_no_semihosting_swi)

volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY;           /* used for Debug Input */
```

```
struct __FILE
{
    int handle;
};

FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f)
{
    return (ITM_SendChar(ch));
}

int fgetc(FILE *f)
{ /* blocking */
    while (ITM_CheckChar() != 1)
        ;
    return (ITM_ReceiveChar());
}

int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}

void _ttywrch(int ch)
{
    ITM_SendChar(ch);
}

void _sys_exit(int return_code)
{
label:
    goto label; /* endless loop */
}
```

### Step 2: Setting up the environment

1. In menu bar, choose Project>Options for target or using Alt+F7 or click.
2. Select "Target" tab and not select "Use MicroLIB".
3. Select "Debug" tab, select "J-Link/J-Trace Cortex" and click "Setting button".
4. Select "Debug" tab and choose Port:SW, then select "Trace" tab, choose "Enable" and click OK.

### Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

### Step 4: Building the project

1. Choose "Debug" on menu bar or Ctrl F5.
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer".
3. Run line by line to see result in Console Window.

### 42.4.3 Guide Semihosting for KDS

**NOTE:** After the setting use "printf" for debugging.

#### Step 1: Setting up the environment

1. In menu bar, choose Project>Properties>C/C++ Build>Settings>Tool Settings.
2. Select “Libraries” on “Cross ARM C Linker” and delete “nosys”.
3. Select “Miscellaneous” on “Cross ARM C Linker”, add “-specs=rdimon.specs” to “Other link flags” and tick “Use newlib-nano”, and click OK.

#### Step 2: Building the project

1. In menu bar, choose Project>Build Project.

#### Step 3: Starting semihosting

1. In Debug configurations, choose "Startup" tab, tick “Enable semihosting and Telnet”. Press “Apply” and “Debug”.
2. After clicking Debug, the Window is displayed same as below. Run line by line to see the result in the Console Window.

### 42.4.4 Guide Semihosting for ATL

**NOTE:** J-Link has to be used to enable semihosting.

#### Step 1: Prepare code

Add the following code to the project.

```
int _write(int file, char *ptr, int len)
{
    /* Implement your write code here. This is used by puts and printf. */
    int i=0;
    for(i=0 ; i<len ; i++)
        ITM_SendChar((*ptr++));
    return len;
}
```

#### Step 2: Setting up the environment

1. In menu bar, choose Debug Configurations. In tab "Embedded C/C++ Application" choose "- Semihosting\_ATL\_xxx debug J-Link".
2. In tab "Debugger" set up as follows.
  - JTAG mode must be selected

## Semihosting

- SWV tracing must be enabled
  - Enter the Core Clock frequency, which is hardware board-specific.
  - Enter the desired SWO Clock frequency. The latter depends on the JTAG Probe and must be a multiple of the Core Clock value.
3. Click "Apply" and "Debug".

### Step 3: Starting semihosting

1. In the Views menu, expand the submenu SWV and open the docking view "SWV Console". 2. Open the SWV settings panel by clicking the "Configure Serial Wire Viewer" button in the SWV Console view toolbar. 3. Configure the data ports to be traced by enabling the ITM channel 0 check-box in the ITM stimulus ports group: Choose "EXETRC: Trace Exceptions" and In tab "ITM Stimulus Ports" choose "Enable Port" 0. Then click "OK".
2. It is recommended not to enable other SWV trace functionalities at the same time because this may over use the SWO pin causing packet loss due to a limited bandwidth (certain other SWV tracing capabilities can send a lot of data at very high-speed). Save the SWV configuration by clicking the OK button. The configuration is saved with other debug configurations and remains effective until changed.
3. Press the red Start/Stop Trace button to send the SWV configuration to the target board to enable SWV trace recording. The board does not send any SWV packages until it is properly configured. The SWV Configuration must be present, if the configuration registers on the target board are reset. Also, tracing does not start until the target starts to execute.
4. Start the target execution again by pressing the green Resume Debug button.
5. The SWV console now shows the printf() output.

### 42.4.5 Guide Semihosting for ARMGCC

#### Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
  - "Host Name (or IP address)" : localhost
  - "Port" :2333
  - "Connection type" : Telnet.
  - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

#### Add to "CMakeLists.txt"

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")  
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")  
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --
```

```
defsym=__heap_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}
--defsym=__heap_size__=0x2000")
```

## Step 2: Building the project

1. Change "CMakeLists.txt":

**Change** "SET(CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE "\${CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE} -specs=nano.specs")"  
**to** "SET(CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE "\${CMAKE\_EXE\_LINKER\_FLAGS\_RELEASE} -specs=rdimon.specs")"

