
MCUXpresso SDK API Reference Manual

NXP Semiconductors

Document Number: MCUXSDKKL46APIRM
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](#)

5.1	Overview	11
5.2	Typical use case	11
5.2.1	Polling Configuration	11
5.2.2	Interrupt Configuration	11
5.3	Data Structure Documentation	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
5.4	Macro Definition Documentation	17
5.4.1	FSL_ADC16_DRIVER_VERSION	17
5.5	Enumeration Type Documentation	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
5.6	Function Documentation	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	
6.1	Overview	27
6.2	Typical use case	27
6.2.1	Polling Configuration	27
6.2.2	Interrupt Configuration	28
6.3	Data Structure Documentation	30
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
6.4	Macro Definition Documentation	32
6.4.1	FSL_CMP_DRIVER_VERSION	32
6.5	Enumeration Type Documentation	32
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
6.6	Function Documentation	33
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_SetDACConfig	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

Chapter COP: Watchdog Driver

7.1	Overview	39
7.2	Typical use case	39
7.3	Data Structure Documentation	40
7.3.1	struct cop_config_t	40
7.4	Macro Definition Documentation	40
7.4.1	FSL_COP_DRIVER_VERSION	40
7.5	Enumeration Type Documentation	40
7.5.1	cop_clock_source_t	40
7.5.2	cop_timeout_cycles_t	40
7.6	Function Documentation	41
7.6.1	COP_GetDefaultConfig	41
7.6.2	COP_Init	41
7.6.3	COP_Disable	41
7.6.4	COP_Refresh	43

Chapter DAC: Digital-to-Analog Converter Driver

8.1	Overview	45
8.2	Typical use case	45
8.2.1	Working as a basic DAC without the hardware buffer feature	45
8.2.2	Working with the hardware buffer	45
8.3	Data Structure Documentation	48
8.3.1	struct dac_config_t	48
8.3.2	struct dac_buffer_config_t	48
8.4	Macro Definition Documentation	49
8.4.1	FSL_DAC_DRIVER_VERSION	49
8.5	Enumeration Type Documentation	49

Contents

Section Number	Title	Page Number
8.5.1	_dac_buffer_status_flags	49
8.5.2	_dac_buffer_interrupt_enable	49
8.5.3	dac_reference_voltage_source_t	49
8.5.4	dac_buffer_trigger_mode_t	49
8.5.5	dac_buffer_work_mode_t	49
8.6	Function Documentation	50
8.6.1	DAC_Init	50
8.6.2	DAC_Deinit	50
8.6.3	DAC_GetDefaultConfig	50
8.6.4	DAC_Enable	50
8.6.5	DAC_EnableBuffer	51
8.6.6	DAC_SetBufferConfig	51
8.6.7	DAC_GetDefaultBufferConfig	51
8.6.8	DAC_EnableBufferDMA	51
8.6.9	DAC_SetBufferValue	52
8.6.10	DAC_DoSoftwareTriggerBuffer	52
8.6.11	DAC_GetBufferReadPointer	52
8.6.12	DAC_SetBufferReadPointer	53
8.6.13	DAC_EnableBufferInterrupts	53
8.6.14	DAC_DisableBufferInterrupts	53
8.6.15	DAC_GetBufferStatusFlags	53
8.6.16	DAC_ClearBufferStatusFlags	53
Chapter	DMA: Direct Memory Access Controller Driver	
9.1	Overview	55
9.2	Typical use case	55
9.2.1	DMA Operation	55
9.3	Data Structure Documentation	58
9.3.1	struct dma_transfer_config_t	58
9.3.2	struct dma_channel_link_config_t	59
9.3.3	struct dma_handle_t	59
9.4	Macro Definition Documentation	60
9.4.1	FSL_DMA_DRIVER_VERSION	60
9.5	Typedef Documentation	60
9.5.1	dma_callback	60
9.6	Enumeration Type Documentation	60
9.6.1	_dma_channel_status_flags	60
9.6.2	dma_transfer_size_t	60
9.6.3	dma_modulo_t	60

Contents

Section Number	Title	Page Number
9.6.4	dma_channel_link_type_t	61
9.6.5	dma_transfer_type_t	61
9.6.6	dma_transfer_options_t	61
9.7	Function Documentation	62
9.7.1	DMA_Init	62
9.7.2	DMA_Deinit	63
9.7.3	DMA_ResetChannel	63
9.7.4	DMA_SetTransferConfig	63
9.7.5	DMA_SetChannelLinkConfig	64
9.7.6	DMA_SetSourceAddress	64
9.7.7	DMA_SetDestinationAddress	64
9.7.8	DMA_SetTransferSize	65
9.7.9	DMA_SetModulo	65
9.7.10	DMA_EnableCycleSteal	65
9.7.11	DMA_EnableAutoAlign	66
9.7.12	DMA_EnableAsyncRequest	66
9.7.13	DMA_EnableInterrupts	66
9.7.14	DMA_DisableInterrupts	67
9.7.15	DMA_EnableChannelRequest	67
9.7.16	DMA_DisableChannelRequest	67
9.7.17	DMA_TriggerChannelStart	67
9.7.18	DMA_GetRemainingBytes	68
9.7.19	DMA_GetChannelStatusFlags	68
9.7.20	DMA_ClearChannelStatusFlags	68
9.7.21	DMA_CreateHandle	69
9.7.22	DMA_SetCallback	69
9.7.23	DMA_PrepareTransfer	69
9.7.24	DMA_SubmitTransfer	70
9.7.25	DMA_StartTransfer	70
9.7.26	DMA_StopTransfer	71
9.7.27	DMA_AbortTransfer	71
9.7.28	DMA_HandleIRQ	71
Chapter	DMAMUX: Direct Memory Access Multiplexer Driver	
10.1	Overview	73
10.2	Typical use case	73
10.2.1	DMAMUX Operation	73
10.3	Macro Definition Documentation	73
10.3.1	FSL_DMAMUX_DRIVER_VERSION	73
10.4	Function Documentation	74

Contents

Section Number	Title	Page Number
10.4.1	DMAMUX_Init	74
10.4.2	DMAMUX_Deinit	75
10.4.3	DMAMUX_EnableChannel	75
10.4.4	DMAMUX_DisableChannel	75
10.4.5	DMAMUX_SetSource	76
10.4.6	DMAMUX_EnablePeriodTrigger	76
10.4.7	DMAMUX_DisablePeriodTrigger	76
Chapter	C90TFS Flash Driver	
11.1	Overview	77
11.2	Data Structure Documentation	86
11.2.1	struct flash_execute_in_ram_function_config_t	86
11.2.2	struct flash_swap_state_config_t	86
11.2.3	struct flash_swap_ifr_field_config_t	86
11.2.4	union flash_swap_ifr_field_data_t	87
11.2.5	union pflash_protection_status_low_t	87
11.2.6	struct pflash_protection_status_t	88
11.2.7	struct flash_prefetch_speculation_status_t	88
11.2.8	struct flash_protection_config_t	88
11.2.9	struct flash_access_config_t	89
11.2.10	struct flash_operation_config_t	89
11.2.11	struct flash_config_t	90
11.3	Macro Definition Documentation	92
11.3.1	MAKE_VERSION	92
11.3.2	FSL_FLASH_DRIVER_VERSION	92
11.3.3	FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT	92
11.3.4	FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT	92
11.3.5	FLASH_DRIVER_IS_FLASH_RESIDENT	92
11.3.6	FLASH_DRIVER_IS_EXPORTED	92
11.3.7	kStatusGroupGeneric	92
11.3.8	MAKE_STATUS	92
11.3.9	FOUR_CHAR_CODE	92
11.4	Enumeration Type Documentation	92
11.4.1	_flash_driver_version_constants	92
11.4.2	_flash_status	93
11.4.3	_flash_driver_api_keys	93
11.4.4	flash_margin_value_t	94
11.4.5	flash_security_state_t	94
11.4.6	flash_protection_state_t	94
11.4.7	flash_execute_only_access_state_t	94
11.4.8	flash_property_tag_t	94

Contents

Section Number	Title	Page Number
11.4.9	_flash_execute_in_ram_function_constants	95
11.4.10	flash_read_resource_option_t	95
11.4.11	_flash_read_resource_range	95
11.4.12	_k3_flash_read_once_index	96
11.4.13	flash_flexram_function_option_t	96
11.4.14	flash_swap_function_option_t	96
11.4.15	flash_swap_control_option_t	96
11.4.16	flash_swap_state_t	97
11.4.17	flash_swap_block_status_t	97
11.4.18	flash_partition_flexram_load_option_t	97
11.4.19	flash_memory_index_t	97
11.4.20	flash_cache_controller_index_t	97
11.4.21	flash_cache_clear_process_t	98
11.5	Function Documentation	98
11.5.1	FLASH_Init	98
11.5.2	FLASH_SetCallback	98
11.5.3	FLASH_PrepareExecuteInRamFunctions	99
11.5.4	FLASH_EraseAll	99
11.5.5	FLASH_Erase	100
11.5.6	FLASH_EraseAllExecuteOnlySegments	101
11.5.7	FLASH_Program	102
11.5.8	FLASH_ProgramOnce	103
11.5.9	FLASH_ReadResource	104
11.5.10	FLASH_ReadOnce	105
11.5.11	FLASH_GetSecurityState	106
11.5.12	FLASH_SecurityBypass	106
11.5.13	FLASH_VerifyEraseAll	107
11.5.14	FLASH_VerifyErase	108
11.5.15	FLASH_VerifyProgram	109
11.5.16	FLASH_VerifyEraseAllExecuteOnlySegments	110
11.5.17	FLASH_IsProtected	111
11.5.18	FLASH_IsExecuteOnly	112
11.5.19	FLASH_GetProperty	113
11.5.20	FLASH_SetProperty	113
11.5.21	FLASH_PflashSetProtection	114
11.5.22	FLASH_PflashGetProtection	114
Chapter	GPIO: General-Purpose Input/Output Driver	
12.1	Overview	117
12.2	Data Structure Documentation	117
12.2.1	struct gpio_pin_config_t	117

Contents

Section Number	Title	Page Number
12.3	Macro Definition Documentation	118
12.3.1	FSL_GPIO_DRIVER_VERSION	118
12.4	Enumeration Type Documentation	118
12.4.1	gpio_pin_direction_t	118
12.5	GPIO Driver	119
12.5.1	Overview	119
12.5.2	Typical use case	119
12.5.3	Function Documentation	120
12.6	FGPIO Driver	123
12.6.1	Overview	123
12.6.2	Typical use case	123
12.6.3	Function Documentation	124
Chapter 12C: Inter-Integrated Circuit Driver		
13.1	Overview	129
13.2	I2C Driver	130
13.2.1	Overview	130
13.2.2	Typical use case	130
13.2.3	Data Structure Documentation	137
13.2.4	Macro Definition Documentation	142
13.2.5	Typedef Documentation	142
13.2.6	Enumeration Type Documentation	142
13.2.7	Function Documentation	144
13.3	I2C eDMA Driver	159
13.3.1	Overview	159
13.3.2	Data Structure Documentation	159
13.3.3	Typedef Documentation	160
13.3.4	Function Documentation	160
13.4	I2C DMA Driver	163
13.4.1	Overview	163
13.4.2	Data Structure Documentation	163
13.4.3	Typedef Documentation	164
13.4.4	Function Documentation	164
13.5	I2C FreeRTOS Driver	166
13.5.1	Overview	166
13.5.2	Function Documentation	166

Contents

Section Number	Title	Page Number
Chapter	LLWU: Low-Leakage Wakeup Unit Driver	
14.1	Overview	169
14.2	External wakeup pins configurations	169
14.3	Internal wakeup modules configurations	169
14.4	Digital pin filter for external wakeup pin configurations	169
14.5	Data Structure Documentation	170
14.5.1	struct llwu_external_pin_filter_mode_t	170
14.6	Macro Definition Documentation	170
14.6.1	FSL_LLWU_DRIVER_VERSION	170
14.7	Enumeration Type Documentation	170
14.7.1	llwu_external_pin_mode_t	170
14.7.2	llwu_pin_filter_mode_t	171
14.8	Function Documentation	171
14.8.1	LLWU_SetExternalWakeupPinMode	171
14.8.2	LLWU_GetExternalWakeupPinFlag	171
14.8.3	LLWU_ClearExternalWakeupPinFlag	172
14.8.4	LLWU_EnableInternalModuleInterruptWakup	173
14.8.5	LLWU_GetInternalWakeupModuleFlag	173
14.8.6	LLWU_SetPinFilterMode	173
14.8.7	LLWU_GetPinFilterFlag	174
14.8.8	LLWU_ClearPinFilterFlag	174
Chapter	LPSCI: Universal Asynchronous Receiver/Transmitter	
15.1	Overview	175
15.2	LPSCI Driver	176
15.2.1	Overview	176
15.2.2	Function groups	176
15.2.3	Typical use case	177
15.2.4	Data Structure Documentation	181
15.2.5	Macro Definition Documentation	182
15.2.6	Typedef Documentation	182
15.2.7	Enumeration Type Documentation	182
15.2.8	Function Documentation	184
15.3	LPSCI DMA Driver	197
15.3.1	Overview	197

Contents

Section Number	Title	Page Number
15.3.2	Data Structure Documentation	197
15.3.3	Typedef Documentation	198
15.3.4	Function Documentation	198
15.4	LPSCI FreeRTOS Driver	202
15.4.1	Overview	202
15.4.2	Data Structure Documentation	202
15.4.3	Function Documentation	203
Chapter	LPTMR: Low-Power Timer	
16.1	Overview	207
16.2	Function groups	207
16.2.1	Initialization and deinitialization	207
16.2.2	Timer period Operations	207
16.2.3	Start and Stop timer operations	207
16.2.4	Status	208
16.2.5	Interrupt	208
16.3	Typical use case	208
16.3.1	LPTMR tick example	208
16.4	Data Structure Documentation	210
16.4.1	struct lptmr_config_t	210
16.5	Enumeration Type Documentation	211
16.5.1	lptmr_pin_select_t	211
16.5.2	lptmr_pin_polarity_t	211
16.5.3	lptmr_timer_mode_t	211
16.5.4	lptmr_prescaler_glitch_value_t	212
16.5.5	lptmr_prescaler_clock_select_t	212
16.5.6	lptmr_interrupt_enable_t	212
16.5.7	lptmr_status_flags_t	213
16.6	Function Documentation	213
16.6.1	LPTMR_Init	213
16.6.2	LPTMR_Deinit	213
16.6.3	LPTMR_GetDefaultConfig	213
16.6.4	LPTMR_EnableInterrupts	214
16.6.5	LPTMR_DisableInterrupts	214
16.6.6	LPTMR_GetEnabledInterrupts	214
16.6.7	LPTMR_GetStatusFlags	214
16.6.8	LPTMR_ClearStatusFlags	215
16.6.9	LPTMR_SetTimerPeriod	215
16.6.10	LPTMR_GetCurrentTimerCount	215

Contents

Section Number	Title	Page Number
16.6.11	LPTMR_StartTimer	216
16.6.12	LPTMR_StopTimer	216
Chapter	PIT: Periodic Interrupt Timer	
17.1	Overview	217
17.2	Function groups	217
17.2.1	Initialization and deinitialization	217
17.2.2	Timer period Operations	217
17.2.3	Start and Stop timer operations	217
17.2.4	Status	218
17.2.5	Interrupt	218
17.3	Typical use case	218
17.3.1	PIT tick example	218
17.4	Data Structure Documentation	220
17.4.1	struct pit_config_t	220
17.5	Enumeration Type Documentation	220
17.5.1	pit_chnl_t	220
17.5.2	pit_interrupt_enable_t	221
17.5.3	pit_status_flags_t	221
17.6	Function Documentation	221
17.6.1	PIT_Init	221
17.6.2	PIT_Deinit	221
17.6.3	PIT_GetDefaultConfig	221
17.6.4	PIT_SetTimerChainMode	222
17.6.5	PIT_EnableInterrupts	222
17.6.6	PIT_DisableInterrupts	222
17.6.7	PIT_GetEnabledInterrupts	223
17.6.8	PIT_GetStatusFlags	223
17.6.9	PIT_ClearStatusFlags	223
17.6.10	PIT_SetTimerPeriod	224
17.6.11	PIT_GetCurrentTimerCount	224
17.6.12	PIT_StartTimer	225
17.6.13	PIT_StopTimer	225
17.6.14	PIT_GetLifetimeTimerCount	225
Chapter	PMC: Power Management Controller	
18.1	Overview	227
18.2	Data Structure Documentation	228

Contents

Section Number	Title	Page Number
18.2.1	struct pmc_low_volt_detect_config_t	228
18.2.2	struct pmc_low_volt_warning_config_t	228
18.2.3	struct pmc_bandgap_buffer_config_t	228
18.3	Macro Definition Documentation	229
18.3.1	FSL_PMC_DRIVER_VERSION	229
18.4	Enumeration Type Documentation	229
18.4.1	pmc_low_volt_detect_volt_select_t	229
18.4.2	pmc_low_volt_warning_volt_select_t	229
18.5	Function Documentation	229
18.5.1	PMC_ConfigureLowVoltDetect	229
18.5.2	PMC_GetLowVoltDetectFlag	230
18.5.3	PMC_ClearLowVoltDetectFlag	230
18.5.4	PMC_ConfigureLowVoltWarning	230
18.5.5	PMC_GetLowVoltWarningFlag	231
18.5.6	PMC_ClearLowVoltWarningFlag	231
18.5.7	PMC_ConfigureBandgapBuffer	231
18.5.8	PMC_GetPeriphIOIsolationFlag	232
18.5.9	PMC_ClearPeriphIOIsolationFlag	232
18.5.10	PMC_IsRegulatorInRunRegulation	232
Chapter	PORT: Port Control and Interrupts	
19.1	Overview	235
19.2	Typical configuration use case	235
19.2.1	Input PORT configuration	235
19.2.2	I2C PORT Configuration	235
19.3	Data Structure Documentation	237
19.3.1	struct port_pin_config_t	237
19.4	Macro Definition Documentation	237
19.4.1	FSL_PORT_DRIVER_VERSION	237
19.5	Enumeration Type Documentation	237
19.5.1	_port_pull	237
19.5.2	_port_slew_rate	238
19.5.3	_port_passive_filter_enable	238
19.5.4	_port_drive_strength	238
19.5.5	port_mux_t	238
19.5.6	port_interrupt_t	239
19.6	Function Documentation	239

Contents

Section Number	Title	Page Number
19.6.1	PORT_SetPinConfig	239
19.6.2	PORT_SetMultiplePinsConfig	239
19.6.3	PORT_SetPinMux	240
19.6.4	PORT_SetPinInterruptConfig	240
19.6.5	PORT_GetPinsInterruptFlags	241
19.6.6	PORT_ClearPinsInterruptFlags	242
Chapter	RCM: Reset Control Module Driver	
20.1	Overview	245
20.2	Data Structure Documentation	246
20.2.1	struct rcm_reset_pin_filter_config_t	246
20.3	Macro Definition Documentation	246
20.3.1	FSL_RCM_DRIVER_VERSION	246
20.4	Enumeration Type Documentation	246
20.4.1	rcm_reset_source_t	246
20.4.2	rcm_run_wait_filter_mode_t	247
20.5	Function Documentation	247
20.5.1	RCM_GetPreviousResetSources	247
20.5.2	RCM_ConfigureResetPinFilter	247
Chapter	RTC: Real Time Clock	
21.1	Overview	249
21.2	Function groups	249
21.2.1	Initialization and deinitialization	249
21.2.2	Set & Get Datetime	249
21.2.3	Set & Get Alarm	249
21.2.4	Start & Stop timer	250
21.2.5	Status	250
21.2.6	Interrupt	250
21.2.7	RTC Oscillator	250
21.2.8	Monotonic Counter	250
21.3	Typical use case	250
21.3.1	RTC tick example	250
21.4	Data Structure Documentation	253
21.4.1	struct rtc_datetime_t	253
21.4.2	struct rtc_config_t	254