### Replace paragraph

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-common")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffunction-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fdata-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffreestanding")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-builtin")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mthumb")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mapcs")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
--gc-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-static")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-z")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
muldefs")
```

### To

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
```

## Semihosting

```
G} --specs=rdimon.specs ")  
Remove  
target_link_libraries(semihosting_ARMGCC.elf debug nosys)  
2. Run "build_debug.bat" to build project
```

### Step 3: Starting semihosting

- (a) Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug  
d:  
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe  
target remote localhost:2331  
monitor reset  
monitor semihosting enable  
monitor semihosting thumbSWI 0xAB  
monitor semihosting IOClient 1  
monitor flash device = MK64FN1M0xxx12  
load semihosting_ARMGCC.elf  
monitor reg pc = (0x00000004)  
monitor reg sp = (0x00000000)  
continue
```

- (b) After the setting, press "enter". The PuTTY window now shows the printf() output.

# Chapter 43

## Notification Framework

### 43.1 Overview

This section describes the programming interface of the Notifier driver.

### 43.2 Notifier Overview

The Notifier provides a configuration dynamic change service. Based on this service, applications can switch between pre-defined configurations. The Notifier enables drivers and applications to register callback functions to this framework. Each time that the configuration is changed, drivers and applications receive a notification and change their settings. To simplify, the Notifier only supports the static callback registration. This means that, for applications, all callback functions are collected into a static table and passed to the Notifier.

These are the steps for the configuration transition.

1. Before configuration transition, the Notifier sends a "BEFORE" message to the callback table. When this message is received, IP drivers should check whether any current processes can be stopped and stop them. If the processes cannot be stopped, the callback function returns an error.  
The Notifier supports two types of transition policies, a graceful policy and a forceful policy. When the graceful policy is used, if some callbacks return an error while sending a "BEFORE" message, the configuration transition stops and the Notifier sends a "RECOVER" message to all drivers that have stopped. Then, these drivers can recover the previous status and continue to work. When the forceful policy is used, drivers are stopped forcefully.
2. After the "BEFORE" message is processed successfully, the system switches to the new configuration.
3. After the configuration changes, the Notifier sends an "AFTER" message to the callback table to notify drivers that the configuration transition is finished.

This example shows how to use the Notifier in the Power Manager application.

```
#include "fsl_notifier.h"

// Definition of the Power Manager callback.
status_t callback0(notifier_notification_block_t *notify, void *data)
{
    status_t ret = kStatus_Success;

    ...
    ...

    return ret;
}
// Definition of the Power Manager user function.
status_t APP_PowerModeSwitch(notifier_user_config_t *targetConfig, void *userData)
```

## Notifier Overview

```
...
...
...
}
...
...
...
...
...
...
// Main function.
int main(void)
{
    // Define a notifier handle.
    notifier_handle_t powerModeHandle;

    // Callback configuration.
    user_callback_data_t callbackData0;

    notifier_callback_config_t callbackCfg0 = {callback0,
                                              kNOTIFIER_CallbackBeforeAfter,
                                              (void *)&callbackData0};

    notifier_callback_config_t callbacks[] = {callbackCfg0};

    // Power mode configurations.
    power_user_config_t vlprConfig;
    power_user_config_t stopConfig;

    notifier_user_config_t *powerConfigs[] = {&vlprConfig, &stopConfig};

    // Definition of a transition to and out the power modes.
    vlprConfig.mode = kAPP_PowerModeVlpr;
    vlprConfig.enableLowPowerWakeUpOnInterrupt = false;

    stopConfig = vlprConfig;
    stopConfig.mode = kAPP_PowerModeStop;

    // Create Notifier handle.
    NOTIFIER_CreateHandle(&powerModeHandle, powerConfigs, 2U, callbacks, 1U,
                          APP_PowerModeSwitch, NULL);
    ...
    ...
    // Power mode switch.
    NOTIFIER_switchConfig(&powerModeHandle, targetConfigIndex,
                          kNOTIFIER_PolicyAgreement);
}
```

## Data Structures

- struct **notifier\_notification\_block\_t**  
*notification block passed to the registered callback function.* [More...](#)
- struct **notifier\_callback\_config\_t**  
*Callback configuration structure.* [More...](#)
- struct **notifier\_handle\_t**  
*Notifier handle structure.* [More...](#)

## Typedefs

- **typedef void notifier\_user\_config\_t**  
*Notifier user configuration type.*
- **typedef status\_t(\* notifier\_user\_function\_t )(notifier\_user\_config\_t \*targetConfig, void \*userData)**  
*Notifier user function prototype Use this function to execute specific operations in configuration switch.*

- `typedef status_t(* notifier_callback_t )(notifier_notification_block_t *notify, void *data)`  
*Callback prototype.*

## Enumerations

- `enum _notifier_status {`  
 `kStatus_NOTIFIER_ErrorNotificationBefore,`  
 `kStatus_NOTIFIER_ErrorNotificationAfter }`  
*Notifier error codes.*
- `enum notifier_policy_t {`  
 `kNOTIFIER_PolicyAgreement,`  
 `kNOTIFIER_PolicyForcible }`  
*Notifier policies.*
- `enum notifier_notification_type_t {`  
 `kNOTIFIER_NotifyRecover = 0x00U,`  
 `kNOTIFIER_NotifyBefore = 0x01U,`  
 `kNOTIFIER_NotifyAfter = 0x02U }`  
*Notification type.*
- `enum notifier_callback_type_t {`  
 `kNOTIFIER_CallbackBefore = 0x01U,`  
 `kNOTIFIER_CallbackAfter = 0x02U,`  
 `kNOTIFIER_CallbackBeforeAfter = 0x03U }`  
*The callback type, which indicates kinds of notification the callback handles.*

## Functions

- `status_t NOTIFIER_CreateHandle (notifier_handle_t *notifierHandle, notifier_user_config_t **configs, uint8_t configsNumber, notifier_callback_config_t *callbacks, uint8_t callbacksNumber, notifier_user_function_t userFunction, void *userData)`  
*Creates a Notifier handle.*
- `status_t NOTIFIER_SwitchConfig (notifier_handle_t *notifierHandle, uint8_t configIndex, notifier_policy_t policy)`  
*Switches the configuration according to a pre-defined structure.*
- `uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t *notifierHandle)`  
*This function returns the last failed notification callback.*

### 43.3 Data Structure Documentation

#### 43.3.1 struct notifier\_notification\_block\_t

##### Data Fields

- `notifier_user_config_t * targetConfig`  
*Pointer to target configuration.*
- `notifier_policy_t policy`  
*Configure transition policy.*
- `notifier_notification_type_t notifyType`  
*Configure notification type.*

## Data Structure Documentation

### 43.3.1.0.0.86 Field Documentation

43.3.1.0.0.86.1 `notifier_user_config_t* notifier_notification_block_t::targetConfig`

43.3.1.0.0.86.2 `notifier_policy_t notifier_notification_block_t::policy`

43.3.1.0.0.86.3 `notifier_notification_type_t notifier_notification_block_t::notifyType`

### 43.3.2 struct notifier\_callback\_config\_t

This structure holds the configuration of callbacks. Callbacks of this type are expected to be statically allocated. This structure contains the following application-defined data. callback - pointer to the callback function callbackType - specifies when the callback is called callbackData - pointer to the data passed to the callback.

## Data Fields

- `notifier_callback_t callback`  
*Pointer to the callback function.*
- `notifier_callback_type_t callbackType`  
*Callback type.*
- `void * callbackData`  
*Pointer to the data passed to the callback.*

### 43.3.2.0.0.87 Field Documentation

43.3.2.0.0.87.1 `notifier_callback_t notifier_callback_config_t::callback`

43.3.2.0.0.87.2 `notifier_callback_type_t notifier_callback_config_t::callbackType`

43.3.2.0.0.87.3 `void* notifier_callback_config_t::callbackData`

### 43.3.3 struct notifier\_handle\_t

Notifier handle structure. Contains data necessary for the Notifier proper function. Stores references to registered configurations, callbacks, information about their numbers, user function, user data, and other internal data. `NOTIFIER_CreateHandle()` must be called to initialize this handle.

## Data Fields

- `notifier_user_config_t ** configsTable`  
*Pointer to configure table.*
- `uint8_t configsNumber`  
*Number of configurations.*
- `notifier_callback_config_t * callbacksTable`  