Contents

Section Number	Title	Page Number
21.5	Enumeration Type Documentation	255
21.5.1	rtc_interrupt_enable_t	255
21.5.2	rtc_status_flags_t	255
21.5.3	rtc_osc_cap_load_t	255
21.6	Function Documentation	255
21.6.1	RTC_Init	255
21.6.2	RTC_Deinit	256
21.6.3	RTC_GetDefaultConfig	256
21.6.4	RTC_SetDatetime	256
21.6.5	RTC_GetDatetime	257
21.6.6	RTC_SetAlarm	257
21.6.7	RTC_GetAlarm	257
21.6.8	RTC_EnableInterrupts	258
21.6.9	RTC_DisableInterrupts	258
21.6.10	RTC_GetEnabledInterrupts	258
21.6.11	RTC_GetStatusFlags	258
21.6.12	RTC_ClearStatusFlags	259
21.6.13	RTC_StartTimer	259
21.6.14	RTC_StopTimer	259
21.6.15	RTC_SetOscCapLoad	259
21.6.16	RTC_Reset	260
Chapter	SAI: Serial Audio Interface	
22.1	Overview	261
22.2	Typical use case	261
22.2.1	SAI Send/receive using an interrupt method	261
22.2.2	SAI Send/receive using a DMA method	262
22.3	Data Structure Documentation	267
22.3.1	struct sai_config_t	267
22.3.2	struct sai_transfer_format_t	268
22.3.3	struct sai_transfer_t	268
22.3.4	struct _sai_handle	269
22.4	Macro Definition Documentation	269
22.4.1	SAI_XFER_QUEUE_SIZE	269
22.5	Enumeration Type Documentation	269
22.5.1	_sai_status_t	269
22.5.2	sai_protocol_t	270
22.5.3	sai_master_slave_t	270
22.5.4	sai_mono_stereo_t	270

Contents

Section Number	Title	Page Number
22.5.5	sai_sync_mode_t	270
22.5.6	sai_mclk_source_t	270
22.5.7	sai_bclk_source_t	271
22.5.8	_sai_interrupt_enable_t	271
22.5.9	_sai_dma_enable_t	271
22.5.10	_sai_flags	271
22.5.11	sai_reset_type_t	271
22.5.12	sai_sample_rate_t	272
22.5.13	sai_word_width_t	272
22.6	Function Documentation	272
22.6.1	SAI_TxInit	272
22.6.2	SAI_RxInit	273
22.6.3	SAI_TxGetDefaultConfig	273
22.6.4	SAI_RxGetDefaultConfig	273
22.6.5	SAI_Deinit	274
22.6.6	SAI_TxReset	274
22.6.7	SAI_RxReset	274
22.6.8	SAI_TxEnable	274
22.6.9	SAI_RxEnable	275
22.6.10	SAI_TxGetStatusFlag	275
22.6.11	SAI_TxClearStatusFlags	275
22.6.12	SAI_RxGetStatusFlag	275
22.6.13	SAI_RxClearStatusFlags	276
22.6.14	SAI_TxEnableInterrupts	276
22.6.15	SAI_RxEnableInterrupts	276
22.6.16	SAI_TxDisableInterrupts	277
22.6.17	SAI_RxDisableInterrupts	278
22.6.18	SAI_TxEnableDMA	278
22.6.19	SAI_RxEnableDMA	278
22.6.20	SAI_TxGetDataRegisterAddress	279
22.6.21	SAI_RxGetDataRegisterAddress	280
22.6.22	SAI_TxSetFormat	280
22.6.23	SAI_RxSetFormat	281
22.6.24	SAI_WriteBlocking	281
22.6.25	SAI_WriteData	281
22.6.26	SAI_ReadBlocking	282
22.6.27	SAI_ReadData	282
22.6.28	SAI_TransferTxCreateHandle	282
22.6.29	SAI_TransferRxCreateHandle	283
22.6.30	SAI_TransferTxSetFormat	283
22.6.31	SAI_TransferRxSetFormat	284
22.6.32	SAI_TransferSendNonBlocking	284
22.6.33	SAI_TransferReceiveNonBlocking	285
22.6.34	SAI_TransferGetSendCount	285

Contents

Section Number	Title	Page Number
22.6.35	SAI_TransferGetReceiveCount	286
22.6.36	SAI_TransferAbortSend	286
22.6.37	SAI_TransferAbortReceive	287
22.6.38	SAI_TransferTxHandleIRQ	287
22.6.39	SAI_TransferRxHandleIRQ	287
22.7	SAI DMA Driver	288
22.7.1	Overview	288
22.7.2	Data Structure Documentation	289
22.7.3	Function Documentation	289
22.8	SAI eDMA Driver	295
22.8.1	Overview	295
22.8.2	Data Structure Documentation	296
22.8.3	Function Documentation	297
Chapter	SIM: System Integration Module Driver	
23.1	Overview	303
23.2	Data Structure Documentation	304
23.2.1	struct sim_uid_t	304
23.3	Enumeration Type Documentation	304
23.3.1	_sim_usb_volt_reg_enable_mode	304
23.3.2	_sim_flash_mode	304
23.4	Function Documentation	304
23.4.1	SIM_SetUsbVoltRegulatorEnableMode	304
23.4.2	SIM_GetUniqueId	306
23.4.3	SIM_SetFlashMode	306
Chapter	SLCD: Segment LCD Driver	
24.1	Overview	307
24.2	Typical use case	307
24.2.1	SLCD Initialization operation	307
24.3	Data Structure Documentation	312
24.3.1	struct slcd_fault_detect_config_t	312
24.3.2	struct slcd_clock_config_t	312
24.3.3	struct slcd_config_t	313
24.4	Macro Definition Documentation	314
24.4.1	FSL_SLCD_DRIVER_VERSION	314

Contents

Section Number	Title	Page Number
24.5	Enumeration Type Documentation	314
24.5.1	slcd_power_supply_option_t	314
24.5.2	slcd_regulated_voltage_trim_t	315
24.5.3	slcd_load_adjust_t	315
24.5.4	slcd_clock_src_t	316
24.5.5	slcd_alt_clock_div_t	316
24.5.6	slcd_clock_prescaler_t	316
24.5.7	slcd_duty_cycle_t	317
24.5.8	slcd_phase_type_t	317
24.5.9	slcd_phase_index_t	317
24.5.10	slcd_display_mode_t	318
24.5.11	slcd_blink_mode_t	318
24.5.12	slcd_blink_rate_t	318
24.5.13	slcd_fault_detect_clock_prescaler_t	318
24.5.14	slcd_fault_detect_sample_window_width_t	319
24.5.15	slcd_interrupt_enable_t	319
24.5.16	slcd_lowpower_behavior	319
24.6	Function Documentation	319
24.6.1	SLCD_Init	319
24.6.2	SLCD_Deinit	320
24.6.3	SLCD_GetDefaultConfig	320
24.6.4	SLCD_StartDisplay	320
24.6.5	SLCD_StopDisplay	321
24.6.6	SLCD_StartBlinkMode	321
24.6.7	SLCD_StopBlinkMode	321
24.6.8	SLCD_SetBackPlanePhase	321
24.6.9	SLCD_SetFrontPlaneSegments	322
24.6.10	SLCD_SetFrontPlaneOnePhase	322
24.6.11	SLCD_EnablePadSafeState	323
24.6.12	SLCD_GetFaultDetectCounter	323
24.6.13	SLCD_EnableInterrupts	323
24.6.14	SLCD_DisableInterrupts	324
24.6.15	SLCD_GetInterruptStatus	324
24.6.16	SLCD_ClearInterruptStatus	324
Chapter	SMC: System Mode Controller Driver	
25.1	Overview	327
25.2	Typical use case	327
25.2.1	Enter wait or stop modes	327
25.3	Data Structure Documentation	329
25.3.1	struct smc_power_mode_vlls_config_t	329

Contents

Section Number	Title	Page Number
25.4	Macro Definition Documentation	329
25.4.1	FSL_SMC_DRIVER_VERSION	329
25.5	Enumeration Type Documentation	330
25.5.1	smc_power_mode_protection_t	330
25.5.2	smc_power_state_t	330
25.5.3	smc_run_mode_t	330
25.5.4	smc_stop_mode_t	330
25.5.5	smc_stop_submode_t	331
25.5.6	smc_partial_stop_option_t	331
25.5.7	_smc_status	331
25.6	Function Documentation	331
25.6.1	SMC_SetPowerModeProtection	331
25.6.2	SMC_GetPowerModeState	332
25.6.3	SMC_PreEnterStopModes	332
25.6.4	SMC_PostExitStopModes	332
25.6.5	SMC_PreEnterWaitModes	332
25.6.6	SMC_PostExitWaitModes	332
25.6.7	SMC_SetPowerModeRun	333
25.6.8	SMC_SetPowerModeWait	334
25.6.9	SMC_SetPowerModeStop	334
25.6.10	SMC_SetPowerModeVlpr	334
25.6.11	SMC_SetPowerModeVlpw	335
25.6.12	SMC_SetPowerModeVlps	335
25.6.13	SMC_SetPowerModeLls	335
25.6.14	SMC_SetPowerModeVlls	335
Chapter	SPI: Serial Peripheral Interface Driver	
26.1	Overview	337
26.2	SPI Driver	338
26.2.1	Overview	338
26.2.2	Typical use case	338
26.2.3	Data Structure Documentation	344
26.2.4	Macro Definition Documentation	346
26.2.5	Enumeration Type Documentation	346
26.2.6	Function Documentation	349
26.3	SPI DMA Driver	359
26.3.1	Overview	359
26.3.2	Data Structure Documentation	360
26.3.3	Typedef Documentation	360
26.3.4	Function Documentation	360

Contents

Section Number	Title	Page Number
26.4	SPI FreeRTOS driver	365
26.4.1	Overview	365
26.4.2	Function Documentation	365
Chapter	TPM: Timer PWM Module	
27.1	Overview	367
27.2	Typical use case	368
27.2.1	PWM output	368
27.3	Data Structure Documentation	372
27.3.1	struct tpm_chnl_pwm_signal_param_t	372
27.3.2	struct tpm_config_t	372
27.4	Enumeration Type Documentation	373
27.4.1	tpm_chnl_t	373
27.4.2	tpm_pwm_mode_t	373
27.4.3	tpm_pwm_level_select_t	373
27.4.4	tpm_trigger_select_t	374
27.4.5	tpm_output_compare_mode_t	374
27.4.6	tpm_input_capture_edge_t	374
27.4.7	tpm_clock_source_t	374
27.4.8	tpm_clock_prescale_t	375
27.4.9	tpm_interrupt_enable_t	375
27.4.10	tpm_status_flags_t	375
27.5	Function Documentation	375
27.5.1	TPM_Init	375
27.5.2	TPM_Deinit	376
27.5.3	TPM_GetDefaultConfig	376
27.5.4	TPM_SetupPwm	376
27.5.5	TPM_UpdatePwmDutycycle	377
27.5.6	TPM_UpdateChnlEdgeLevelSelect	377
27.5.7	TPM_SetupInputCapture	378
27.5.8	TPM_SetupOutputCompare	378
27.5.9	TPM_EnableInterrupts	378
27.5.10	TPM_DisableInterrupts	379
27.5.11	TPM_GetEnabledInterrupts	379
27.5.12	TPM_GetStatusFlags	379
27.5.13	TPM_ClearStatusFlags	379
27.5.14	TPM_SetTimerPeriod	380
27.5.15	TPM_GetCurrentTimerCount	380
27.5.16	TPM_StartTimer	381
27.5.17	TPM_StopTimer	382

Contents

Section Number	Title	Page Number
Chapter	TSI: Touch Sensing Input	
28.1	Overview	383
28.2	TSIv4 Driver	384
28.2.1	Overview	384
28.2.2	Typical use case	384
28.2.3	Data Structure Documentation	388
28.2.4	Enumeration Type Documentation	389
28.2.5	Function Documentation	393
Chapter	UART: Universal Asynchronous Receiver/Transmitter Driver	
29.1	Overview	405
29.2	UART Driver	406
29.2.1	Overview	406
29.2.2	Typical use case	406
29.2.3	Data Structure Documentation	414
29.2.4	Macro Definition Documentation	416
29.2.5	Typedef Documentation	416
29.2.6	Enumeration Type Documentation	416
29.2.7	Function Documentation	418
29.3	UART DMA Driver	431
29.3.1	Overview	431
29.3.2	Data Structure Documentation	431
29.3.3	Typedef Documentation	432
29.3.4	Function Documentation	432
29.4	UART eDMA Driver	436
29.4.1	Overview	436
29.4.2	Data Structure Documentation	436
29.4.3	Typedef Documentation	437
29.4.4	Function Documentation	437
29.5	UART FreeRTOS Driver	441
29.5.1	Overview	441
29.5.2	Data Structure Documentation	441
29.5.3	Function Documentation	442
Chapter	Clock Driver	
30.1	Overview	445
30.2	Get frequency	445

Contents

Section Number	Title	Page Number
30.3	External clock frequency	445
30.4	Data Structure Documentation	452
30.4.1	struct sim_clock_config_t	452
30.4.2	struct oscr_config_t	453
30.4.3	struct osc_config_t	453
30.4.4	struct mcg_pll_config_t	454
30.4.5	struct mcg_config_t	454
30.5	Macro Definition Documentation	455
30.5.1	MCG_CONFIG_CHECK_PARAM	455
30.5.2	FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL	455
30.5.3	FSL_CLOCK_DRIVER_VERSION	456
30.5.4	DMAMUX_CLOCKS	456
30.5.5	RTC_CLOCKS	456
30.5.6	SAI_CLOCKS	456
30.5.7	SPI_CLOCKS	456
30.5.8	SLCD_CLOCKS	457
30.5.9	PIT_CLOCKS	457
30.5.10	PORT_CLOCKS	457
30.5.11	TSI_CLOCKS	457
30.5.12	DAC_CLOCKS	457
30.5.13	LPTMR_CLOCKS	458
30.5.14	ADC16_CLOCKS	458
30.5.15	DMA_CLOCKS	458
30.5.16	UART0_CLOCKS	458
30.5.17	UART_CLOCKS	458
30.5.18	TPM_CLOCKS	459
30.5.19	I2C_CLOCKS	459
30.5.20	FTF_CLOCKS	459
30.5.21	CMP_CLOCKS	459
30.5.22	SYS_CLK	459
30.6	Enumeration Type Documentation	459
30.6.1	clock_name_t	459
30.6.2	clock_usb_src_t	460
30.6.3	clock_ip_name_t	460
30.6.4	osc_mode_t	460
30.6.5	_osc_cap_load	460
30.6.6	_oscer_enable_mode	461
30.6.7	mcg_fll_src_t	461
30.6.8	mcg_irc_mode_t	461
30.6.9	mcg_dm32_t	461
30.6.10	mcg_drs_t	461
30.6.11	mcg_pll_ref_src_t	462

Contents

Section Number	Title	Page Number
30.6.12	mcg_clkout_src_t	462
30.6.13	mcg_atm_select_t	462
30.6.14	mcg_oscsel_t	462
30.6.15	mcg_pll_clk_select_t	462
30.6.16	mcg_monitor_mode_t	462
30.6.17	_mcg_status	463
30.6.18	_mcg_status_flags_t	463
30.6.19	_mcg_ircclk_enable_mode	463
30.6.20	_mcg_pll_enable_mode	463
30.6.21	mcg_mode_t	464
30.7	Function Documentation	464
30.7.1	CLOCK_EnableClock	464
30.7.2	CLOCK_DisableClock	464
30.7.3	CLOCK_SetEr32kClock	464
30.7.4	CLOCK_SetTpmClock	464
30.7.5	CLOCK_SetLpsci0Clock	465
30.7.6	CLOCK_SetPllFllSelClock	465
30.7.7	CLOCK_SetClkOutClock	465
30.7.8	CLOCK_SetRtcClkOutClock	465
30.7.9	CLOCK_EnableUsbfs0Clock	465
30.7.10	CLOCK_DisableUsbfs0Clock	466
30.7.11	CLOCK_SetOutDiv	466
30.7.12	CLOCK_GetFreq	466
30.7.13	CLOCK_GetCoreSysClkFreq	466
30.7.14	CLOCK_GetPlatClkFreq	467
30.7.15	CLOCK_GetBusClkFreq	467
30.7.16	CLOCK_GetFlashClkFreq	467
30.7.17	CLOCK_GetPllFllSelClkFreq	467
30.7.18	CLOCK_GetEr32kClkFreq	467
30.7.19	CLOCK_GetOsc0ErClkFreq	467
30.7.20	CLOCK_SetSimConfig	468
30.7.21	CLOCK_SetSimSafeDivs	469
30.7.22	CLOCK_GetOutClkFreq	469
30.7.23	CLOCK_GetFllFreq	469
30.7.24	CLOCK_GetInternalRefClkFreq	469
30.7.25	CLOCK_GetFixedFreqClkFreq	470
30.7.26	CLOCK_GetPll0Freq	470
30.7.27	CLOCK_SetLowPowerEnable	470
30.7.28	CLOCK_SetInternalRefClkConfig	470
30.7.29	CLOCK_SetExternalRefClkConfig	471
30.7.30	CLOCK_SetFllExtRefDiv	471
30.7.31	CLOCK_EnablePll0	472
30.7.32	CLOCK_DisablePll0	472
30.7.33	CLOCK_CalcPllDiv	472

Contents

Section Number	Title	Page Number
30.7.34	CLOCK_SetOsc0MonitorMode	472
30.7.35	CLOCK_SetPll0MonitorMode	473
30.7.36	CLOCK_GetStatusFlags	473
30.7.37	CLOCK_ClearStatusFlags	473
30.7.38	OSC_SetExtRefClkConfig	474
30.7.39	OSC_SetCapLoad	474
30.7.40	CLOCK_InitOsc0	474
30.7.41	CLOCK_DeinitOsc0	475
30.7.42	CLOCK_SetXtal0Freq	475
30.7.43	CLOCK_SetXtal32Freq	475
30.7.44	CLOCK_TrimInternalRefClk	475
30.7.45	CLOCK_GetMode	476
30.7.46	CLOCK_SetFeiMode	476
30.7.47	CLOCK_SetFeeMode	477
30.7.48	CLOCK_SetFbiMode	477
30.7.49	CLOCK_SetFbeMode	478
30.7.50	CLOCK_SetBlpiMode	478
30.7.51	CLOCK_SetBlpeMode	479
30.7.52	CLOCK_SetPbeMode	479
30.7.53	CLOCK_SetPeeMode	480
30.7.54	CLOCK_ExternalModeToFbeModeQuick	480
30.7.55	CLOCK_InternalModeToFbiModeQuick	480
30.7.56	CLOCK_BootToFeiMode	481
30.7.57	CLOCK_BootToFeeMode	481
30.7.58	CLOCK_BootToBlpiMode	482
30.7.59	CLOCK_BootToBlpeMode	482
30.7.60	CLOCK_BootToPeeMode	483
30.7.61	CLOCK_SetMcgConfig	483
30.8	Variable Documentation	484
30.8.1	g_xtal0Freq	484
30.8.2	g_xtal32Freq	484
30.9	Multipurpose Clock Generator (MCG)	485
30.9.1	Function description	485
30.9.2	Typical use case	487
30.9.3	Code Configuration Option	490
Chapter	DMA Manager	
31.1	Overview	493
31.2	Function groups	493
31.2.1	DMAMGR Initialization and De-initialization	493
31.2.2	DMAMGR Operation	493