*Pointer to callback table.*

- `uint8_t callbacksNumber`  
*Maximum number of callback configurations.*
- `uint8_t errorCallbackIndex`  
*Index of callback returns error.*
- `uint8_t currentConfigIndex`  
*Index of current configuration.*
- `notifier_user_function_t userFunction`  
*User function.*
- `void *userData`  
*User data passed to user function.*

#### 43.3.3.0.0.88 Field Documentation

43.3.3.0.0.88.1 `notifier_user_config_t** notifier_handle_t::configsTable`

43.3.3.0.0.88.2 `uint8_t notifier_handle_t::configsNumber`

43.3.3.0.0.88.3 `notifier_callback_config_t* notifier_handle_t::callbacksTable`

43.3.3.0.0.88.4 `uint8_t notifier_handle_t::callbacksNumber`

43.3.3.0.0.88.5 `uint8_t notifier_handle_t::errorCallbackIndex`

43.3.3.0.0.88.6 `uint8_t notifier_handle_t::currentConfigIndex`

43.3.3.0.0.88.7 `notifier_user_function_t notifier_handle_t::userFunction`

43.3.3.0.0.88.8 `void* notifier_handle_t::userData`

### 43.4 Typedef Documentation

#### 43.4.1 `typedef void notifier_user_config_t`

Reference of the user defined configuration is stored in an array; the notifier switches between these configurations based on this array.

#### 43.4.2 `typedef status_t(* notifier_user_function_t)(notifier_user_config_t *targetConfig, void *userData)`

Before and after this function execution, different notification is sent to registered callbacks. If this function returns any error code, `NOTIFIER_SwitchConfig()` exits.

Parameters

---

## Enumeration Type Documentation

<i>targetConfig</i>	target Configuration.
<i>userData</i>	Refers to other specific data passed to user function.

Returns

An error code or kStatus\_Success.

### 43.4.3 **typedef status\_t(\* notifier\_callback\_t)(notifier\_notification\_block\_t \*notify, void \*data)**

Declaration of a callback. It is common for registered callbacks. Reference to function of this type is part of the [notifier\\_callback\\_config\\_t](#) callback configuration structure. Depending on callback type, function of this prototype is called (see [NOTIFIER\\_SwitchConfig\(\)](#)) before configuration switch, after it or in both use cases to notify about the switch progress (see [notifier\\_callback\\_type\\_t](#)). When called, the type of the notification is passed as a parameter along with the reference to the target configuration structure (see [notifier\\_notification\\_block\\_t](#)) and any data passed during the callback registration. When notified before the configuration switch, depending on the configuration switch policy (see [notifier\\_policy\\_t](#)), the callback may deny the execution of the user function by returning an error code different than kStatus\_Success (see [NOTIFIER\\_SwitchConfig\(\)](#)).

Parameters

<i>notify</i>	Notification block.
<i>data</i>	Callback data. Refers to the data passed during callback registration. Intended to pass any driver or application data such as internal state information.

Returns

An error code or kStatus\_Success.

## 43.5 Enumeration Type Documentation

### 43.5.1 enum \_notifier\_status

Used as return value of Notifier functions.

Enumerator

***kStatus\_NOTIFIER\_ErrorNotificationBefore*** An error occurs during send "BEFORE" notification.

***kStatus\_NOTIFIER\_ErrorNotificationAfter*** An error occurs during send "AFTER" notification.

### 43.5.2 enum notifier\_policy\_t

Defines whether the user function execution is forced or not. For `kNOTIFIER_PolicyForcible`, the user function is executed regardless of the callback results, while `kNOTIFIER_PolicyAgreement` policy is used to exit `NOTIFIER_SwitchConfig()` when any of the callbacks returns error code. See also `NOTIFIER_SwitchConfig()` description.

Enumerator

***kNOTIFIER\_PolicyAgreement*** `NOTIFIER_SwitchConfig()` method is exited when any of the callbacks returns error code.

***kNOTIFIER\_PolicyForcible*** The user function is executed regardless of the results.

### 43.5.3 enum notifier\_notification\_type\_t

Used to notify registered callbacks

Enumerator

***kNOTIFIER\_NotifyRecover*** Notify IP to recover to previous work state.

***kNOTIFIER\_NotifyBefore*** Notify IP that configuration setting is going to change.

***kNOTIFIER\_NotifyAfter*** Notify IP that configuration setting has been changed.

### 43.5.4 enum notifier\_callback\_type\_t

Used in the callback configuration structure (`notifier_callback_config_t`) to specify when the registered callback is called during configuration switch initiated by the `NOTIFIER_SwitchConfig()`. Callback can be invoked in following situations.