Contents

Section Number	Title	Page Number
31.3	Typical use case	493
31.3.1	DMAMGR static channel allocattion	493
31.3.2	DMAMGR dynamic channel allocation	493
31.4	Data Structure Documentation	494
31.4.1	struct dmamanager_handle_t	494
31.5	Macro Definition Documentation	495
31.5.1	DMAMGR_DYNAMIC_ALLOCATE	495
31.6	Enumeration Type Documentation	495
31.6.1	_dma_manager_status	495
31.7	Function Documentation	495
31.7.1	DMAMGR_Init	495
31.7.2	DMAMGR_Deinit	496
31.7.3	DMAMGR_RequestChannel	496
31.7.4	DMAMGR_ReleaseChannel	497
31.7.5	DMAMGR_IsChannelOccupied	498
Chapter	Secure Digital Card/Embedded MultiMedia Card (CARD)	
32.1	Overview	499
32.2	Data Structure Documentation	504
32.2.1	struct sd_card_t	504
32.2.2	struct sdio_card_t	505
32.2.3	struct mmc_card_t	506
32.2.4	struct mmc_boot_config_t	507
32.3	Macro Definition Documentation	507
32.3.1	FSL_SDMMC_DRIVER_VERSION	507
32.4	Enumeration Type Documentation	507
32.4.1	_sdmmc_status	507
32.4.2	_sd_card_flag	508
32.4.3	_mmc_card_flag	509
32.4.4	card_operation_voltage_t	509
32.4.5	_host_endian_mode	509
32.5	Function Documentation	509
32.5.1	SD_Init	509
32.5.2	SD_Deinit	510
32.5.3	SD_CheckReadOnly	511
32.5.4	SD_ReadBlocks	511
32.5.5	SD_WriteBlocks	512

Contents

Section Number	Title	Page Number
32.5.6	SD_EraseBlocks	513
32.5.7	MMC_Init	513
32.5.8	MMC_Deinit	514
32.5.9	MMC_CheckReadOnly	514
32.5.10	MMC_ReadBlocks	515
32.5.11	MMC_WriteBlocks	515
32.5.12	MMC_EraseGroups	516
32.5.13	MMC_SelectPartition	517
32.5.14	MMC_SetBootConfig	517
32.5.15	SDIO_CardInActive	518
32.5.16	SDIO_IO_Write_Direct	518
32.5.17	SDIO_IO_Read_Direct	519
32.5.18	SDIO_IO_Write_Extended	520
32.5.19	SDIO_IO_Read_Extended	520
32.5.20	SDIO_GetCardCapability	521
32.5.21	SDIO_SetBlockSize	521
32.5.22	SDIO_CardReset	522
32.5.23	SDIO_SetDataBusWidth	522
32.5.24	SDIO_SwitchToHighSpeed	523
32.5.25	SDIO_ReadCIS	523
32.5.26	SDIO_Init	524
32.5.27	SDIO_EnableIOInterrupt	525
32.5.28	SDIO_EnableIO	526
32.5.29	SDIO_SelectIO	526
32.5.30	SDIO_AbortIO	526
32.5.31	SDIO_DeInit	527
32.5.32	HOST_NotSupport	527
32.5.33	CardInsertDetect	527
32.5.34	HOST_Init	528
32.5.35	HOST_Deinit	528

Chapter SPI based Secure Digital Card (SDSPI)

33.1	Overview	529
33.2	Data Structure Documentation	531
33.2.1	struct sdspi_command_t	531
33.2.2	struct sdspi_host_t	531
33.2.3	struct sdspi_card_t	531
33.3	Enumeration Type Documentation	532
33.3.1	_sdspi_status	532
33.3.2	_sdspi_card_flag	533
33.3.3	sdspi_response_type_t	533

Contents

Section Number	Title	Page Number
33.4	Function Documentation	533
33.4.1	SDSPI_Init	533
33.4.2	SDSPI_Deinit	534
33.4.3	SDSPI_CheckReadOnly	534
33.4.4	SDSPI_ReadBlocks	535
33.4.5	SDSPI_WriteBlocks	535
Chapter	Debug Console	
34.1	Overview	537
34.2	Function groups	537
34.2.1	Initialization	537
34.2.2	Advanced Feature	538
34.3	Typical use case	541
34.4	Semihosting	543
34.4.1	Guide Semihosting for IAR	543
34.4.2	Guide Semihosting for Keil μ Vision	543
34.4.3	Guide Semihosting for KDS	545
34.4.4	Guide Semihosting for ATL	545
34.4.5	Guide Semihosting for ARMGCC	546
Chapter	Notification Framework	
35.1	Overview	549
35.2	Notifier Overview	549
35.3	Data Structure Documentation	551
35.3.1	struct notifier_notification_block_t	551
35.3.2	struct notifier_callback_config_t	552
35.3.3	struct notifier_handle_t	552
35.4	Typedef Documentation	553
35.4.1	notifier_user_config_t	553
35.4.2	notifier_user_function_t	553
35.4.3	notifier_callback_t	554
35.5	Enumeration Type Documentation	554
35.5.1	_notifier_status	554
35.5.2	notifier_policy_t	555
35.5.3	notifier_notification_type_t	555
35.5.4	notifier_callback_type_t	555

Contents

Section Number	Title	Page Number
35.6	Function Documentation	556
35.6.1	NOTIFIER_CreateHandle	556
35.6.2	NOTIFIER_SwitchConfig	557
35.6.3	NOTIFIER_GetErrorCallbackIndex	558
Chapter	Shell	
36.1	Overview	559
36.2	Function groups	559
36.2.1	Initialization	559
36.2.2	Advanced Feature	559
36.2.3	Shell Operation	560
36.3	Data Structure Documentation	561
36.3.1	struct shell_context_struct	561
36.3.2	struct shell_command_context_t	562
36.3.3	struct shell_command_context_list_t	562
36.4	Macro Definition Documentation	563
36.4.1	SHELL_USE_HISTORY	563
36.4.2	SHELL_SEARCH_IN_HIST	563
36.4.3	SHELL_USE_FILE_STREAM	563
36.4.4	SHELL_AUTO_COMPLETE	563
36.4.5	SHELL_BUFFER_SIZE	563
36.4.6	SHELL_MAX_ARGS	563
36.4.7	SHELL_HIST_MAX	563
36.4.8	SHELL_MAX_CMD	563
36.5	Typedef Documentation	563
36.5.1	send_data_cb_t	563
36.5.2	recv_data_cb_t	563
36.5.3	printf_data_t	563
36.5.4	cmd_function_t	563
36.6	Enumeration Type Documentation	563
36.6.1	fun_key_status_t	563
36.7	Function Documentation	564
36.7.1	SHELL_Init	564
36.7.2	SHELL_RegisterCommand	564
36.7.3	SHELL_Main	564

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™. 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 MCUXSDKRN) 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® 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](#).

Deliverable	Location
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_DMA_Busy = 5000
- 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_SMC_StopAbort = 3900
- kStatus_SPI_Busy = 1400
- kStatus_SPI_Idle = 1401
- kStatus_SPI_Error = 1402
- 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

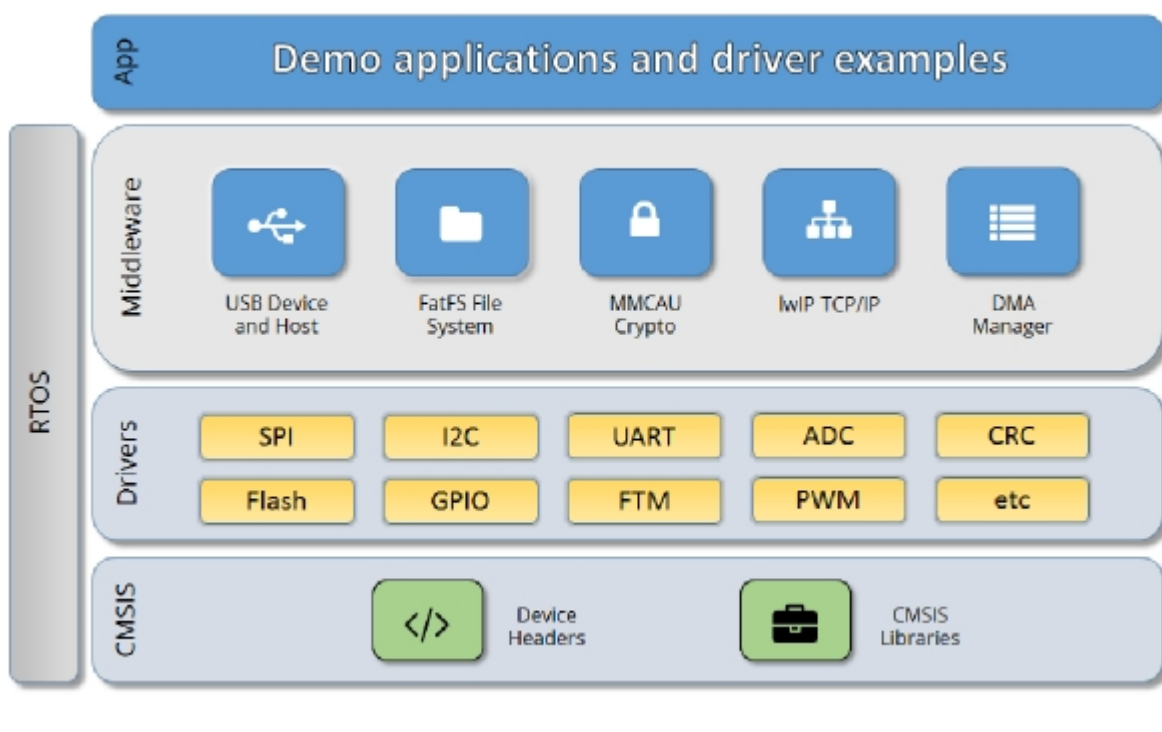


Figure 1: MCUXpresso SDK Block Diagram

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 a 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/.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 (B .). The MCUXpresso SDK drivers with transactional APIs provide the reimplement 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

Web Support: 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

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_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 (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.enabledDifferentialConversion = 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_Adc16ConversionDoneFlag = false;
volatile uint32_t g_Adc16ConversionValue;
volatile uint32_t g_Adc16InterruptCount = 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_COCO_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](#)

- *Generate an interrupt request once the conversion is completed.*
 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.

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 $x > \text{value1} \parallel x \leq \text{value2}$.

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".

Chapter 6

CMP: Analog Comparator Driver

6.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Analog Comparator (CMP) module of MCU-Xpresso 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_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
        );

    // 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.
- bool [enableTriggerMode](#)
Enable the trigger mode.

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`

6.3.1.0.0.4.7 `bool cmp_config_t::enableTriggerMode`

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".

Chapter 7

COP: Watchdog Driver

7.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Computer Operating Properly module (COP) of MCUXpresso SDK devices.

7.2 Typical use case

```
cop_config_t config;
COP_GetDefaultConfig(&config);
config.timeoutCycles = kCOP_2Power8CyclesOr2Power16Cycles;
COP_Init(sim_base, &config);
```

Data Structures

- struct `cop_config_t`
Describes COP configuration structure. [More...](#)

Enumerations

- enum `cop_clock_source_t` {
 `kCOP_LpoClock` = 0U,
 `kCOP_BusClock` = 3U }
 COP clock source selection.
- enum `cop_timeout_cycles_t` {
 `kCOP_2Power5CyclesOr2Power13Cycles` = 1U,
 `kCOP_2Power8CyclesOr2Power16Cycles` = 2U,
 `kCOP_2Power10CyclesOr2Power18Cycles` = 3U }
 Define the COP timeout cycles.

Driver version

- #define `FSL_COP_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)
 COP driver version 2.0.0.

COP refresh sequence.

- #define `COP_FIRST_BYTE_OF_REFRESH` (0x55U)
 First byte of refresh sequence.
- #define `COP_SECOND_BYTE_OF_REFRESH` (0xAAU)
 Second byte of refresh sequence.

Enumeration Type Documentation

COP Functional Operation

- void [COP_GetDefaultConfig](#) ([cop_config_t](#) *config)
Initializes the COP configuration structure.
- void [COP_Init](#) (SIM_Type *base, const [cop_config_t](#) *config)
Initializes the COP module.
- static void [COP_Disable](#) (SIM_Type *base)
De-initializes the COP module.
- void [COP_Refresh](#) (SIM_Type *base)
Refreshes the COP timer.

7.3 Data Structure Documentation

7.3.1 struct cop_config_t

Data Fields

- bool [enableWindowMode](#)
COP run mode: window mode or normal mode.
- [cop_clock_source_t](#) [clockSource](#)
Set COP clock source.
- [cop_timeout_cycles_t](#) [timeoutCycles](#)
Set COP timeout value.

7.4 Macro Definition Documentation

7.4.1 #define FSL_COP_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

7.5 Enumeration Type Documentation

7.5.1 enum cop_clock_source_t

Enumerator

kCOP_LpoClock COP clock sourced from LPO.
kCOP_BusClock COP clock sourced from Bus clock.

7.5.2 enum cop_timeout_cycles_t

Enumerator

kCOP_2Power5CyclesOr2Power13Cycles 2^5 or 2^{13} clock cycles
kCOP_2Power8CyclesOr2Power16Cycles 2^8 or 2^{16} clock cycles
kCOP_2Power10CyclesOr2Power18Cycles 2^{10} or 2^{18} clock cycles

7.6 Function Documentation

7.6.1 void COP_GetDefaultConfig (cop_config_t * *config*)

This function initializes the COP configuration structure to default values. The default values are:

```
*  copConfig->enableWindowMode = false;
*  copConfig->timeoutMode = kCOP_LongTimeoutMode;
*  copConfig->enableStop = false;
*  copConfig->enableDebug = false;
*  copConfig->clockSource = kCOP_LpoClock;
*  copConfig->timeoutCycles = kCOP_2Power10CyclesOr2Power18Cycles;
*
```

Parameters

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

See Also

[cop_config_t](#)

7.6.2 void COP_Init (SIM_Type * *base*, const cop_config_t * *config*)

This function configures the COP. After it is called, the COP starts running according to the configuration. Because all COP control registers are write-once only, the COP_Init function and the COP_Disable function can be called only once. A second call has no effect.

Example:

```
*  cop_config_t config;
*  COP_GetDefaultConfig(&config);
*  config.timeoutCycles = kCOP_2Power8CyclesOr2Power16Cycles
*  ;
*  COP_Init(sim_base,&config);
*
```

Parameters

<i>base</i>	SIM peripheral base address.
<i>config</i>	The configuration of COP.

7.6.3 static void COP_Disable (SIM_Type * *base*) [inline], [static]

This dedicated function is not provided. Instead, the COP_Disable function can be used to disable the COP.

Function Documentation

Disables the COP module.

This function disables the COP Watchdog. Note: The COP configuration register is a write-once after reset. To disable the COP Watchdog, call this function first.

Parameters

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

7.6.4 void COP_Refresh (SIM_Type * *base*)

This function feeds the COP.

Parameters

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

Chapter 8

DAC: Digital-to-Analog Converter Driver

8.1 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.

8.2 Typical use case

8.2.1 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);
```

8.2.2 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_GetDefaultBufferConfig(&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 += (0xFFFU /
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_BufferReadPointerTopPositionFlag](#) = DAC_SR_DACBFRPTF_MASK,
 [kDAC_BufferReadPointerBottomPositionFlag](#) = DAC_SR_DACBFRPBF_MASK }

- DAC buffer flags.*
 - enum `_dac_buffer_interrupt_enable` {
`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_work_mode_t` {
`kDAC_BufferWorkAsNormalMode` = `0U`,
`kDAC_BufferWorkAsOneTimeScanMode` }
- 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.

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.

Data Structure Documentation

- 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.

8.3 Data Structure Documentation

8.3.1 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.

8.3.1.0.0.7 Field Documentation

8.3.1.0.0.7.1 [dac_reference_voltage_source_t](#) [dac_config_t::referenceVoltageSource](#)

8.3.1.0.0.7.2 [bool](#) [dac_config_t::enableLowPowerMode](#)

8.3.2 struct dac_buffer_config_t

Data Fields

- [dac_buffer_trigger_mode_t](#) triggerMode
Select the buffer's trigger mode.
- [dac_buffer_work_mode_t](#) workMode
Select the buffer's work mode.
- uint8_t [upperLimit](#)
Set the upper limit for the buffer index.

8.3.2.0.0.8 Field Documentation

8.3.2.0.0.8.1 [dac_buffer_trigger_mode_t](#) [dac_buffer_config_t::triggerMode](#)

8.3.2.0.0.8.2 [dac_buffer_work_mode_t](#) [dac_buffer_config_t::workMode](#)

8.3.2.0.0.8.3 [uint8_t](#) [dac_buffer_config_t::upperLimit](#)

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

8.4 Macro Definition Documentation

8.4.1 #define FSL_DAC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

8.5 Enumeration Type Documentation

8.5.1 enum _dac_buffer_status_flags

Enumerator

kDAC_BufferReadPointerTopPositionFlag DAC Buffer Read Pointer Top Position Flag.

kDAC_BufferReadPointerBottomPositionFlag DAC Buffer Read Pointer Bottom Position Flag.

8.5.2 enum _dac_buffer_interrupt_enable

Enumerator

kDAC_BufferReadPointerTopInterruptEnable DAC Buffer Read Pointer Top Flag Interrupt Enable.

kDAC_BufferReadPointerBottomInterruptEnable DAC Buffer Read Pointer Bottom Flag Interrupt Enable.

8.5.3 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.

8.5.4 enum dac_buffer_trigger_mode_t

Enumerator

kDAC_BufferTriggerByHardwareMode The DAC hardware trigger is selected.

kDAC_BufferTriggerBySoftwareMode The DAC software trigger is selected.

8.5.5 enum dac_buffer_work_mode_t

Enumerator

kDAC_BufferWorkAsNormalMode Normal mode.

kDAC_BufferWorkAsOneTimeScanMode One-Time Scan mode.

Function Documentation

8.6 Function Documentation

8.6.1 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".

8.6.2 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.
-------------	------------------------------

8.6.3 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".
---------------	---

8.6.4 static void DAC_Enable (DAC_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

8.6.5 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.

8.6.6 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".

8.6.7 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

Function Documentation

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

8.6.8 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.

8.6.9 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.

8.6.10 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.
-------------	------------------------------

8.6.11 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.

8.6.12 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.

8.6.13 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".

8.6.14 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".

8.6.15 uint32_t DAC_GetBufferStatusFlags (DAC_Type * *base*)

Function Documentation

Parameters

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

Returns

Mask value for the asserted flags. See "_dac_buffer_status_flags".

8.6.16 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".

Chapter 9

DMA: Direct Memory Access Controller Driver

9.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Direct Memory Access (DMA) of MCU-Xpresso SDK devices.

9.2 Typical use case

9.2.1 DMA Operation

```
dma_transfer_config_t transferConfig;
uint32_t transferDone = false;

DMA_Init(DMA0);
DMA_CreateHandle(&g_DMA_Handle, DMA0, channel);
DMA_InstallCallback(&g_DMA_Handle, DMA_Callback, &transferDone);
DMA_PrepareTransfer(&transferConfig, srcAddr, srcWidth, destAddr, destWidth,
    transferBytes,
    kDMA_MemoryToMemory);
DMA_SubmitTransfer(&g_DMA_Handle, &transferConfig, true);
DMA_StartTransfer(&g_DMA_Handle);
/* Wait for DMA transfer finish */
while (transferDone != true);
```

Data Structures

- struct `dma_transfer_config_t`
DMA transfer configuration structure. [More...](#)
- struct `dma_channel_link_config_t`
DMA transfer configuration structure. [More...](#)
- struct `dma_handle_t`
DMA DMA handle structure. [More...](#)

Typedefs

- typedef void(* `dma_callback`)(struct `_dma_handle` *handle, void *userData)
Callback function prototype for the DMA driver.

Enumerations

- enum `_dma_channel_status_flags` {
 `kDMA_TransactionsBCRFlag` = `DMA_DSR_BCR_BCR_MASK`,
 `kDMA_TransactionsDoneFlag` = `DMA_DSR_BCR_DONE_MASK`,
 `kDMA_TransactionsBusyFlag` = `DMA_DSR_BCR_BSY_MASK`,
 `kDMA_TransactionsRequestFlag` = `DMA_DSR_BCR_REQ_MASK`,
 `kDMA_BusErrorOnDestinationFlag` = `DMA_DSR_BCR_BED_MASK`,
 `kDMA_BusErrorOnSourceFlag` = `DMA_DSR_BCR_BES_MASK`,
}

Typical use case

```
kDMA_ConfigurationErrorFlag = DMA_DSR_BCR_CE_MASK }
```

status flag for the DMA driver.

- enum `dma_transfer_size_t` {
 `kDMA_Transfersize32bits` = 0x0U,
 `kDMA_Transfersize8bits`,
 `kDMA_Transfersize16bits` }
 DMA transfer size type.
- enum `dma_modulo_t` {
 `kDMA_ModuloDisable` = 0x0U,
 `kDMA_Modulo16Bytes`,
 `kDMA_Modulo32Bytes`,
 `kDMA_Modulo64Bytes`,
 `kDMA_Modulo128Bytes`,
 `kDMA_Modulo256Bytes`,
 `kDMA_Modulo512Bytes`,
 `kDMA_Modulo1KBytes`,
 `kDMA_Modulo2KBytes`,
 `kDMA_Modulo4KBytes`,
 `kDMA_Modulo8KBytes`,
 `kDMA_Modulo16KBytes`,
 `kDMA_Modulo32KBytes`,
 `kDMA_Modulo64KBytes`,
 `kDMA_Modulo128KBytes`,
 `kDMA_Modulo256KBytes` }
 Configuration type for the DMA modulo.
- enum `dma_channel_link_type_t` {
 `kDMA_ChannelLinkDisable` = 0x0U,
 `kDMA_ChannelLinkChannel1AndChannel2`,
 `kDMA_ChannelLinkChannel1`,
 `kDMA_ChannelLinkChannel1AfterBCR0` }
 DMA channel link type.
- enum `dma_transfer_type_t` {
 `kDMA_MemoryToMemory` = 0x0U,
 `kDMA_PeripheralToMemory`,
 `kDMA_MemoryToPeripheral` }
 DMA transfer type.
- enum `dma_transfer_options_t` {
 `kDMA_NoOptions` = 0x0U,
 `kDMA_EnableInterrupt` }
 DMA transfer options.
- enum `_dma_transfer_status`
 DMA transfer status.

Driver version

- #define `FSL_DMA_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 1)`)
 DMA driver version 2.0.1.

DMA Initialization and De-initialization

- void [DMA_Init](#) (DMA_Type *base)
Initializes the DMA peripheral.
- void [DMA_Deinit](#) (DMA_Type *base)
Deinitializes the DMA peripheral.

DMA Channel Operation

- void [DMA_ResetChannel](#) (DMA_Type *base, uint32_t channel)
Resets the DMA channel.
- void [DMA_SetTransferConfig](#) (DMA_Type *base, uint32_t channel, const [dma_transfer_config_t](#) *config)
Configures the DMA transfer attribute.
- void [DMA_SetChannelLinkConfig](#) (DMA_Type *base, uint32_t channel, const [dma_channel_link_config_t](#) *config)
Configures the DMA channel link feature.
- static void [DMA_SetSourceAddress](#) (DMA_Type *base, uint32_t channel, uint32_t srcAddr)
Sets the DMA source address for the DMA transfer.
- static void [DMA_SetDestinationAddress](#) (DMA_Type *base, uint32_t channel, uint32_t destAddr)
Sets the DMA destination address for the DMA transfer.
- static void [DMA_SetTransferSize](#) (DMA_Type *base, uint32_t channel, uint32_t size)
Sets the DMA transfer size for the DMA transfer.
- void [DMA_SetModulo](#) (DMA_Type *base, uint32_t channel, [dma_modulo_t](#) srcModulo, [dma_modulo_t](#) destModulo)
Sets the DMA modulo for the DMA transfer.
- static void [DMA_EnableCycleSteal](#) (DMA_Type *base, uint32_t channel, bool enable)
Enables the DMA cycle steal for the DMA transfer.
- static void [DMA_EnableAutoAlign](#) (DMA_Type *base, uint32_t channel, bool enable)
Enables the DMA auto align for the DMA transfer.
- static void [DMA_EnableAsyncRequest](#) (DMA_Type *base, uint32_t channel, bool enable)
Enables the DMA async request for the DMA transfer.
- static void [DMA_EnableInterrupts](#) (DMA_Type *base, uint32_t channel)
Enables an interrupt for the DMA transfer.
- static void [DMA_DisableInterrupts](#) (DMA_Type *base, uint32_t channel)
Disables an interrupt for the DMA transfer.

DMA Channel Transfer Operation

- static void [DMA_EnableChannelRequest](#) (DMA_Type *base, uint32_t channel)
Enables the DMA hardware channel request.
- static void [DMA_DisableChannelRequest](#) (DMA_Type *base, uint32_t channel)
Disables the DMA hardware channel request.
- static void [DMA_TriggerChannelStart](#) (DMA_Type *base, uint32_t channel)
Starts the DMA transfer with a software trigger.

DMA Channel Status Operation

- static uint32_t [DMA_GetRemainingBytes](#) (DMA_Type *base, uint32_t channel)
Gets the remaining bytes of the current DMA transfer.
- static uint32_t [DMA_GetChannelStatusFlags](#) (DMA_Type *base, uint32_t channel)

Data Structure Documentation

- Gets the DMA channel status flags.*
- static void [DMA_ClearChannelStatusFlags](#) (DMA_Type *base, uint32_t channel, uint32_t mask)
Clears the DMA channel status flags.

DMA Channel Transactional Operation

- void [DMA_CreateHandle](#) (dma_handle_t *handle, DMA_Type *base, uint32_t channel)
Creates the DMA handle.
- void [DMA_SetCallback](#) (dma_handle_t *handle, dma_callback callback, void *userData)
Sets the DMA callback function.
- void [DMA_PrepareTransfer](#) (dma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t transferBytes, dma_transfer_type_t type)
Prepares the DMA transfer configuration structure.
- status_t [DMA_SubmitTransfer](#) (dma_handle_t *handle, const dma_transfer_config_t *config, uint32_t options)
Submits the DMA transfer request.
- static void [DMA_StartTransfer](#) (dma_handle_t *handle)
DMA starts a transfer.
- static void [DMA_StopTransfer](#) (dma_handle_t *handle)
DMA stops a transfer.
- void [DMA_AbortTransfer](#) (dma_handle_t *handle)
DMA aborts a transfer.
- void [DMA_HandleIRQ](#) (dma_handle_t *handle)
DMA IRQ handler for current transfer complete.

9.3 Data Structure Documentation

9.3.1 struct dma_transfer_config_t

Data Fields

- uint32_t [srcAddr](#)
DMA transfer source address.
- uint32_t [destAddr](#)
DMA destination address.
- bool [enableSrcIncrement](#)
Source address increase after each transfer.
- [dma_transfer_size_t](#) [srcSize](#)
Source transfer size unit.
- bool [enableDestIncrement](#)
Destination address increase after each transfer.
- [dma_transfer_size_t](#) [destSize](#)
Destination transfer unit.
- uint32_t [transferSize](#)
The number of bytes to be transferred.

9.3.1.0.0.9 Field Documentation

9.3.1.0.0.9.1 `uint32_t dma_transfer_config_t::srcAddr`

9.3.1.0.0.9.2 `uint32_t dma_transfer_config_t::destAddr`

9.3.1.0.0.9.3 `bool dma_transfer_config_t::enableSrcIncrement`

9.3.1.0.0.9.4 `dma_transfer_size_t dma_transfer_config_t::srcSize`

9.3.1.0.0.9.5 `bool dma_transfer_config_t::enableDestIncrement`

9.3.1.0.0.9.6 `dma_transfer_size_t dma_transfer_config_t::destSize`

9.3.1.0.0.9.7 `uint32_t dma_transfer_config_t::transferSize`

9.3.2 struct dma_channel_link_config_t

Data Fields

- `dma_channel_link_type_t linkType`
Channel link type.
- `uint32_t channel1`
The index of channel 1.
- `uint32_t channel2`
The index of channel 2.

9.3.2.0.0.10 Field Documentation

9.3.2.0.0.10.1 `dma_channel_link_type_t dma_channel_link_config_t::linkType`

9.3.2.0.0.10.2 `uint32_t dma_channel_link_config_t::channel1`

9.3.2.0.0.10.3 `uint32_t dma_channel_link_config_t::channel2`

9.3.3 struct dma_handle_t

Data Fields

- `DMA_Type * base`
DMA peripheral address.
- `uint8_t channel`
DMA channel used.
- `dma_callback callback`
DMA callback function.
- `void * userData`
Callback parameter.

Enumeration Type Documentation

9.3.3.0.0.11 Field Documentation

9.3.3.0.0.11.1 **DMA_Type*** dma_handle_t::base

9.3.3.0.0.11.2 **uint8_t** dma_handle_t::channel

9.3.3.0.0.11.3 **dma_callback** dma_handle_t::callback

9.3.3.0.0.11.4 **void*** dma_handle_t::userData

9.4 Macro Definition Documentation

9.4.1 **#define** FSL_DMA_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

9.5 Typedef Documentation

9.5.1 **typedef void**(* dma_callback)(struct _dma_handle *handle, void *userData)

9.6 Enumeration Type Documentation

9.6.1 **enum** _dma_channel_status_flags

Enumerator

kDMA_TransactionsBCRFlag Contains the number of bytes yet to be transferred for a given block.

kDMA_TransactionsDoneFlag Transactions Done.

kDMA_TransactionsBusyFlag Transactions Busy.

kDMA_TransactionsRequestFlag Transactions Request.

kDMA_BusErrorOnDestinationFlag Bus Error on Destination.

kDMA_BusErrorOnSourceFlag Bus Error on Source.

kDMA_ConfigurationErrorFlag Configuration Error.

9.6.2 **enum** dma_transfer_size_t

Enumerator

kDMA_Transfersize32bits 32 bits are transferred for every read/write

kDMA_Transfersize8bits 8 bits are transferred for every read/write

kDMA_Transfersize16bits 16b its are transferred for every read/write

9.6.3 **enum** dma_modulo_t

Enumerator

kDMA_ModuloDisable Buffer disabled.

kDMA_Modulo16Bytes Circular buffer size is 16 bytes.
kDMA_Modulo32Bytes Circular buffer size is 32 bytes.
kDMA_Modulo64Bytes Circular buffer size is 64 bytes.
kDMA_Modulo128Bytes Circular buffer size is 128 bytes.
kDMA_Modulo256Bytes Circular buffer size is 256 bytes.
kDMA_Modulo512Bytes Circular buffer size is 512 bytes.
kDMA_Modulo1KBytes Circular buffer size is 1 KB.
kDMA_Modulo2KBytes Circular buffer size is 2 KB.
kDMA_Modulo4KBytes Circular buffer size is 4 KB.
kDMA_Modulo8KBytes Circular buffer size is 8 KB.
kDMA_Modulo16KBytes Circular buffer size is 16 KB.
kDMA_Modulo32KBytes Circular buffer size is 32 KB.
kDMA_Modulo64KBytes Circular buffer size is 64 KB.
kDMA_Modulo128KBytes Circular buffer size is 128 KB.
kDMA_Modulo256KBytes Circular buffer size is 256 KB.

9.6.4 enum dma_channel_link_type_t

Enumerator

kDMA_ChannelLinkDisable No channel link.
kDMA_ChannelLinkChannel1AndChannel2 Perform a link to channel LCH1 after each cycle-steal transfer. followed by a link to LCH2 after the BCR decrements to 0.
kDMA_ChannelLinkChannel1 Perform a link to LCH1 after each cycle-steal transfer.
kDMA_ChannelLinkChannel1AfterBCR0 Perform a link to LCH1 after the BCR decrements.

9.6.5 enum dma_transfer_type_t

Enumerator

kDMA_MemoryToMemory Memory to Memory transfer.
kDMA_PeripheralToMemory Peripheral to Memory transfer.
kDMA_MemoryToPeripheral Memory to Peripheral transfer.

9.6.6 enum dma_transfer_options_t

Enumerator

kDMA_NoOptions Transfer without options.
kDMA_EnableInterrupt Enable interrupt while transfer complete.

9.7 Function Documentation

9.7.1 void DMA_Init (DMA_Type * *base*)

This function ungates the DMA clock.

Parameters

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

9.7.2 void DMA_Deinit (DMA_Type * *base*)

This function gates the DMA clock.

Parameters

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

9.7.3 void DMA_ResetChannel (DMA_Type * *base*, uint32_t *channel*)

Sets all register values to reset values and enables the cycle steal and auto stop channel request features.

Parameters

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

9.7.4 void DMA_SetTransferConfig (DMA_Type * *base*, uint32_t *channel*, const dma_transfer_config_t * *config*)

This function configures the transfer attribute including the source address, destination address, transfer size, and so on. This example shows how to set up the the [dma_transfer_config_t](#) parameters and how to call the DMA_ConfigBasicTransfer function.

```
*  dma_transfer_config_t transferConfig;
*  memset(&transferConfig, 0, sizeof(transferConfig));
*  transferConfig.srcAddr = (uint32_t)srcAddr;
*  transferConfig.destAddr = (uint32_t)destAddr;
*  transferConfig.enableSrcIncrement = true;
*  transferConfig.enableDestIncrement = true;
*  transferConfig.srcSize = kDMA_Transfersize32bits;
*  transferConfig.destSize = kDMA_Transfersize32bits;
*  transferConfig.transferSize = sizeof(uint32_t) * BUFF_LENGTH;
*  DMA_SetTransferConfig(DMA0, 0, &transferConfig);
*
```

Function Documentation

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>config</i>	Pointer to the DMA transfer configuration structure.

9.7.5 void DMA_SetChannelLinkConfig (DMA_Type * *base*, uint32_t *channel*, const dma_channel_link_config_t * *config*)

This function allows DMA channels to have their transfers linked. The current DMA channel triggers a DMA request to the linked channels (LCH1 or LCH2) depending on the channel link type. Perform a link to channel LCH1 after each cycle-steal transfer followed by a link to LCH2 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AndChannel2. Perform a link to LCH1 after each cycle-steal transfer if the type is kDMA_ChannelLinkChannel1. Perform a link to LCH1 after the BCR decrements to 0 if the type is kDMA_ChannelLinkChannel1AfterBCR0.

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>config</i>	Pointer to the channel link configuration structure.

9.7.6 static void DMA_SetSourceAddress (DMA_Type * *base*, uint32_t *channel*, uint32_t *srcAddr*) [inline], [static]

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>srcAddr</i>	DMA source address.

9.7.7 static void DMA_SetDestinationAddress (DMA_Type * *base*, uint32_t *channel*, uint32_t *destAddr*) [inline], [static]

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>destAddr</i>	DMA destination address.

9.7.8 static void DMA_SetTransferSize (DMA_Type * *base*, uint32_t *channel*, uint32_t *size*) [inline], [static]

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>size</i>	The number of bytes to be transferred.

9.7.9 void DMA_SetModulo (DMA_Type * *base*, uint32_t *channel*, dma_modulo_t *srcModulo*, dma_modulo_t *destModulo*)

This function defines a specific address range specified to be the value after (SAR + SSIZE)/(DAR + DS-IZE) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>srcModulo</i>	source address modulo.
<i>destModulo</i>	destination address modulo.

9.7.10 static void DMA_EnableCycleSteal (DMA_Type * *base*, uint32_t *channel*, bool *enable*) [inline], [static]

If the cycle steal feature is enabled (true), the DMA controller forces a single read/write transfer per request, or it continuously makes read/write transfers until the BCR decrements to 0.

Function Documentation

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>enable</i>	The command for enable (true) or disable (false).

9.7.11 static void DMA_EnableAutoAlign (DMA_Type * *base*, uint32_t *channel*, bool *enable*) [inline], [static]

If the auto align feature is enabled (true), the appropriate address register increments regardless of DINC or SINC.

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>enable</i>	The command for enable (true) or disable (false).

9.7.12 static void DMA_EnableAsyncRequest (DMA_Type * *base*, uint32_t *channel*, bool *enable*) [inline], [static]

If the async request feature is enabled (true), the DMA supports asynchronous DREQs while the MCU is in stop mode.

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>enable</i>	The command for enable (true) or disable (false).

9.7.13 static void DMA_EnableInterrupts (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

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

9.7.14 static void DMA_DisableInterrupts (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

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

9.7.15 static void DMA_EnableChannelRequest (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	The DMA channel number.

9.7.16 static void DMA_DisableChannelRequest (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

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

9.7.17 static void DMA_TriggerChannelStart (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

This function starts only one read/write iteration.

Function Documentation

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	The DMA channel number.

9.7.18 static uint32_t DMA_GetRemainingBytes (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

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

Returns

The number of bytes which have not been transferred yet.

9.7.19 static uint32_t DMA_GetChannelStatusFlags (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

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

Returns

The mask of the channel status. Use the `_dma_channel_status_flags` type to decode the return 32 bit variables.

9.7.20 static void DMA_ClearChannelStatusFlags (DMA_Type * *base*, uint32_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.
<i>mask</i>	The mask of the channel status to be cleared. Use the defined <code>_dma_channel_status- _flags</code> type.

9.7.21 void DMA_CreateHandle (dma_handle_t * *handle*, DMA_Type * *base*, uint32_t *channel*)

This function is called first if using the transactional API for the DMA. This function initializes the internal state of the DMA handle.

Parameters

<i>handle</i>	DMA handle pointer. The DMA handle stores callback function and parameters.
<i>base</i>	DMA peripheral base address.
<i>channel</i>	DMA channel number.

9.7.22 void DMA_SetCallback (dma_handle_t * *handle*, dma_callback *callback*, void * *userData*)

This callback is called in the DMA IRQ handler. Use the callback to do something after the current transfer complete.

Parameters

<i>handle</i>	DMA handle pointer.
<i>callback</i>	DMA callback function pointer.
<i>userData</i>	Parameter for callback function. If it is not needed, just set to NULL.

9.7.23 void DMA_PrepareTransfer (dma_transfer_config_t * *config*, void * *srcAddr*, uint32_t *srcWidth*, void * *destAddr*, uint32_t *destWidth*, uint32_t *transferBytes*, dma_transfer_type_t *type*)

This function prepares the transfer configuration structure according to the user input.

Function Documentation

Parameters

<i>config</i>	Pointer to the user configuration structure of type dma_transfer_config_t .
<i>srcAddr</i>	DMA transfer source address.
<i>srcWidth</i>	DMA transfer source address width (byte).
<i>destAddr</i>	DMA transfer destination address.
<i>destWidth</i>	DMA transfer destination address width (byte).
<i>transferBytes</i>	DMA transfer bytes to be transferred.
<i>type</i>	DMA transfer type.

9.7.24 **status_t DMA_SubmitTransfer (dma_handle_t * *handle*, const dma_transfer_config_t * *config*, uint32_t *options*)**

This function submits the DMA transfer request according to the transfer configuration structure.

Parameters

<i>handle</i>	DMA handle pointer.
<i>config</i>	Pointer to DMA transfer configuration structure.
<i>options</i>	Additional configurations for transfer. Use the defined dma_transfer_options_t type.

Return values

<i>kStatus_DMA_Success</i>	It indicates that the DMA submit transfer request succeeded.
<i>kStatus_DMA_Busy</i>	It indicates that the DMA is busy. Submit transfer request is not allowed.

Note

This function can't process multi transfer request.

9.7.25 **static void DMA_StartTransfer (dma_handle_t * *handle*) [inline], [static]**

This function enables the channel request. Call this function after submitting a transfer request.

Parameters

<i>handle</i>	DMA handle pointer.
---------------	---------------------

Return values

<i>kStatus_DMA_Success</i>	It indicates that the DMA start transfer succeed.
<i>kStatus_DMA_Busy</i>	It indicates that the DMA has started a transfer.

9.7.26 static void DMA_StopTransfer (dma_handle_t * *handle*) [inline], [static]

This function disables the channel request to stop a DMA transfer. The transfer can be resumed by calling the DMA_StartTransfer.

Parameters

<i>handle</i>	DMA handle pointer.
---------------	---------------------

9.7.27 void DMA_AbortTransfer (dma_handle_t * *handle*)

This function disables the channel request and clears all status bits. Submit another transfer after calling this API.

Parameters

<i>handle</i>	DMA handle pointer.
---------------	---------------------

9.7.28 void DMA_HandleIRQ (dma_handle_t * *handle*)

This function clears the channel interrupt flag and calls the callback function if it is not NULL.

Parameters

<i>handle</i>	DMA handle pointer.
---------------	---------------------

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)`)

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

C90TFS Flash Driver

11.1 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_ResourceOptionFlashIfc`,
 `kFLASH_ResourceOptionVersionId` = 0x01U }

Enumeration for the two possible options of flash read resource command.

- enum `_flash_read_resource_range` {