- Before the configuration switch (Callback return value can affect `NOTIFIER_SwitchConfig()` execution. See the `NOTIFIER_SwitchConfig()` and `notifier_policy_t` documentation).
- After an unsuccessful attempt to switch configuration
- After a successful configuration switch

Enumerator

***kNOTIFIER\_CallbackBefore*** Callback handles BEFORE notification.

***kNOTIFIER\_CallbackAfter*** Callback handles AFTER notification.

***kNOTIFIER\_CallbackBeforeAfter*** Callback handles BEFORE and AFTER notification.

## Function Documentation

### 43.6 Function Documentation

43.6.1 `status_t NOTIFIER_CreateHandle ( notifier_handle_t * notifierHandle,  
notifier_user_config_t ** configs, uint8_t configsNumber, notifier_callback-  
_config_t * callbacks, uint8_t callbacksNumber, notifier_user_function_t  
userFunction, void * userData )`

## Parameters

<i>notifierHandle</i>	A pointer to the notifier handle.
<i>configs</i>	A pointer to an array with references to all configurations which is handled by the Notifier.
<i>configsNumber</i>	Number of configurations. Size of the configuration array.
<i>callbacks</i>	A pointer to an array of callback configurations. If there are no callbacks to register during Notifier initialization, use NULL value.
<i>callbacks-Number</i>	Number of registered callbacks. Size of the callbacks array.
<i>userFunction</i>	User function.
<i>userData</i>	User data passed to user function.

## Returns

An error Code or kStatus\_Success.

### 43.6.2 **status\_t NOTIFIER\_SwitchConfig ( notifier\_handle\_t \* *notifierHandle*, uint8\_t *configIndex*, notifier\_policy\_t *policy* )**

This function sets the system to the target configuration. Before transition, the Notifier sends notifications to all callbacks registered to the callback table. Callbacks are invoked in the following order: All registered callbacks are notified ordered by index in the callbacks array. The same order is used for before and after switch notifications. The notifications before the configuration switch can be used to obtain confirmation about the change from registered callbacks. If any registered callback denies the configuration change, further execution of this function depends on the notifier policy: the configuration change is either forced (kNOTIFIER\_PolicyForcible) or exited (kNOTIFIER\_PolicyAgreement). When configuration change is forced, the result of the before switch notifications are ignored. If an agreement is required, if any callback returns an error code, further notifications before switch notifications are cancelled and all already notified callbacks are re-invoked. The index of the callback which returned error code during pre-switch notifications is stored (any error codes during callbacks re-invocation are ignored) and NOTIFIER\_GetErrorCallback() can be used to get it. Regardless of the policies, if any callback returns an error code, an error code indicating in which phase the error occurred is returned when NOTIFIER\_SwitchConfig() exits.

## Parameters

## Function Documentation

<i>notifierHandle</i>	pointer to notifier handle
<i>configIndex</i>	Index of the target configuration.
<i>policy</i>	Transaction policy, kNOTIFIER_PolicyAgreement or kNOTIFIER_PolicyForcible.

Returns

An error code or kStatus\_Success.

### 43.6.3 **uint8\_t NOTIFIER\_GetErrorCallbackIndex ( notifier\_handle\_t \* *notifierHandle* )**

This function returns an index of the last callback that failed during the configuration switch while the last [NOTIFIER\\_SwitchConfig\(\)](#) was called. If the last [NOTIFIER\\_SwitchConfig\(\)](#) call ended successfully value equal to callbacks number is returned. The returned value represents an index in the array of static call-backs.

Parameters

<i>notifierHandle</i>	Pointer to the notifier handle
-----------------------	--------------------------------

Returns

Callback Index of the last failed callback or value equal to callbacks count.

# Chapter 44

## Shell

### 44.1 Overview

This part describes the programming interface of the Shell middleware. Shell controls MCUs by commands via the specified communication peripheral based on the debug console driver.

### 44.2 Function groups

#### 44.2.1 Initialization

To initialize the Shell middleware, call the `SHELL_Init()` function with these parameters. This function automatically enables the middleware.

```
void SHELL_Init(p_shell_context_t context, send_data_cb_t send_cb,  
                recv_data_cb_t recv_cb, char *prompt);
```

Then, after the initialization was successful, call a command to control MCUs.

This example shows how to call the `SHELL_Init()` given the user configuration structure.

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
```

#### 44.2.2 Advanced Feature

- Support to get a character from standard input devices.

```
static uint8_t GetChar(p_shell_context_t context);
```

Commands	Description
Help	Lists all commands which are supported by Shell.
Exit	Exits the Shell program.
strCompare	Compares the two input strings.

Input character	Description
A	Gets the latest command in the history.
B	Gets the first command in the history.
C	Replaces one character at the right of the pointer.

## Function groups

Input character	Description
D	Replaces one character at the left of the pointer.
	Run AutoComplete function
	Run cmdProcess function
	Clears a command.

### 44.2.3 Shell Operation

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
SHELL_Main(&user_context);
```

## Data Structures

- struct [p\\_shell\\_context\\_t](#)  
*Data structure for Shell environment.* [More...](#)
- struct [shell\\_command\\_context\\_t](#)  
*User command data structure.* [More...](#)
- struct [shell\\_command\\_context\\_list\\_t](#)  
*Structure list command.* [More...](#)

## Macros

- #define [SHELL\\_USE\\_HISTORY](#) (0U)  
*Macro to set on/off history feature.*
- #define [SHELL\\_SEARCH\\_IN\\_HIST](#) (1U)  
*Macro to set on/off history feature.*
- #define [SHELL\\_USE\\_FILE\\_STREAM](#) (0U)  
*Macro to select method stream.*
- #define [SHELL\\_AUTO\\_COMPLETE](#) (1U)  
*Macro to set on/off auto-complete feature.*
- #define [SHELL\\_BUFFER\\_SIZE](#) (64U)  
*Macro to set console buffer size.*
- #define [SHELL\\_MAX\\_ARGS](#) (8U)  
*Macro to set maximum arguments in command.*
- #define [SHELL\\_HIST\\_MAX](#) (3U)  
*Macro to set maximum count of history commands.*
- #define [SHELL\\_MAX\\_CMD](#) (20U)  
*Macro to set maximum count of commands.*
- #define [SHELL\\_OPTIONAL\\_PARAMS](#) (0xFF)  
*Macro to bypass arguments check.*

## Typedefs

- typedef void(\* [send\\_data\\_cb\\_t](#) )(uint8\_t \*buf, uint32\_t len)  
*Shell user send data callback prototype.*
- typedef void(\* [recv\\_data\\_cb\\_t](#) )(uint8\_t \*buf, uint32\_t len)

- *Shell user receiver data callback prototype.*  
typedef int(\* **printf\_data\_t** )(const char \*format,...)
- *Shell user printf data prototype.*  
typedef int32\_t(\* **cmd\_function\_t** )(p\_shell\_context\_t context, int32\_t argc, char \*\*argv)
- *User command function prototype.*

## Enumerations

- enum **fun\_key\_status\_t** {
   
kSHELL\_Normal = 0U,  
kSHELL\_Special = 1U,  
kSHELL\_Function = 2U }

*A type for the handle special key.*

## Shell functional operation

- void **SHELL\_Init** (p\_shell\_context\_t context, **send\_data\_cb\_t** send\_cb, **recv\_data\_cb\_t** recv\_cb, **printf\_data\_t** shell\_printf, char \*prompt)
   
*Enables the clock gate and configures the Shell module according to the configuration structure.*
- int32\_t **SHELL\_RegisterCommand** (const **shell\_command\_context\_t** \*command\_context)
   
*Shell register command.*
- int32\_t **SHELL\_Main** (p\_shell\_context\_t context)
   
*Main loop for Shell.*