```

kFLASH_ResourceRangePflashIfrSizeInBytes = 256U,
kFLASH_ResourceRangeVersionIdSizeInBytes = 8U,
kFLASH_ResourceRangeVersionIdStart = 0x00U,
kFLASH_ResourceRangeVersionIdEnd = 0x07U,
kFLASH_ResourceRangePflashSwapIfrStart = 0x10000U,
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.

Reading

Programs flash with data at locations passed in through parameters via the Program Section command.

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

kStatus_FLASH_Success	API was executed successfully.
kStatus_FLASH_InvalidArgument	An invalid argument is provided.
kStatus_FLASH_AlignmentError	Parameter is not aligned with specified baseline.

<i>kStatus_FLASH_Address-Error</i>	Address is out of range.
<i>kStatus_FLASH_Set-FlexramAsRamError</i>	Failed to set flexram as RAM.
<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.
<i>kStatus_FLASH_Recover-FlexramAsEepromError</i>	Failed to recover FlexRAM as EEPROM.

Programs the EEPROM with data at locations passed in through parameters.

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.
<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_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Address-Error</i>	Address is out of range.

Overview

<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.

- 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 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 [FLASH_GetProperty](#) (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.

Flash Protection Utilities

Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

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>eeepromData-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_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.

- 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.

11.2 Data Structure Documentation

11.2.1 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](#)
Execute-in-RAM function: flash_common_bit_operation.

11.2.1.0.0.12 Field Documentation

11.2.1.0.0.12.1 uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount

11.2.1.0.0.12.2 uint32_t* flash_execute_in_ram_function_config_t::flashRunCommand

11.2.1.0.0.12.3 uint32_t* flash_execute_in_ram_function_config_t::flashCommonBitOperation

11.2.2 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.

11.2.2.0.0.13 Field Documentation

11.2.2.0.0.13.1 flash_swap_state_t flash_swap_state_config_t::flashSwapState

11.2.2.0.0.13.2 flash_swap_block_status_t flash_swap_state_config_t::currentSwapBlockStatus

11.2.2.0.0.13.3 flash_swap_block_status_t flash_swap_state_config_t::nextSwapBlockStatus

11.2.3 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.

11.2.3.0.0.14 Field Documentation

11.2.3.0.0.14.1 uint16_t flash_swap_ifr_field_config_t::swapIndicatorAddress

11.2.3.0.0.14.2 uint16_t flash_swap_ifr_field_config_t::swapEnableWord

11.2.3.0.0.14.3 uint8_t flash_swap_ifr_field_config_t::reserved0[4]

11.2.4 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.

11.2.4.0.0.15 Field Documentation

11.2.4.0.0.15.1 uint32_t flash_swap_ifr_field_data_t::flashSwapIfrData[2]

11.2.4.0.0.15.2 flash_swap_ifr_field_config_t flash_swap_ifr_field_data_t::flashSwapIfrField

11.2.5 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

11.2.5.0.0.16 Field Documentation

11.2.5.0.0.16.1 `uint32_t pflash_protection_status_low_t::protl32b`

11.2.5.0.0.16.2 `uint8_t pflash_protection_status_low_t::protsl`

11.2.5.0.0.16.3 `uint8_t pflash_protection_status_low_t::protsh`

11.2.6 `struct pflash_protection_status_t`

Data Fields

- `pflash_protection_status_low_t valueLow32b`
PROT[31:0] or PROTS[15:0].

11.2.6.0.0.17 Field Documentation

11.2.6.0.0.17.1 `pflash_protection_status_low_t pflash_protection_status_t::valueLow32b`

11.2.7 `struct flash_prefetch_speculation_status_t`

Data Fields

- `flash_prefetch_speculation_option_t instructionOption`
Instruction speculation.
- `flash_prefetch_speculation_option_t dataOption`
Data speculation.

11.2.7.0.0.18 Field Documentation

11.2.7.0.0.18.1 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::instructionOption`

11.2.7.0.0.18.2 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::dataOption`

11.2.8 `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.

11.2.8.0.0.19 Field Documentation**11.2.8.0.0.19.1** uint32_t flash_protection_config_t::regionBase**11.2.8.0.0.19.2** uint32_t flash_protection_config_t::regionSize**11.2.8.0.0.19.3** uint32_t flash_protection_config_t::regionCount**11.2.9 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.

11.2.9.0.0.20 Field Documentation**11.2.9.0.0.20.1** uint32_t flash_access_config_t::SegmentBase**11.2.9.0.0.20.2** uint32_t flash_access_config_t::SegmentSize**11.2.9.0.0.20.3** uint32_t flash_access_config_t::SegmentCount**11.2.10 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 [sectorCmdAddressAligment](#)
An erase sector command address alignment.
- uint32_t [partCmdAddressAligment](#)
A program/verify part command address alignment.
- 32_t [resourceCmdAddressAligment](#)
A read resource command address alignment.
- uint32_t [checkCmdAddressAligment](#)
A program check command address alignment.

Data Structure Documentation

11.2.10.0.0.21 Field Documentation

11.2.10.0.0.21.1 uint32_t flash_operation_config_t::convertedAddress

11.2.10.0.0.21.2 uint32_t flash_operation_config_t::activeSectorSize

11.2.10.0.0.21.3 uint32_t flash_operation_config_t::activeBlockSize

11.2.10.0.0.21.4 uint32_t flash_operation_config_t::blockWriteUnitSize

11.2.10.0.0.21.5 uint32_t flash_operation_config_t::sectorCmdAddressAligment

11.2.10.0.0.21.6 uint32_t flash_operation_config_t::partCmdAddressAligment

11.2.10.0.0.21.7 uint32_t flash_operation_config_t::resourceCmdAddressAligment

11.2.10.0.0.21.8 uint32_t flash_operation_config_t::checkCmdAddressAligment

11.2.11 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 EEpromTotalSize`
For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM.

11.2.11.0.0.22 Field Documentation

11.2.11.0.0.22.1 `uint32_t flash_config_t::PFlashTotalSize`

11.2.11.0.0.22.2 `uint8_t flash_config_t::PFlashBlockCount`

11.2.11.0.0.22.3 `uint32_t flash_config_t::PFlashSectorSize`

11.2.11.0.0.22.4 `flash_callback_t flash_config_t::PFlashCallback`

11.2.11.0.0.22.5 `uint32_t flash_config_t::PFlashAccessSegmentSize`

11.2.11.0.0.22.6 `uint32_t flash_config_t::PFlashAccessSegmentCount`

11.2.11.0.0.22.7 `uint32_t* flash_config_t::flashExecuteInRamFunctionInfo`

11.2.11.0.0.22.8 `uint32_t flash_config_t::FlexRAMBlockBase`

For the non-FlexNVM device, this is the base address of the acceleration RAM memory

11.2.11.0.0.22.9 `uint32_t flash_config_t::FlexRAMTotalSize`

For the non-FlexNVM device, this is the size of the acceleration RAM memory

11.2.11.0.0.22.10 `uint32_t flash_config_t::DFlashBlockBase`

For the non-FlexNVM device, this field is unused

11.2.11.0.0.22.11 `uint32_t flash_config_t::DFlashTotalSize`

For the non-FlexNVM device, this field is unused

11.2.11.0.0.22.12 `uint32_t flash_config_t::EEpromTotalSize`

For the non-FlexNVM device, this field is unused

Enumeration Type Documentation

11.3 Macro Definition Documentation

11.3.1 #define MAKE_VERSION(*major*, *minor*, *bugfix*) (((major) << 16) | ((minor) << 8) | (bugfix))

11.3.2 #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))

Version 2.3.1.

11.3.3 #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Enables the FlexNVM support by default.

11.3.4 #define FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT 1

Enables the secondary flash support by default.

11.3.5 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for the flash resident application.

11.3.6 #define FLASH_DRIVER_IS_EXPORTED 0

Used for the KSDK application.

11.3.7 #define kStatusGroupGeneric 0

11.3.8 #define MAKE_STATUS(*group*, *code*) (((group)*100) + (code))

11.3.9 #define FOUR_CHAR_CODE(*a*, *b*, *c*, *d*) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))

11.4 Enumeration Type Documentation

11.4.1 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.

11.4.2 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.

11.4.3 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

11.4.4 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.

11.4.5 enum flash_security_state_t

Enumerator

kFLASH_SecurityStateNotSecure Flash is not secure.

kFLASH_SecurityStateBackdoorEnabled Flash backdoor is enabled.

kFLASH_SecurityStateBackdoorDisabled Flash backdoor is disabled.

11.4.6 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.

11.4.7 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.

11.4.8 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.

11.4.9 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.

11.4.10 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.

11.4.11 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.

11.4.12 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.

11.4.13 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.

11.4.14 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.

11.4.15 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.

11.4.16 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.

11.4.17 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.

11.4.18 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.

11.4.19 enum flash_memory_index_t

Enumerator

kFLASH_MemoryIndexPrimaryFlash Current flash memory is primary flash.
kFLASH_MemoryIndexSecondaryFlash Current flash memory is secondary flash.

11.4.20 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

11.4.21 enum flash_cache_clear_process_t

Enumerator

kFLASH_CacheClearProcessPre Pre flash cache clear process.

kFLASH_CacheClearProcessPost Post flash cache clear process.

11.5 Function Documentation

11.5.1 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_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_-PartitionStatusUpdate-Failure</i>	Failed to update the partition status.

11.5.2 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.

11.5.3 status_t FLASH_PrepareExecuteInRamFunctions (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.

11.5.4 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-PartitionStatusUpdate-Failure</i>	Failed to update the partition status.

11.5.5 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_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH-AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FLASH_Address-Error</i>	The address is out of range.

<i>kStatus_FLASH_Erase-KeyError</i>	The API erase key is invalid.
<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.

11.5.6 **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_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.
<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.

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.

11.5.7 **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.

11.5.8 **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.

11.5.9 **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.

11.5.10 **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.

11.5.11 **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 back-door 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.

11.5.12 **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.

11.5.13 **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.

11.5.14 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_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH-AlignmentError</i>	Parameter is not aligned with specified baseline.
<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.

11.5.15 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 programmed 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_Invalid-Argument</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.

11.5.16 **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.

11.5.17 `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.

11.5.18 **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.

11.5.19 **status_t FLASH_GetProperty (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.

11.5.20 **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.
---------------	--

Function Documentation

<i>whichProperty</i>	The desired property from the list of properties in enum flash_property_tag_t
<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.

11.5.21 status_t FLASH_PflashSetProtection (flash_config_t * *config*, pflash_protection_status_t * *protectStatus*)

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.

11.5.22 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_Invalid-Argument</i>	An invalid argument is provided.

Chapter 12

GPIO: General-Purpose Input/Output Driver

12.1 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.

12.2 Data Structure Documentation

12.2.1 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

12.3 Macro Definition Documentation

12.3.1 #define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))

12.4 Enumeration Type Documentation

12.4.1 enum gpio_pin_direction_t

Enumerator

kGPIO_DigitalInput Set current pin as digital input.

kGPIO_DigitalOutput Set current pin as digital output.

12.5 GPIO Driver

12.5.1 Overview

The MCUXpresso SDK provides a peripheral driver for the General-Purpose Input/Output (GPIO) module of MCUXpresso SDK devices.

12.5.2 Typical use case

12.5.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/* Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

12.5.2.2 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 Driver

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.

12.5.3 Function Documentation

12.5.3.1 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

12.5.3.2 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"> • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

12.5.3.3 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

12.5.3.4 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

12.5.3.5 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

12.5.3.6 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">• 0: corresponding pin input low-logic level.• 1: corresponding pin input high-logic level.
-------------	---

12.5.3.7 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.
------------	---

12.5.3.8 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

12.6 FGPIO Driver

12.6.1 Overview

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.

12.6.2 Typical use case

12.6.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
    kGpioDigitalOutput,
    1,
};
/* Sets the configuration */
FGPIO_PinInit(FGPIO_LED, LED_PINNUM, &led_config);
```

12.6.2.2 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 */
FGPIO_PinInit(FGPIO_SW1, SW1_PINNUM, &sw1_config);
```

FGPIO Configuration

- void **FGPIO_PinInit** (FGPIO_Type *base, uint32_t pin, const **gpio_pin_config_t** *config)
Initializes a FGPIO pin used by the board.

FGPIO Output Operations

- static void **FGPIO_WritePinOutput** (FGPIO_Type *base, uint32_t pin, uint8_t output)
Sets the output level of the multiple FGPIO pins to the logic 1 or 0.

FGPIO Driver

- static void [FGPIO_SetPinsOutput](#) (FGPIO_Type *base, uint32_t mask)
Sets the output level of the multiple FGPIO pins to the logic 1.
- static void [FGPIO_ClearPinsOutput](#) (FGPIO_Type *base, uint32_t mask)
Sets the output level of the multiple FGPIO pins to the logic 0.
- static void [FGPIO_TogglePinsOutput](#) (FGPIO_Type *base, uint32_t mask)
Reverses the current output logic of the multiple FGPIO pins.

FGPIO Input Operations

- static uint32_t [FGPIO_ReadPinInput](#) (FGPIO_Type *base, uint32_t pin)
Reads the current input value of the FGPIO port.

FGPIO Interrupt

- uint32_t [FGPIO_GetPinsInterruptFlags](#) (FGPIO_Type *base)
Reads the FGPIO port interrupt status flag.
- void [FGPIO_ClearPinsInterruptFlags](#) (FGPIO_Type *base, uint32_t mask)
Clears the multiple FGPIO pin interrupt status flag.

12.6.3 Function Documentation

12.6.3.1 void FGPIO_PinInit (FGPIO_Type * *base*, uint32_t *pin*, const gpio_pin_config_t * *config*)

To initialize the FGPIO driver, define a pin configuration, as either input or output, in the user file. Then, call the [FGPIO_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>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
<i>pin</i>	FGPIO port pin number
<i>config</i>	FGPIO pin configuration pointer

12.6.3.2 static void FGPIO_WritePinOutput (FGPIO_Type * *base*, uint32_t *pin*, uint8_t *output*) [inline], [static]

Parameters

<i>base</i>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
<i>pin</i>	FGPIO pin number
<i>output</i>	FGPIOpin output logic level. <ul style="list-style-type: none"> • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

12.6.3.3 static void FGPIO_SetPinsOutput (FGPIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
<i>mask</i>	FGPIO pin number macro

12.6.3.4 static void FGPIO_ClearPinsOutput (FGPIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
<i>mask</i>	FGPIO pin number macro

12.6.3.5 static void FGPIO_TogglePinsOutput (FGPIO_Type * *base*, uint32_t *mask*) [inline], [static]

FGPIO Driver

Parameters

<i>base</i>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
<i>mask</i>	FGPIO pin number macro

12.6.3.6 static uint32_t FGPIO_ReadPinInput (FGPIO_Type * *base*, uint32_t *pin*) [inline], [static]

Parameters

<i>base</i>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
<i>pin</i>	FGPIO pin number

Return values

<i>FGPIO</i>	port input value <ul style="list-style-type: none">• 0: corresponding pin input low-logic level.• 1: corresponding pin input high-logic level.
--------------	---

12.6.3.7 uint32_t FGPIO_GetPinsInterruptFlags (FGPIO_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>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
-------------	--

Return values

<i>The</i>	current FGPIO port interrupt status flags, for example, 0x00010001 means the pin 0 and 17 have the interrupt.
------------	---

12.6.3.8 void FGPIO_ClearPinsInterruptFlags (FGPIO_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	FGPIO peripheral base pointer (FGPIOA, FGPIOB, FGPIOC, and so on.)
<i>mask</i>	FGPIO pin number macro



Chapter 13

I2C: Inter-Integrated Circuit Driver

13.1 Overview

Modules

- [I2C DMA Driver](#)
- [I2C Driver](#)
- [I2C FreeRTOS Driver](#)
- [I2C eDMA Driver](#)

13.2 I2C Driver

13.2.1 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.

13.2.2 Typical use case

13.2.2.1 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;
}

```

13.2.2.2 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;

```

13.2.2.3 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;
```

13.2.2.4 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;
```

13.2.2.5 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

- *I2C master transfer structure. [More...](#)*
• 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_SlaveGeneralCallEvent = 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.

13.2.3 Data Structure Documentation

13.2.3.1 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.

I2C Driver

13.2.3.1.0.23 Field Documentation

13.2.3.1.0.23.1 bool i2c_master_config_t::enableMaster

13.2.3.1.0.23.2 bool i2c_master_config_t::enableStopHold

13.2.3.1.0.23.3 uint32_t i2c_master_config_t::baudRate_Bps

13.2.3.1.0.23.4 uint8_t i2c_master_config_t::glitchFilterWidth

13.2.3.2 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.

13.2.3.2.0.24 Field Documentation**13.2.3.2.0.24.1** `bool i2c_slave_config_t::enableSlave`**13.2.3.2.0.24.2** `bool i2c_slave_config_t::enableGeneralCall`**13.2.3.2.0.24.3** `bool i2c_slave_config_t::enableWakeUp`**13.2.3.2.0.24.4** `bool i2c_slave_config_t::enableBaudRateCtl`**13.2.3.2.0.24.5** `uint16_t i2c_slave_config_t::slaveAddress`**13.2.3.2.0.24.6** `uint16_t i2c_slave_config_t::upperAddress`**13.2.3.2.0.24.7** `i2c_slave_address_mode_t i2c_slave_config_t::addressingMode`**13.2.3.2.0.24.8** `uint32_t i2c_slave_config_t::sclStopHoldTime_ns`**13.2.3.3 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.

13.2.3.3.0.25 Field Documentation**13.2.3.3.0.25.1** `uint32_t i2c_master_transfer_t::flags`**13.2.3.3.0.25.2** `uint8_t i2c_master_transfer_t::slaveAddress`**13.2.3.3.0.25.3** `i2c_direction_t i2c_master_transfer_t::direction`**13.2.3.3.0.25.4** `uint32_t i2c_master_transfer_t::subaddress`

Transferred MSB first.

I2C Driver

13.2.3.3.0.25.5 `uint8_t i2c_master_transfer_t::subaddressSize`

13.2.3.3.0.25.6 `uint8_t* volatile i2c_master_transfer_t::data`

13.2.3.3.0.25.7 `volatile size_t i2c_master_transfer_t::dataSize`

13.2.3.4 `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.

13.2.3.4.0.26 Field Documentation

13.2.3.4.0.26.1 `i2c_master_transfer_t i2c_master_handle_t::transfer`

13.2.3.4.0.26.2 `size_t i2c_master_handle_t::transferSize`

13.2.3.4.0.26.3 `uint8_t i2c_master_handle_t::state`

13.2.3.4.0.26.4 `i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback`

13.2.3.4.0.26.5 `void* i2c_master_handle_t::userData`

13.2.3.5 `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.

13.2.3.5.0.27 Field Documentation**13.2.3.5.0.27.1** `i2c_slave_transfer_event_t i2c_slave_transfer_t::event`**13.2.3.5.0.27.2** `uint8_t* volatile i2c_slave_transfer_t::data`**13.2.3.5.0.27.3** `volatile size_t i2c_slave_transfer_t::dataSize`**13.2.3.5.0.27.4** `status_t i2c_slave_transfer_t::completionStatus`

Only applies for [kI2C_SlaveCompletionEvent](#).

13.2.3.5.0.27.5 `size_t i2c_slave_transfer_t::transferredCount`**13.2.3.6 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

13.2.3.6.0.28 Field Documentation

13.2.3.6.0.28.1 `volatile bool i2c_slave_handle_t::isBusy`

13.2.3.6.0.28.2 `i2c_slave_transfer_t i2c_slave_handle_t::transfer`

13.2.3.6.0.28.3 `uint32_t i2c_slave_handle_t::eventMask`

13.2.3.6.0.28.4 `i2c_slave_transfer_callback_t i2c_slave_handle_t::callback`

13.2.3.6.0.28.5 `void* i2c_slave_handle_t::userData`

13.2.4 Macro Definition Documentation

13.2.4.1 `#define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))`

13.2.5 Typedef Documentation

13.2.5.1 `typedef void(* i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *userData)`

13.2.5.2 `typedef void(* i2c_slave_transfer_callback_t)(I2C_Type *base, i2c_slave_transfer_t *xfer, void *userData)`

13.2.6 Enumeration Type Documentation

13.2.6.1 `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.

13.2.6.2 `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.

13.2.6.3 enum _i2c_interrupt_enable

Enumerator

kI2C_GlobalInterruptEnable I2C global interrupt.
kI2C_StartStopDetectInterruptEnable I2C start&stop detect interrupt.

13.2.6.4 enum i2c_direction_t

Enumerator

kI2C_Write Master transmits to the slave.
kI2C_Read Master receives from the slave.

13.2.6.5 enum i2c_slave_address_mode_t

Enumerator

kI2C_Address7bit 7-bit addressing mode.
kI2C_RangeMatch Range address match addressing mode.

13.2.6.6 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.

13.2.6.7 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.

13.2.7 Function Documentation

13.2.7.1 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

13.2.7.2 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

13.2.7.3 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
-------------	------------------

13.2.7.4 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
-------------	------------------

13.2.7.5 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.
---------------------	--

13.2.7.6 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.
--------------------	---

13.2.7.7 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.

13.2.7.8 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.

13.2.7.9 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.

13.2.7.10 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 <code>i2c_status_flag_t</code> . The parameter can be any combination of the following values: <ul style="list-style-type: none">• <code>kI2C_StartDetectFlag</code> (if available)• <code>kI2C_StopDetectFlag</code> (if available)• <code>kI2C_ArbitrationLostFlag</code>• <code>kI2C_IntPendingFlagFlag</code>

13.2.7.11 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 <code>i2c_status_flag_t</code> . The parameter can be any combination of the following values: <ul style="list-style-type: none">• <code>kI2C_StartDetectFlag</code> (if available)• <code>kI2C_StopDetectFlag</code> (if available)• <code>kI2C_ArbitrationLostFlag</code>• <code>kI2C_IntPendingFlagFlag</code>

13.2.7.12 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">• <code>kI2C_GlobalInterruptEnable</code>• <code>kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</code>• <code>kI2C_SdaTimeoutInterruptEnable</code>

13.2.7.13 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"> • kI2C_GlobalInterruptEnable • kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable • kI2C_SdaTimeoutInterruptEnable

13.2.7.14 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

13.2.7.15 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

13.2.7.16 void I2C_MasterSetBaudRate (I2C_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

Parameters

<i>base</i>	I2C base pointer
<i>baudRate_Bps</i>	the baud rate value in bps
<i>srcClock_Hz</i>	Source clock

13.2.7.17 **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.

13.2.7.18 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.

13.2.7.19 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.

13.2.7.20 status_t I2C_MasterWriteBlocking (I2C_Type * *base*, const uint8_t * *txBuff*, size_t *txSize*, uint32_t *flags*)

I2C Driver

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.

13.2.7.21 **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.

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.

13.2.7.22 **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.

13.2.7.23 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.

13.2.7.24 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

I2C Driver

<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.

13.2.7.25 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.
<i>userData</i>	user parameter passed to the callback function.

13.2.7.26 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 i2c_master_transfer_t 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.

13.2.7.27 `status_t I2C_MasterTransferGetCount (I2C_Type * base, i2c_master_handle_t * handle, size_t * count)`

I2C Driver

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.

13.2.7.28 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

13.2.7.29 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.

13.2.7.30 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 <code>i2c_slave_handle_t</code> structure to store the transfer state.
<i>callback</i>	pointer to user callback function.
<i>userData</i>	user parameter passed to the callback function.

13.2.7.31 `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 [kI2C_SlaveTransmitEvent](#) and [#kLPI2C_SlaveReceiveEvent](#) events are always enabled and do not need to 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 <code>#i2c_slave_handle_t</code> structure which stores the transfer state.
<i>eventMask</i>	Bit mask formed by OR'ing together i2c_slave_transfer_event_t 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 kI2C_SlaveAllEvents to enable all events.

Return values

<i>#kStatus_Success</i>	Slave transfers were successfully started.
<i>kStatus_I2C_Busy</i>	Slave transfers have already been started on this handle.

13.2.7.32 `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.

I2C Driver

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_slave_handle_t structure which stores the transfer state.

13.2.7.33 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.

13.2.7.34 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

13.3 I2C eDMA Driver

13.3.1 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.

13.3.2 Data Structure Documentation

13.3.2.1 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 eDMA Driver

- 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.

13.3.2.1.0.29 Field Documentation

13.3.2.1.0.29.1 `i2c_master_transfer_t i2c_master_edma_handle_t::transfer`

13.3.2.1.0.29.2 `size_t i2c_master_edma_handle_t::transferSize`

13.3.2.1.0.29.3 `uint8_t i2c_master_edma_handle_t::nbytes`

13.3.2.1.0.29.4 `uint8_t i2c_master_edma_handle_t::state`

13.3.2.1.0.29.5 `edma_handle_t* i2c_master_edma_handle_t::dmaHandle`

13.3.2.1.0.29.6 `i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completion-Callback`

13.3.2.1.0.29.7 `void* i2c_master_edma_handle_t::userData`

13.3.3 Typedef Documentation

13.3.3.1 `typedef void(* i2c_master_edma_transfer_callback_t)(I2C_Type *base, i2c_master_edma_handle_t *handle, status_t status, void *userData)`

13.3.4 Function Documentation

13.3.4.1 `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

<i>base</i>	I2C peripheral base address.
<i>handle</i>	A pointer to the <code>i2c_master_edma_handle_t</code> structure.
<i>callback</i>	A pointer to the user callback function.

<i>userData</i>	A user parameter passed to the callback function.
<i>edmaHandle</i>	eDMA handle pointer.

13.3.4.2 `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 <code>i2c_master_edma_handle_t</code> structure.
<i>xfer</i>	A pointer to the transfer structure of i2c_master_transfer_t .

Return values

<i>kStatus_Success</i>	Successfully 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.

13.3.4.3 `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 <code>i2c_master_edma_handle_t</code> structure.
<i>count</i>	A number of bytes transferred by the non-blocking transaction.

13.3.4.4 `void I2C_MasterTransferAbortEDMA (I2C_Type * base, i2c_master_edma_handle_t * handle)`

I2C eDMA Driver

Parameters

<i>base</i>	I2C peripheral base address.
<i>handle</i>	A pointer to the <code>i2c_master_edma_handle_t</code> structure.

13.4 I2C DMA Driver

13.4.1 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.

13.4.2 Data Structure Documentation

13.4.2.1 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](#)

I2C DMA Driver

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.

13.4.2.1.0.30 Field Documentation

13.4.2.1.0.30.1 `i2c_master_transfer_t i2c_master_dma_handle_t::transfer`

13.4.2.1.0.30.2 `size_t i2c_master_dma_handle_t::transferSize`

13.4.2.1.0.30.3 `uint8_t i2c_master_dma_handle_t::state`

13.4.2.1.0.30.4 `dma_handle_t* i2c_master_dma_handle_t::dmaHandle`

13.4.2.1.0.30.5 `i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completion-
Callback`

13.4.2.1.0.30.6 `void* i2c_master_dma_handle_t::userData`

13.4.3 Typedef Documentation

13.4.3.1 `typedef void(* i2c_master_dma_transfer_callback_t)(I2C_Type *base,
i2c_master_dma_handle_t *handle, status_t status, void *userData)`

13.4.4 Function Documentation

13.4.4.1 `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

13.4.4.2 `status_t I2C_MasterTransferDMA (I2C_Type * base, i2c_master_dma_handle_t
* handle, i2c_master_transfer_t * xfer)`

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	A pointer to the <code>i2c_master_dma_handle_t</code> structure
<i>xfer</i>	A pointer to the transfer structure of the i2c_master_transfer_t

Return values

<i>kStatus_Success</i>	Successfully 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.

13.4.4.3 `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 <code>i2c_master_dma_handle_t</code> structure
<i>count</i>	A number of bytes transferred so far by the non-blocking transaction.

13.4.4.4 `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 <code>i2c_master_dma_handle_t</code> structure.

13.5 I2C FreeRTOS Driver

13.5.1 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.

13.5.2 Function Documentation

13.5.2.1 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.

13.5.2.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t * *handle*)

This function deinitializes the I2C module and the related RTOS context.

Parameters

<i>handle</i>	The RTOS I2C handle.
---------------	----------------------

13.5.2.3 **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 14

LLWU: Low-Leakage Wakeup Unit Driver

14.1 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.

14.2 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.

14.3 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.

14.4 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.

Enumeration Type 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_EnableInternalModuleInterruptWakup** (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.

14.5 Data Structure Documentation

14.5.1 struct llwu_external_pin_filter_mode_t

Data Fields

- uint32_t **pinIndex**
A pin number.
- **llwu_pin_filter_mode_t** filterMode
Filter mode.

14.6 Macro Definition Documentation

14.6.1 #define FSL_LLWU_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

14.7 Enumeration Type Documentation

14.7.1 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.

14.7.2 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.

14.8 Function Documentation

14.8.1 void LLWU_SetExternalWakeupPinMode (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.

14.8.2 bool LLWU_GetExternalWakeupPinFlag (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.

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.

14.8.3 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.

14.8.4 static void LLWU_EnableInternalModuleInterruptWakeup (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

14.8.5 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.

14.8.6 void LLWU_SetPinFilterMode (LLWU_Type * *base*, uint32_t *filterIndex*, llwu_external_pin_filter_mode_t *filterMode*)

This function sets the pin filter configuration.

Function Documentation

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

14.8.7 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.

14.8.8 void LLWU_ClearPinFilterFlag (LLWU_Type * *base*, uint32_t *filterIndex*)

This function clears the pin filter flag.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>filterIndex</i>	A pin filter index to clear the flag, starting from 1.



Chapter 15

LPSCI: Universal Asynchronous Receiver/Transmitter

15.1 Overview

Modules

- [LPSCI DMA Driver](#)
- [LPSCI Driver](#)
- [LPSCI FreeRTOS Driver](#)

15.2 LPSCI Driver

15.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (LPSCI) module of MCUXpresso SDK devices.

The LPSCI driver can be split into 2 parts: functional APIs and transactional APIs.

Functional APIs are feature/property target low level APIs. Functional APIs can be used for the LPSCI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires knowledge of the LPSCI peripheral and how to organize functional APIs to meet the application requirements. All functional APIs use the peripheral base address as the first parameter. The LPSCI functional operation groups provide the functional APIs set.

The transactional APIs are transaction target high level APIs. Transactional APIs can be used to enable the peripheral quickly and also in the user's application if the code size and performance of transactional APIs can satisfy the user's requirements. If there are special requirements for the code size and performance, see the transactional API implementation and write custom code. All transactional APIs use the `lpsci_handle_t` as the second parameter. Initialize the handle by calling the [LPSCI_TransferCreateHandle\(\)](#) API.

Transactional APIs support queue feature for both transmit/receive. Whenever the user calls the [LPSCI_TransferSendNonBlocking\(\)](#) or [LPSCI_TransferReceiveNonBlocking\(\)](#), the transfer structure is queued into the internally maintained software queue. The driver automatically continues the transmit/receive if the queue is not empty. When a transfer is finished, the callback is called to inform the user about the completion.

The LPSCI transactional APIs support the background receive. Provide the ringbuffer address and size while calling the [LPSCI_TransferCreateHandle\(\)](#) API. The driver automatically starts receiving the data from the receive buffer into the ringbuffer. When the user makes subsequent calls to the `LPSCI_ReceiveDataIRQ()`, the driver provides the received data in the ringbuffer for user buffer directly and queues the left buffer into the receive queue.

15.2.2 Function groups

15.2.2.1 LPSCI functional Operation

This function group implements the LPSCI functional API. Functional APIs are feature-oriented.

15.2.2.2 LPSCI transactional Operation

This function group implements the LPSCI transactional API.

15.2.2.3 LPSCI transactional Operation

This function group implements the LPSCI DMA transactional API.

15.2.3 Typical use case

15.2.3.1 LPSCI Operation

```
uint8_t ch;
LPSCI_GetDefaultConfig(UART0,&user_config);
user_config.baudRate = 115200U;

LPSCI_Init(UART0,&user_config,120000000U);
LPSCI_EnableTx(UART0, true);
LPSCI_EnableRx(UART0, true);

LPSCI_WriteBlocking(UART0, txbuff, sizeof(txbuff)-1);

while(1)
{
    LPSCI_ReadBlocking(UART0,&ch, 1);
    LPSCI_WriteBlocking(UART0, &ch, 1);
}
```

15.2.3.2 LPSCI Send/Receive using an interrupt method

15.2.3.3 LPSCI Receive using the ringbuffer feature

15.2.3.4 LPSCI Send/Receive using the DMA method

Data Structures

- struct [lpsci_config_t](#)
LPSCI configure structure. [More...](#)
- struct [lpsci_transfer_t](#)
LPSCI transfer structure. [More...](#)

Driver version

- enum `_lpsci_status` {
 `kStatus_LPSCI_TxBusy` = MAKE_STATUS(kStatusGroup_LPSCI, 0),
 `kStatus_LPSCI_RxBusy` = MAKE_STATUS(kStatusGroup_LPSCI, 1),
 `kStatus_LPSCI_TxIdle` = MAKE_STATUS(kStatusGroup_LPSCI, 2),
 `kStatus_LPSCI_RxIdle` = MAKE_STATUS(kStatusGroup_LPSCI, 3),
 `kStatus_LPSCI_FlagCannotClearManually`,
 `kStatus_LPSCI_BaudrateNotSupport`,
 `kStatus_LPSCI_Error` = MAKE_STATUS(kStatusGroup_LPSCI, 6),
 `kStatus_LPSCI_RxRingBufferOverrun`,
 `kStatus_LPSCI_RxHardwareOverrun` = MAKE_STATUS(kStatusGroup_LPSCI, 8),
 `kStatus_LPSCI_NoiseError` = MAKE_STATUS(kStatusGroup_LPSCI, 9),
 `kStatus_LPSCI_FramingError` = MAKE_STATUS(kStatusGroup_LPSCI, 10),
 `kStatus_LPSCI_ParityError` = MAKE_STATUS(kStatusGroup_LPSCI, 11) }
 Error codes for the LPSCI driver.
- enum `lpsci_parity_mode_t` {
 `kLPSCI_ParityDisabled` = 0x0U,
 `kLPSCI_ParityEven` = 0x2U,
 `kLPSCI_ParityOdd` = 0x3U }
 LPSCI parity mode.
- enum `lpsci_stop_bit_count_t` {
 `kLPSCI_OneStopBit` = 0U,
 `kLPSCI_TwoStopBit` = 1U }
 LPSCI stop bit count.
- enum `_lpsci_interrupt_enable_t` {
 `kLPSCI_LinBreakInterruptEnable` = (UART0_BDH_LBKDIE_MASK),
 `kLPSCI_RxActiveEdgeInterruptEnable` = (UART0_BDH_RXEDGIE_MASK),
 `kLPSCI_TxDataRegEmptyInterruptEnable` = (UART0_C2_TIE_MASK << 8),
 `kLPSCI_TransmissionCompleteInterruptEnable` = (UART0_C2_TCIE_MASK << 8),
 `kLPSCI_RxDataRegFullInterruptEnable` = (UART0_C2_RIE_MASK << 8),
 `kLPSCI_IdleLineInterruptEnable` = (UART0_C2_ILIE_MASK << 8),
 `kLPSCI_RxOverrunInterruptEnable` = (UART0_C3_ORIE_MASK << 16),
 `kLPSCI_NoiseErrorInterruptEnable` = (UART0_C3_NEIE_MASK << 16),
 `kLPSCI_FramingErrorInterruptEnable` = (UART0_C3_FEIE_MASK << 16),
 `kLPSCI_ParityErrorInterruptEnable` = (UART0_C3_PEIE_MASK << 16) }
 LPSCI interrupt configuration structure, default settings all disabled.
- enum `_lpsci_status_flag_t` {


```

kLPSCI_TxDataRegEmptyFlag = (UART0_S1_TDRE_MASK),
kLPSCI_TransmissionCompleteFlag,
kLPSCI_RxDataRegFullFlag,
kLPSCI_IdleLineFlag = (UART0_S1_IDLE_MASK),
kLPSCI_RxOverrunFlag,
kLPSCI_NoiseErrorFlag = (UART0_S1_NF_MASK),
kLPSCI_FramingErrorFlag,
kLPSCI_ParityErrorFlag = (UART0_S1_PF_MASK),
kLPSCI_LinBreakFlag,
kLPSCI_RxActiveEdgeFlag,
kLPSCI_RxActiveFlag }
    LPSCI status flags.
• typedef void(* lpsci_transfer_callback_t )(UART0_Type *base, lpsci_handle_t *handle, status_t
    status, void *userData)
    LPSCI transfer callback function.
• #define FSL_LPSCI_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))
    LPSCI driver version 2.0.3.