## 44.3 Data Structure Documentation

### 44.3.1 struct shell\_context\_struct

#### Data Fields

- char \* **prompt**
  
*Prompt string.*
- enum **\_fun\_key\_status** **stat**
  
*Special key status.*
- char **line** [**SHELL\_BUFFER\_SIZE**]
   
*Consult buffer.*
- uint8\_t **cmd\_num**
  
*Number of user commands.*
- uint8\_t **l\_pos**
  
*Total line position.*
- uint8\_t **c\_pos**
  
*Current line position.*
- **send\_data\_cb\_t** **send\_data\_func**
  
*Send data interface operation.*
- **recv\_data\_cb\_t** **recv\_data\_func**
  
*Receive data interface operation.*
- uint16\_t **hist\_current**
  
*Current history command in hist buff.*
- uint16\_t **hist\_count**

## Data Structure Documentation

- char **hist\_buf** [SHELL\_HIST\_MAX][SHELL\_BUFFER\_SIZE]  
*Total history command in hist buff.*
- History buffer.
- bool **exit**  
*Exit Flag.*

### 44.3.2 struct shell\_command\_context\_t

#### Data Fields

- const char \* **pcCommand**  
*The command that is executed.*
- char \* **pcHelpString**  
*String that describes how to use the command.*
- const cmd\_function\_t **pFuncCallBack**  
*A pointer to the callback function that returns the output generated by the command.*
- uint8\_t **cExpectedNumberOfParameters**  
*Commands expect a fixed number of parameters, which may be zero.*

#### 44.3.2.0.0.89 Field Documentation

##### 44.3.2.0.0.89.1 const char\* shell\_command\_context\_t::pcCommand

For example "help". It must be all lower case.

##### 44.3.2.0.0.89.2 char\* shell\_command\_context\_t::pcHelpString

It should start with the command itself, and end with "\r\n". For example "help: Returns a list of all the commands\r\n".

##### 44.3.2.0.0.89.3 const cmd\_function\_t shell\_command\_context\_t::pFuncCallBack

##### 44.3.2.0.0.89.4 uint8\_t shell\_command\_context\_t::cExpectedNumberOfParameters

### 44.3.3 struct shell\_command\_context\_list\_t

#### Data Fields

- const shell\_command\_context\_t \* **CommandList** [SHELL\_MAX\_CMD]  
*The command table list.*
- uint8\_t **numberOfCommandInList**  
*The total command in list.*

## 44.4 Macro Definition Documentation

44.4.1 `#define SHELL_USE_HISTORY (0U)`

44.4.2 `#define SHELL_SEARCH_IN_HIST (1U)`

44.4.3 `#define SHELL_USE_FILE_STREAM (0U)`

44.4.4 `#define SHELL_AUTO_COMPLETE (1U)`

44.4.5 `#define SHELL_BUFFER_SIZE (64U)`

44.4.6 `#define SHELL_MAX_ARGS (8U)`

44.4.7 `#define SHELL_HIST_MAX (3U)`

44.4.8 `#define SHELL_MAX_CMD (20U)`

## 44.5 Typedef Documentation

44.5.1 `typedef void(* send_data_cb_t)(uint8_t *buf, uint32_t len)`

44.5.2 `typedef void(* recv_data_cb_t)(uint8_t *buf, uint32_t len)`

44.5.3 `typedef int(* printf_data_t)(const char *format,...)`

44.5.4 `typedef int32_t(* cmd_function_t)(p_shell_context_t context, int32_t argc, char **argv)`

## 44.6 Enumeration Type Documentation

44.6.1 `enum fun_key_status_t`

Enumerator

*kSHELL\_Normal* Normal key.

*kSHELL\_Special* Special key.

*kSHELL\_Function* Function key.

## Function Documentation

### 44.7 Function Documentation

#### 44.7.1 void SHELL\_Init ( *p\_shell\_context\_t context*, *send\_data\_cb\_t send\_cb*, *recv\_data\_cb\_t recv\_cb*, *printf\_data\_t shell\_printf*, *char \* prompt* )

This function must be called before calling all other Shell functions. Call operation the Shell commands with user-defined settings. The example below shows how to set up the middleware Shell and how to call the SHELL\_Init function by passing in these parameters. This is an example.