```

Initialization and deinitialization

- status_t **LPSCI_Init** (UART0_Type *base, const lpsci_config_t *config, uint32_t srcClock_Hz)
Initializes an LPSCI instance with the user configuration structure and the peripheral clock.
- void **LPSCI_Deinit** (UART0_Type *base)
Deinitializes an LPSCI instance.
- void **LPSCI_GetDefaultConfig** (lpsci_config_t *config)
Gets the default configuration structure and saves the configuration to a user-provided pointer.
- status_t **LPSCI_SetBaudRate** (UART0_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
Sets the LPSCI instance baudrate.

Status

- uint32_t **LPSCI_GetStatusFlags** (UART0_Type *base)
Gets LPSCI status flags.
- status_t **LPSCI_ClearStatusFlags** (UART0_Type *base, uint32_t mask)

Interrupts

- void **LPSCI_EnableInterrupts** (UART0_Type *base, uint32_t mask)
Enables an LPSCI interrupt according to a provided mask.
- void **LPSCI_DisableInterrupts** (UART0_Type *base, uint32_t mask)
Disables the LPSCI interrupt according to a provided mask.
- uint32_t **LPSCI_GetEnabledInterrupts** (UART0_Type *base)
Gets the enabled LPSCI interrupts.

LPSCI Driver

DMA Control

- static uint32_t [LPSCI_GetDataRegisterAddress](#) (UART0_Type *base)
Gets the LPSCI data register address.
- static void [LPSCI_EnableTxDMA](#) (UART0_Type *base, bool enable)
Enables or disable LPSCI transmitter DMA request.
- static void [LPSCI_EnableRxDMA](#) (UART0_Type *base, bool enable)
Enables or disables the LPSCI receiver DMA.

Bus Operations

- static void [LPSCI_EnableTx](#) (UART0_Type *base, bool enable)
Enables or disables the LPSCI transmitter.
- static void [LPSCI_EnableRx](#) (UART0_Type *base, bool enable)
Enables or disables the LPSCI receiver.
- static void [LPSCI_WriteByte](#) (UART0_Type *base, uint8_t data)
Writes to the TX register.
- static uint8_t [LPSCI_ReadByte](#) (UART0_Type *base)
Reads the RX data register.
- void [LPSCI_WriteBlocking](#) (UART0_Type *base, const uint8_t *data, size_t length)
Writes to the TX register using a blocking method.
- status_t [LPSCI_ReadBlocking](#) (UART0_Type *base, uint8_t *data, size_t length)
Reads the RX register using a blocking method.

Transactional

- void [LPSCI_TransferCreateHandle](#) (UART0_Type *base, lpsci_handle_t *handle, [lpsci_transfer_callback_t](#) callback, void *userData)
Initializes the LPSCI handle.
- void [LPSCI_TransferStartRingBuffer](#) (UART0_Type *base, lpsci_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
Sets up the RX ring buffer.
- void [LPSCI_TransferStopRingBuffer](#) (UART0_Type *base, lpsci_handle_t *handle)
Aborts the background transfer and uninstalls the ring buffer.
- status_t [LPSCI_TransferSendNonBlocking](#) (UART0_Type *base, lpsci_handle_t *handle, [lpsci_transfer_t](#) *xfer)
Transmits a buffer of data using the interrupt method.
- void [LPSCI_TransferAbortSend](#) (UART0_Type *base, lpsci_handle_t *handle)
Aborts the interrupt-driven data transmit.
- status_t [LPSCI_TransferGetSendCount](#) (UART0_Type *base, lpsci_handle_t *handle, uint32_t *count)
Get the number of bytes that have been written to LPSCI TX register.
- status_t [LPSCI_TransferReceiveNonBlocking](#) (UART0_Type *base, lpsci_handle_t *handle, [lpsci_transfer_t](#) *xfer, size_t *receivedBytes)
Receives buffer of data using the interrupt method.
- void [LPSCI_TransferAbortReceive](#) (UART0_Type *base, lpsci_handle_t *handle)
Aborts interrupt driven data receiving.

- status_t [LPSCI_TransferGetReceiveCount](#) (UART0_Type *base, lpsci_handle_t *handle, uint32_t *count)
Get the number of bytes that have been received.
- void [LPSCI_TransferHandleIRQ](#) (UART0_Type *base, lpsci_handle_t *handle)
LPSCI IRQ handle function.
- void [LPSCI_TransferHandleErrorIRQ](#) (UART0_Type *base, lpsci_handle_t *handle)
LPSCI Error IRQ handle function.

15.2.4 Data Structure Documentation

15.2.4.1 struct lpsci_config_t

Data Fields

- uint32_t [baudRate_Bps](#)
LPSCI baud rate.
- lpsci_parity_mode_t [parityMode](#)
Parity mode, disabled (default), even, odd.
- lpsci_stop_bit_count_t [stopBitCount](#)
Number of stop bits, 1 stop bit (default) or 2 stop bits.
- bool [enableTx](#)
Enable TX.
- bool [enableRx](#)
Enable RX.

15.2.4.2 struct lpsci_transfer_t

Data Fields

- uint8_t * [data](#)
The buffer of data to be transfer.
- size_t [dataSize](#)
The byte count to be transfer.

LPSCI Driver

15.2.4.2.0.31 Field Documentation

15.2.4.2.0.31.1 `uint8_t* lpsci_transfer_t::data`

15.2.4.2.0.31.2 `size_t lpsci_transfer_t::dataSize`

15.2.5 Macro Definition Documentation

15.2.5.1 `#define FSL_LPSCI_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))`

15.2.6 Typedef Documentation

15.2.6.1 `typedef void(* lpsci_transfer_callback_t)(UART0_Type *base, lpsci_handle_t *handle, status_t status, void *userData)`

15.2.7 Enumeration Type Documentation

15.2.7.1 `enum _lpsci_status`

Enumerator

kStatus_LPSCI_TxBusy Transmitter is busy.
kStatus_LPSCI_RxBusy Receiver is busy.
kStatus_LPSCI_TxIdle Transmitter is idle.
kStatus_LPSCI_RxIdle Receiver is idle.
kStatus_LPSCI_FlagCannotClearManually Status flag can't be manually cleared.
kStatus_LPSCI_BaudrateNotSupport Baudrate is not support in current clock source.
kStatus_LPSCI_Error Error happens on LPSCI.
kStatus_LPSCI_RxRingBufferOverflow LPSCI RX software ring buffer overrun.
kStatus_LPSCI_RxHardwareOverflow LPSCI RX receiver overrun.
kStatus_LPSCI_NoiseError LPSCI noise error.
kStatus_LPSCI_FramingError LPSCI framing error.
kStatus_LPSCI_ParityError LPSCI parity error.

15.2.7.2 `enum lpsci_parity_mode_t`

Enumerator

kLPSCI_ParityDisabled Parity disabled.
kLPSCI_ParityEven Parity enabled, type even, bit setting: PE|PT = 10.
kLPSCI_ParityOdd Parity enabled, type odd, bit setting: PE|PT = 11.

15.2.7.3 enum lpsci_stop_bit_count_t

Enumerator

kLPSCI_OneStopBit One stop bit.
kLPSCI_TwoStopBit Two stop bits.

15.2.7.4 enum _lpsci_interrupt_enable_t

This structure contains the settings for all LPSCI interrupt configurations.

Enumerator

kLPSCI_LinBreakInterruptEnable LIN break detect interrupt.
kLPSCI_RxActiveEdgeInterruptEnable RX Active Edge interrupt.
kLPSCI_TxDataRegEmptyInterruptEnable Transmit data register empty interrupt.
kLPSCI_TransmissionCompleteInterruptEnable Transmission complete interrupt.
kLPSCI_RxDataRegFullInterruptEnable Receiver data register full interrupt.
kLPSCI_IdleLineInterruptEnable Idle line interrupt.
kLPSCI_RxOverrunInterruptEnable Receiver Overrun interrupt.
kLPSCI_NoiseErrorInterruptEnable Noise error flag interrupt.
kLPSCI_FramingErrorInterruptEnable Framing error flag interrupt.
kLPSCI_ParityErrorInterruptEnable Parity error flag interrupt.

15.2.7.5 enum _lpsci_status_flag_t

This provides constants for the LPSCI status flags for use in the LPSCI functions.

Enumerator

kLPSCI_TxDataRegEmptyFlag Tx data register empty flag, sets when Tx buffer is empty.
kLPSCI_TransmissionCompleteFlag Transmission complete flag, sets when transmission activity complete.
kLPSCI_RxDataRegFullFlag Rx data register full flag, sets when the receive data buffer is full.
kLPSCI_IdleLineFlag Idle line detect flag, sets when idle line detected.
kLPSCI_RxOverrunFlag Rx Overrun, sets when new data is received before data is read from receive register.
kLPSCI_NoiseErrorFlag Rx takes 3 samples of each received bit. If any of these samples differ, noise flag sets
kLPSCI_FramingErrorFlag Frame error flag, sets if logic 0 was detected where stop bit expected.
kLPSCI_ParityErrorFlag If parity enabled, sets upon parity error detection.
kLPSCI_LinBreakFlag LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.
kLPSCI_RxActiveEdgeFlag Rx pin active edge interrupt flag, sets when active edge detected.
kLPSCI_RxActiveFlag Receiver Active Flag (RAF), sets at beginning of valid start bit.

15.2.8 Function Documentation

15.2.8.1 **status_t LPSCI_Init (UART0_Type * *base*, const lpsci_config_t * *config*, uint32_t *srcClock_Hz*)**

This function configures the LPSCI module with user-defined settings. The user can configure the configuration structure and can also get the default configuration by calling the [LPSCI_GetDefaultConfig\(\)](#) function. Example below shows how to use this API to configure the LPSCI.

```
*  lpsci_config_t lpsciConfig;
*  lpsciConfig.baudRate_Bps = 115200U;
*  lpsciConfig.parityMode = kLPSCI_ParityDisabled;
*  lpsciConfig.stopBitCount = kLPSCI_OneStopBit;
*  LPSCI_Init(UART0, &lpsciConfig, 20000000U);
*
```

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>config</i>	Pointer to user-defined configuration structure.
<i>srcClock_Hz</i>	LPSCI clock source frequency in HZ.

Return values

<i>kStatus_LPSCI_BaudrateNotSupport</i>	Baudrate is not support in current clock source.
<i>kStatus_Success</i>	LPSCI initialize succeed

15.2.8.2 **void LPSCI_Deinit (UART0_Type * *base*)**

This function waits for TX complete, disables TX and RX, and disables the LPSCI clock.

Parameters

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

15.2.8.3 **void LPSCI_GetDefaultConfig (lpsci_config_t * *config*)**

This function initializes the LPSCI configure structure to default value. the default value are: lpsciConfig->baudRate_Bps = 115200U; lpsciConfig->parityMode = kLPSCI_ParityDisabled; lpsciConfig->stopBitCount = kLPSCI_OneStopBit; lpsciConfig->enableTx = false; lpsciConfig->enableRx = false;

Parameters

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

15.2.8.4 status_t LPSCI_SetBaudRate (UART0_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

This function configures the LPSCI module baudrate. This function is used to update the LPSCI module baudrate after the LPSCI module is initialized with the LPSCI_Init.

```
* LPSCI_SetBaudRate(UART0, 115200U, 200000000U);
*
```

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>baudRate_Bps</i>	LPSCI baudrate to be set.
<i>srcClock_Hz</i>	LPSCI clock source frequency in HZ.

Return values

<i>kStatus_LPSCI_-BaudrateNotSupport</i>	Baudrate is not supported in the current clock source.
<i>kStatus_Success</i>	Set baudrate succeed

15.2.8.5 uint32_t LPSCI_GetStatusFlags (UART0_Type * *base*)

This function gets all LPSCI status flags. The flags are returned as the logical OR value of the enumerators `_lpsci_flags`. To check a specific status, compare the return value to the enumerators in `_LPSCI_flags`. For example, to check whether the TX is empty:

```
* if (kLPSCI_TxDataRegEmptyFlag |
*    LPSCI_GetStatusFlags(UART0))
* {
*     ...
* }
*
```

LPSCI Driver

Parameters

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

Returns

LPSCI status flags which are ORed by the enumerators in the `_lpsci_flags`.

15.2.8.6 void LPSCI_EnableInterrupts (UART0_Type * *base*, uint32_t *mask*)

This function enables the LPSCI interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_lpsci_interrupt_enable`. For example, to enable the TX empty interrupt and RX full interrupt:

```
* LPSCI_EnableInterrupts (UART0,  
  kLPSCI_TxDataRegEmptyInterruptEnable |  
  kLPSCI_RxDataRegFullInterruptEnable);  
*
```

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of <code>_lpsci_interrupt_enable</code> .

15.2.8.7 void LPSCI_DisableInterrupts (UART0_Type * *base*, uint32_t *mask*)

This function disables the LPSCI interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_lpsci_interrupt_enable`. For example, to disable TX empty interrupt and RX full interrupt:

```
* LPSCI_DisableInterrupts (UART0,  
  kLPSCI_TxDataRegEmptyInterruptEnable |  
  kLPSCI_RxDataRegFullInterruptEnable);  
*
```

Parameters

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

<i>mask</i>	The interrupts to disable. Logical OR of <code>_LPSCI_interrupt_enable</code> .
-------------	---

15.2.8.8 `uint32_t LPSCI_GetEnabledInterrupts (UART0_Type * base)`

This function gets the enabled LPSCI interrupts, which are returned as the logical OR value of the enumerators `_lpsci_interrupt_enable`. To check a specific interrupts enable status, compare the return value to the enumerators in `_LPSCI_interrupt_enable`. For example, to check whether TX empty interrupt is enabled:

```
*  uint32_t enabledInterrupts = LPSCI_GetEnabledInterrupts(UART0);
*
*  if (kLPSCI_TxDataRegEmptyInterruptEnable & enabledInterrupts)
*  {
*      ...
*  }
*
```

Parameters

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

Returns

LPSCI interrupt flags which are logical OR of the enumerators in `_LPSCI_interrupt_enable`.

15.2.8.9 `static uint32_t LPSCI_GetDataRegisterAddress (UART0_Type * base)` `[inline], [static]`

This function returns the LPSCI data register address, which is mainly used by DMA/eDMA case.

Parameters

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

Returns

LPSCI data register address which are used both by transmitter and receiver.

15.2.8.10 `static void LPSCI_EnableTxDMA (UART0_Type * base, bool enable)` `[inline], [static]`

This function enables or disables the transmit data register empty flag, `S1[TDRE]`, to generate DMA requests.

LPSCI Driver

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>enable</i>	True to enable, false to disable.

15.2.8.11 static void LPSCI_EnableRxDMA (UART0_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>	LPSCI peripheral base address.
<i>enable</i>	True to enable, false to disable.

15.2.8.12 static void LPSCI_EnableTx (UART0_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the LPSCI transmitter.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>enable</i>	True to enable, false to disable.

15.2.8.13 static void LPSCI_EnableRx (UART0_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the LPSCI receiver.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>enable</i>	True to enable, false to disable.

15.2.8.14 static void LPSCI_WriteByte (UART0_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 before calling this function.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>data</i>	Data write to TX register.

15.2.8.15 static uint8_t LPSCI_ReadByte (UART0_Type * *base*) [inline], [static]

This function reads data from the RX register directly. The upper layer must ensure that the RX register is full before calling this function.

Parameters

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

Returns

Data read from RX data register.

15.2.8.16 void LPSCI_WriteBlocking (UART0_Type * *base*, const uint8_t * *data*, size_t *length*)

This function polls the TX register, waits for the TX register empty, and writes data to the TX buffer.

Note

This function does not check whether all the data has been sent out to bus, so before disable TX, check kLPSCI_TransmissionCompleteFlag to ensure the TX is finished.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>data</i>	Start address of the data to write.
<i>length</i>	Size of the data to write.

15.2.8.17 status_t LPSCI_ReadBlocking (UART0_Type * *base*, uint8_t * *data*, size_t *length*)

This function polls the RX register, waits for the RX register to be full, and reads data from the RX register.

LPSCI Driver

Parameters

<i>base</i>	LPSCI 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_LPSCI_Rx-HardwareOverrun</i>	Receiver overrun happened while receiving data.
<i>kStatus_LPSCI_Noise-Error</i>	Noise error happened while receiving data.
<i>kStatus_LPSCI_Framing-Error</i>	Framing error happened while receiving data.
<i>kStatus_LPSCI_Parity-Error</i>	Parity error happened while receiving data.
<i>kStatus_Success</i>	Successfully received all data.

15.2.8.18 void LPSCI_TransferCreateHandle (UART0_Type * *base*, lpsci_handle_t * *handle*, lpsci_transfer_callback_t *callback*, void * *userData*)

This function initializes the LPSCI handle, which can be used for other LPSCI transactional APIs. Usually, for a specified LPSCI instance, call this API once to get the initialized handle.

LPSCI driver supports the "background" receiving, which means that the user can set up an RX ring buffer optionally. Data received are stored into the ring buffer even when the user doesn't call the [LPSCI_TransferReceiveNonBlocking\(\)](#) API. If there is already data received in the ring buffer, get the received data from the ring buffer directly. The ring buffer is disabled if pass NULL as *ringBuffer*.

Parameters

<i>handle</i>	LPSCI handle pointer.
<i>base</i>	LPSCI peripheral base address.
<i>ringBuffer</i>	Start address of ring buffer for background receiving. Pass NULL to disable the ring buffer.

<i>ringBufferSize</i>	size of the ring buffer.
-----------------------	--------------------------

15.2.8.19 void LPSCI_TransferStartRingBuffer (UART0_Type * *base*, lpsci_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize*)

This function sets up the RX ring buffer to a specific LPSCI handle.