```
*     shell_context_struct user_context;
*     SHELL_Init(&user_context, SendDataFunc, ReceiveDataFunc, "SHELL>> ");
*
```

#### Parameters

<i>context</i>	The pointer to the Shell environment and runtime states.
<i>send_cb</i>	The pointer to call back send data function.
<i>recv_cb</i>	The pointer to call back receive data function.
<i>prompt</i>	The string prompt of Shell

#### 44.7.2 int32\_t SHELL\_RegisterCommand ( *const shell\_command\_context\_t \* command\_context* )

#### Parameters

<i>command_context</i>	The pointer to the command data structure.
------------------------	--

#### Returns

-1 if error or 0 if success

#### 44.7.3 int32\_t SHELL\_Main ( *p\_shell\_context\_t context* )

Main loop for Shell; After this function is called, Shell begins to initialize the basic variables and starts to work.

### Parameters

<i>context</i>	The pointer to the Shell environment and runtime states.
----------------	--

### Returns

This function does not return until Shell command exit was called.

## Function Documentation

# Chapter 45

## Smartcard\_phy\_ncn8025\_driver

### 45.1 Overview

#### Macros

- `#define SMARTCARD_ATR_DURATION_ADJUSTMENT (360u)`  
*Smart card definition which specifies the adjustment number of clock cycles during which an ATR string has to be received.*
- `#define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)`  
*Smart card definition which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.*
- `#define SMARTCARD_NCN8025_STATUS_PRES (0x01u)`  
*Masks for NCN8025 status register.*
- `#define SMARTCARD_NCN8025_STATUS_ACTIVE (0x02u)`  
*Smart card phy NCN8025 Smart card active status.*
- `#define SMARTCARD_NCN8025_STATUS_FAULTY (0x04u)`  
*Smart card phy NCN8025 Smart card faulty status.*
- `#define SMARTCARD_NCN8025_STATUS_CARD_REMOVED (0x08u)`  
*Smart card phy NCN8025 Smart card removed status.*
- `#define SMARTCARD_NCN8025_STATUS_CARD_DEACTIVATED (0x10u)`  
*Smart card phy NCN8025 Smart card deactivated status.*

#### Functions

- `void SMARTCARD_PHY_NCN8025_GetDefaultConfig (smartcard_interface_config_t *config)`  
*Fills in the configuration structure with default values.*
- `status_t SMARTCARD_PHY_NCN8025_Init (void *base, smartcard_interface_config_t const *config, uint32_t srcClock_Hz)`  
*Initializes a Smart card interface instance for operation.*
- `void SMARTCARD_PHY_NCN8025_Deinit (void *base, smartcard_interface_config_t *config)`  
*De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.*
- `status_t SMARTCARD_PHY_NCN8025_Activate (void *base, smartcard_context_t *context, smartcard_reset_type_t resetType)`  
*Activates the Smart card IC.*
- `status_t SMARTCARD_PHY_NCN8025_Deactivate (void *base, smartcard_context_t *context)`  
*De-activates the Smart card IC.*
- `status_t SMARTCARD_PHY_NCN8025_Control (void *base, smartcard_context_t *context, smartcard_interface_control_t control, uint32_t param)`  
*Controls the Smart card interface IC.*
- `void SMARTCARD_PHY_NCN8025_IRQHandler (void *base, smartcard_context_t *context)`  
*Smart card interface IC IRQ ISR.*

## Function Documentation

### 45.2 Macro Definition Documentation

45.2.1 `#define SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT (4200u)`

45.2.2 `#define SMARTCARD_NCN8025_STATUS_PRES (0x01u)`

Smart card phy NCN8025 Smart card present status

### 45.3 Function Documentation

45.3.1 `void SMARTCARD_PHY_NCN8025_GetDefaultConfig ( smartcard_interface_config_t * config )`

Parameters

<i>config</i>	The Smart card user configuration structure which contains configuration structure of type <code>smartcard_interface_config_t</code> . Function fill in members: <code>clockToResetDelay</code> = 42000, <code>vcc</code> = <code>kSmartcardVoltageClassB3_3V</code> , with default values.
---------------	---

45.3.2 `status_t SMARTCARD_PHY_NCN8025_Init ( void * base, smartcard_interface_config_t const * config, uint32_t srcClock_Hz )`

Parameters

<i>base</i>	The Smart card peripheral base address.
<i>config</i>	The user configuration structure of type <code>smartcard_interface_config_t</code> . Call the function <code>SMARTCARD_PHY_NCN8025_GetDefaultConfig()</code> to fill out the configuration structure.
<i>srcClock_Hz</i>	Smart card clock generation module source clock.

Return values

<i>kStatus_SMARTCARD_Success</i>	or <code>kStatus_SMARTCARD_OtherError</code> in case of error.
----------------------------------	--

45.3.3 `void SMARTCARD_PHY_NCN8025_Deinit ( void * base, smartcard_interface_config_t * config )`

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>config</i>	The user configuration structure of type <a href="#">smartcard_interface_config_t</a> .

#### 45.3.4 **status\_t SMARTCARD\_PHY\_NCN8025\_Activate ( void \* *base*, smartcard\_context\_t \* *context*, smartcard\_reset\_type\_t *resetType* )**

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>resetType</i>	type of reset to be performed, possible values = kSmartcardColdReset, kSmartcard-WarmReset

Return values

<i>kStatus_SMARTCARD_-Success</i>	or kStatus_SMARTCARD_OtherError in case of error.
-----------------------------------	---

#### 45.3.5 **status\_t SMARTCARD\_PHY\_NCN8025\_Deactivate ( void \* *base*, smartcard\_context\_t \* *context* )**

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.

Return values

<i>kStatus_SMARTCARD_-Success</i>	or kStatus_SMARTCARD_OtherError in case of error.
-----------------------------------	---

#### 45.3.6 **status\_t SMARTCARD\_PHY\_NCN8025\_Control ( void \* *base*, smartcard\_context\_t \* *context*, smartcard\_interface\_control\_t *control*, uint32\_t *param* )**

## Function Documentation

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	A pointer to a Smart card driver context structure.
<i>control</i>	A interface command type.
<i>param</i>	Integer value specific to control type

Return values

<i>kStatus_SMARTCARD_-Success</i>	or kStatus_SMARTCARD_OtherError in case of error.
-----------------------------------	---

### 45.3.7 void SMARTCARD\_PHY\_NCN8025\_IRQHandler ( void \* *base*, smartcard\_context\_t \* *context* )

Parameters

<i>base</i>	The Smart card peripheral module base address.
<i>context</i>	The Smart card context pointer.

**How to Reach Us:**

**Home Page:**

[nxp.com](http://nxp.com)

**Web Support:**

[nxp.com/support](http://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](http://nxp.com/SalesTermsandConditions).

NXP, the NXP logo, Freescale, the Freescale logo, Kinetis, Processor Expert are trademarks of NXP B.V. Tower is a trademark of NXP. All other product or service names are the property of their respective owners. ARM, ARM Powered logo, and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2017 NXP B.V.

Document Number: MCUXSDKK21FAPIRM  
Rev. 0  
Mar 2017