When the RX ring buffer is used, data received is stored into the ring buffer even when the user doesn't call the [LPSCI_TransferReceiveNonBlocking\(\)](#) API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

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>	LPSCI peripheral base address.
<i>handle</i>	LPSCI handle pointer.
<i>ringBuffer</i>	Start address of ring buffer for background receiving. Pass NULL to disable the ring buffer.
<i>ringBufferSize</i>	size of the ring buffer.

15.2.8.20 void LPSCI_TransferStopRingBuffer (UART0_Type * *base*, lpsci_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ringbuffer.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>handle</i>	LPSCI handle pointer.

15.2.8.21 status_t LPSCI_TransferSendNonBlocking (UART0_Type * *base*, lpsci_handle_t * *handle*, lpsci_transfer_t * *xfer*)

This function sends data using the 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 ISR, LPSCI driver calls the callback function and passes the [kStatus_LPSCI_TxIdle](#) as status parameter.

LPSCI Driver

Note

The `kStatus_LPSCI_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 `kLPSCI_TransmissionCompleteFlag` to ensure that the TX is complete.

Parameters

<i>handle</i>	LPSCI handle pointer.
<i>xfer</i>	LPSCI transfer structure, refer to <code>#LPSCI_transfer_t</code> .

Return values

<i>kStatus_Success</i>	Successfully start the data transmission.
<i>kStatus_LPSCI_TxBusy</i>	Previous transmission still not finished, data not all written to the TX register.
<i>kStatus_InvalidArgument</i>	Invalid argument.

15.2.8.22 void LPSCI_TransferAbortSend (UART0_Type * *base*, lpsci_handle_t * *handle*)

This function aborts the interrupt driven data send.

Parameters

<i>handle</i>	LPSCI handle pointer.
---------------	-----------------------

15.2.8.23 status_t LPSCI_TransferGetSendCount (UART0_Type * *base*, lpsci_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to LPSCI TX register by interrupt method.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>handle</i>	LPSCI handle pointer.

<i>count</i>	Send bytes count.
--------------	-------------------

Return values

<i>kStatus_NoTransferInProgress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <i>count</i> ;

15.2.8.24 **status_t LPSCI_TransferReceiveNonBlocking (UART0_Type * *base*, lpsci_handle_t * *handle*, lpsci_transfer_t * *xfer*, size_t * *receivedBytes*)**

This function receives data using the 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 ring buffer is copied and the parameter *receivedBytes* shows how many bytes are copied from the ring buffer. After copying, if the data in ring buffer is not enough to read, the receive request is saved by the LPSCI driver. When new data arrives, the receive request is serviced first. When all data is received, the LPSCI driver notifies the upper layer through a callback function and passes the status parameter [kStatus_LPSCI_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 the function returns with the parameter *receivedBytes* set to 5. For the remaining 5 bytes, newly arrived data is saved from the *xfer->data[5]*. When 5 bytes are received, the LPSCI 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>handle</i>	LPSCI handle pointer.
<i>xfer</i>	lpsci transfer structure. See lpsci_transfer_t .
<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_LPSCI_RxBusy</i>	Previous receive request is not finished.
<i>kStatus_InvalidArgument</i>	Invalid argument.

15.2.8.25 void LPSCI_TransferAbortReceive (UART0_Type * *base*, lpsci_handle_t * *handle*)

This function aborts interrupt driven data receiving.

Parameters

<i>handle</i>	LPSCI handle pointer.
---------------	-----------------------

15.2.8.26 **status_t LPSCI_TransferGetReceiveCount (UART0_Type * *base*, lpsci_handle_t * *handle*, uint32_t * *count*)**

This function gets the number of bytes that have been received.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>handle</i>	LPSCI 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> ;

15.2.8.27 **void LPSCI_TransferHandleIRQ (UART0_Type * *base*, lpsci_handle_t * *handle*)**

This function handles the LPSCI transmit and receive IRQ request.

Parameters

<i>handle</i>	LPSCI handle pointer.
---------------	-----------------------

15.2.8.28 **void LPSCI_TransferHandleErrorIRQ (UART0_Type * *base*, lpsci_handle_t * *handle*)**

This function handle the LPSCI error IRQ request.

LPSCI Driver

Parameters

<i>handle</i>	LPSCI handle pointer.
---------------	-----------------------

15.3 LPSCI DMA Driver

15.3.1 Overview

Data Structures

- struct [lpsci_dma_handle_t](#)
LPSCI DMA handle. [More...](#)

Typedefs

- typedef void(* [lpsci_dma_transfer_callback_t](#))(UART0_Type *base, lpsci_dma_handle_t *handle, status_t status, void *userData)
LPSCI transfer callback function.

eDMA transactional

- void [LPSCI_TransferCreateHandleDMA](#) (UART0_Type *base, lpsci_dma_handle_t *handle, [lpsci_dma_transfer_callback_t](#) callback, void *userData, [dma_handle_t](#) *txDmaHandle, [dma_handle_t](#) *rxDmaHandle)
Initializes the LPSCI handle which is used in transactional functions.
- status_t [LPSCI_TransferSendDMA](#) (UART0_Type *base, lpsci_dma_handle_t *handle, [lpsci_transfer_t](#) *xfer)
Sends data using DMA.
- status_t [LPSCI_TransferReceiveDMA](#) (UART0_Type *base, lpsci_dma_handle_t *handle, [lpsci_transfer_t](#) *xfer)
Receives data using DMA.
- void [LPSCI_TransferAbortSendDMA](#) (UART0_Type *base, lpsci_dma_handle_t *handle)
Aborts the sent data using DMA.
- void [LPSCI_TransferAbortReceiveDMA](#) (UART0_Type *base, lpsci_dma_handle_t *handle)
Aborts the receive data using DMA.
- status_t [LPSCI_TransferGetSendCountDMA](#) (UART0_Type *base, lpsci_dma_handle_t *handle, uint32_t *count)
Gets the number of bytes written to the LPSCI TX register.
- status_t [LPSCI_TransferGetReceiveCountDMA](#) (UART0_Type *base, lpsci_dma_handle_t *handle, uint32_t *count)
Gets the number of bytes that have been received.

15.3.2 Data Structure Documentation

15.3.2.1 struct _lpsci_dma_handle

Data Fields

- UART0_Type * [base](#)

LPSCI DMA Driver

- LPSCI peripheral base address.*
- [lpsci_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.

15.3.2.1.0.32 Field Documentation

15.3.2.1.0.32.1 **UART0_Type* lpsci_dma_handle_t::base**

15.3.2.1.0.32.2 **lpsci_dma_transfer_callback_t lpsci_dma_handle_t::callback**

15.3.2.1.0.32.3 **void* lpsci_dma_handle_t::userData**

15.3.2.1.0.32.4 **size_t lpsci_dma_handle_t::rxDataSizeAll**

15.3.2.1.0.32.5 **size_t lpsci_dma_handle_t::txDataSizeAll**

15.3.2.1.0.32.6 **dma_handle_t* lpsci_dma_handle_t::txDmaHandle**

15.3.2.1.0.32.7 **dma_handle_t* lpsci_dma_handle_t::rxDmaHandle**

15.3.2.1.0.32.8 **volatile uint8_t lpsci_dma_handle_t::txState**

15.3.3 Typedef Documentation

15.3.3.1 **typedef void(* lpsci_dma_transfer_callback_t)(UART0_Type *base, lpsci_dma_handle_t *handle, status_t status, void *userData)**

15.3.4 Function Documentation

15.3.4.1 **void LPSCI_TransferCreateHandleDMA (UART0_Type * base, lpsci_dma_handle_t * handle, lpsci_dma_transfer_callback_t callback, void * userData, dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle)**

Parameters

<i>handle</i>	Pointer to <code>lpsci_dma_handle_t</code> structure
<i>base</i>	LPSCI peripheral base address
<i>rxDmaHandle</i>	User requested DMA handle for RX DMA transfer
<i>txDmaHandle</i>	User requested DMA handle for TX DMA transfer

15.3.4.2 **status_t LPSCI_TransferSendDMA (UART0_Type * *base*, lpsci_dma_handle_t * *handle*, lpsci_transfer_t * *xfer*)**

This function sends data using DMA. This is a non-blocking function, which returns immediately. When all data is sent, the send callback function is called.

Parameters

<i>handle</i>	LPSCI handle pointer.
<i>xfer</i>	LPSCI DMA transfer structure, see lpsci_transfer_t .

Return values

<i>kStatus_Success</i>	if successful, others failed.
<i>kStatus_LPSCI_TxBusy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

15.3.4.3 **status_t LPSCI_TransferReceiveDMA (UART0_Type * *base*, lpsci_dma_handle_t * *handle*, lpsci_transfer_t * *xfer*)**

This function receives data using DMA. This is a non-blocking function, which returns immediately. When all data is received, the receive callback function is called.

Parameters

<i>handle</i>	Pointer to <code>lpsci_dma_handle_t</code> structure
<i>xfer</i>	LPSCI DMA transfer structure, see lpsci_transfer_t .

Return values

LPSCI DMA Driver

<i>kStatus_Success</i>	if successful, others failed.
<i>kStatus_LPSCI_RxBusy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

15.3.4.4 void LPSCI_TransferAbortSendDMA (UART0_Type * *base*, lpsci_dma_handle_t * *handle*)

This function aborts the sent data using DMA.

Parameters

<i>handle</i>	Pointer to lpsci_dma_handle_t structure.
---------------	--

15.3.4.5 void LPSCI_TransferAbortReceiveDMA (UART0_Type * *base*, lpsci_dma_handle_t * *handle*)

This function aborts the receive data using DMA.

Parameters

<i>handle</i>	Pointer to lpsci_dma_handle_t structure.
---------------	--

15.3.4.6 status_t LPSCI_TransferGetSendCountDMA (UART0_Type * *base*, lpsci_dma_handle_t * *handle*, uint32_t * *count*)

This function gets the number of bytes that have been written to the LPSCI TX register by DMA.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>handle</i>	LPSCI handle pointer.
<i>count</i>	Send bytes count.

Return values

<i>kStatus_NoTransferInProgress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

15.3.4.7 **status_t LPSCI_TransferGetReceiveCountDMA (UART0_Type * *base*, lpsci_dma_handle_t * *handle*, uint32_t * *count*)**

This function gets the number of bytes that have been received.

Parameters

<i>base</i>	LPSCI peripheral base address.
<i>handle</i>	LPSCI 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 <code>count</code> ;

15.4 LPSCI FreeRTOS Driver

15.4.1 Overview

Data Structures

- struct [lpsci_rtos_config_t](#)
LPSCI RTOS configuration structure. [More...](#)

LPSCI RTOS Operation

- int [LPSCI_RTOS_Init](#) (lpsci_rtos_handle_t *handle, lpsci_handle_t *t_handle, const [lpsci_rtos_config_t](#) *cfg)
Initializes an LPSCI instance for operation in RTOS.
- int [LPSCI_RTOS_Deinit](#) (lpsci_rtos_handle_t *handle)
Deinitializes an LPSCI instance for operation.

LPSCI transactional Operation

- int [LPSCI_RTOS_Send](#) (lpsci_rtos_handle_t *handle, const uint8_t *buffer, uint32_t length)
Send data in background.
- int [LPSCI_RTOS_Receive](#) (lpsci_rtos_handle_t *handle, uint8_t *buffer, uint32_t length, size_t *received)
Receives data.

15.4.2 Data Structure Documentation

15.4.2.1 struct lpsci_rtos_config_t

Data Fields

- UART0_Type * [base](#)
LPSCI base address.
- uint32_t [srcclk](#)
LPSCI source clock in Hz.
- uint32_t [baudrate](#)
Desired communication speed.
- [lpsci_parity_mode_t](#) [parity](#)
Parity setting.
- [lpsci_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.

15.4.3 Function Documentation

15.4.3.1 `int LPSCI_RTOS_Init (lpsci_rtos_handle_t * handle, lpsci_handle_t * t_handle,
const lpsci_rtos_config_t * cfg)`

LPSCI FreeRTOS Driver

Parameters

<i>handle</i>	The RTOS LPSCI handle, the pointer to allocated space for RTOS context.
<i>t_handle</i>	The pointer to allocated space where to store transactional layer internal state.
<i>cfg</i>	The pointer to the parameters required to configure the LPSCI after initialization.

Returns

0 succeed, others failed

15.4.3.2 int LPSCI_RTOS_Deinit (lpsci_rtos_handle_t * *handle*)

This function deinitializes the LPSCI module, set all register value to reset value and releases the resources.

Parameters

<i>handle</i>	The RTOS LPSCI handle.
---------------	------------------------

15.4.3.3 int LPSCI_RTOS_Send (lpsci_rtos_handle_t * *handle*, const uint8_t * *buffer*, uint32_t *length*)

This function sends data. It is synchronous API. If the HW buffer is full, the task is in the blocked state.

Parameters

<i>handle</i>	The RTOS LPSCI handle.
<i>buffer</i>	The pointer to buffer to send.
<i>length</i>	The number of bytes to send.

15.4.3.4 int LPSCI_RTOS_Receive (lpsci_rtos_handle_t * *handle*, uint8_t * *buffer*, uint32_t *length*, size_t * *received*)

It is synchronous API.

This function receives data from LPSCI. If any data is immediately available it is returned immediately and the number of bytes received.

Parameters

<i>handle</i>	The RTOS LPSCI handle.
<i>buffer</i>	The pointer to buffer where to write received data.
<i>length</i>	The number of bytes to receive.
<i>received</i>	The pointer to variable of size_t where the number of received data is filled.

Chapter 16

LPTMR: Low-Power Timer

16.1 Overview

The MCUXpresso SDK provides a driver for the Low-Power Timer (LPTMR) of MCUXpresso SDK devices.

16.2 Function groups

The LPTMR driver supports operating the module as a time counter or as a pulse counter.

16.2.1 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. 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.

16.2.2 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.

16.2.3 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

16.2.4 Status

Provides functions to get and clear the LPTMR status.

16.2.5 Interrupt

Provides functions to enable/disable LPTMR interrupts and get the currently enabled interrupts.

16.3 Typical use case

16.3.1 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` }
 - enum `lptmr_status_flags_t` { `kLPTMR_TimerCompareFlag` = `LPTMR_CSR_TCF_MASK` }
- List of the LPTMR interrupts.*
- List of the LPTMR status flags.*

Driver version

- #define `FSL_LPTMR_DRIVER_VERSION` (`MAKE_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.

16.4 Data Structure Documentation

16.4.1 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.

16.5 Enumeration Type Documentation

16.5.1 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.

16.5.2 enum lptmr_pin_polarity_t

Enumerator

kLPTMR_PinPolarityActiveHigh Pulse Counter input source is active-high.
kLPTMR_PinPolarityActiveLow Pulse Counter input source is active-low.

16.5.3 enum lptmr_timer_mode_t

Enumerator

kLPTMR_TimerModeTimeCounter Time Counter mode.
kLPTMR_TimerModePulseCounter Pulse Counter mode.

16.5.4 enum lptmr_prescaler_glitch_value_t

Enumerator

<i>kLPTMR_Prescale_Glitch_0</i>	Prescaler divide 2, glitch filter does not support this setting.
<i>kLPTMR_Prescale_Glitch_1</i>	Prescaler divide 4, glitch filter 2.
<i>kLPTMR_Prescale_Glitch_2</i>	Prescaler divide 8, glitch filter 4.
<i>kLPTMR_Prescale_Glitch_3</i>	Prescaler divide 16, glitch filter 8.
<i>kLPTMR_Prescale_Glitch_4</i>	Prescaler divide 32, glitch filter 16.
<i>kLPTMR_Prescale_Glitch_5</i>	Prescaler divide 64, glitch filter 32.
<i>kLPTMR_Prescale_Glitch_6</i>	Prescaler divide 128, glitch filter 64.
<i>kLPTMR_Prescale_Glitch_7</i>	Prescaler divide 256, glitch filter 128.
<i>kLPTMR_Prescale_Glitch_8</i>	Prescaler divide 512, glitch filter 256.
<i>kLPTMR_Prescale_Glitch_9</i>	Prescaler divide 1024, glitch filter 512.
<i>kLPTMR_Prescale_Glitch_10</i>	Prescaler divide 2048 glitch filter 1024.
<i>kLPTMR_Prescale_Glitch_11</i>	Prescaler divide 4096, glitch filter 2048.
<i>kLPTMR_Prescale_Glitch_12</i>	Prescaler divide 8192, glitch filter 4096.
<i>kLPTMR_Prescale_Glitch_13</i>	Prescaler divide 16384, glitch filter 8192.
<i>kLPTMR_Prescale_Glitch_14</i>	Prescaler divide 32768, glitch filter 16384.
<i>kLPTMR_Prescale_Glitch_15</i>	Prescaler divide 65536, glitch filter 32768.

16.5.5 enum lptmr_prescaler_clock_select_t

Note

Clock connections are SoC-specific

Enumerator

<i>kLPTMR_PrescalerClock_0</i>	Prescaler/glitch filter clock 0 selected.
<i>kLPTMR_PrescalerClock_1</i>	Prescaler/glitch filter clock 1 selected.
<i>kLPTMR_PrescalerClock_2</i>	Prescaler/glitch filter clock 2 selected.
<i>kLPTMR_PrescalerClock_3</i>	Prescaler/glitch filter clock 3 selected.

16.5.6 enum lptmr_interrupt_enable_t

Enumerator

<i>kLPTMR_TimerInterruptEnable</i>	Timer interrupt enable.
---	-------------------------

16.5.7 enum lptmr_status_flags_t

Enumerator

kLPTMR_TimerCompareFlag Timer compare flag.

16.6 Function Documentation

16.6.1 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.

16.6.2 void LPTMR_Deinit (LPTMR_Type * *base*)

Parameters

<i>base</i>	LPTMR peripheral base address
-------------	-------------------------------

16.6.3 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.
---------------	---

16.6.4 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 lptmr-_interrupt_enable_t

16.6.5 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 lptmr-_interrupt_enable_t .

16.6.6 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](#)

16.6.7 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](#)

16.6.8 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 lptmr_status_flags_t .

16.6.9 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.

16.6.10 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

16.6.11 `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
-------------	-------------------------------

16.6.12 `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 17

PIT: Periodic Interrupt Timer

17.1 Overview

The MCUXpresso SDK provides a driver for the Periodic Interrupt Timer (PIT) of MCUXpresso SDK devices.

17.2 Function groups

The PIT driver supports operating the module as a time counter.

17.2.1 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.

17.2.2 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.

17.2.3 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

17.2.4 Status

Provides functions to get and clear the PIT status.

17.2.5 Interrupt

Provides functions to enable/disable PIT interrupts and get current enabled interrupts.

17.3 Typical use case

17.3.1 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](#) }
List of PIT interrupts.
- enum [pit_status_flags_t](#) { [kPIT_TimerFlag](#) = [PIT_TFLG_TIF_MASK](#) }
List of PIT status flags.

Functions

- uint64_t [PIT_GetLifetimeTimerCount](#) (PIT_Type *base)
Reads the current lifetime counter value.

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)
Un-gates 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.

Enumeration Type Documentation

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.

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.

17.4 Data Structure Documentation

17.4.1 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

17.5 Enumeration Type Documentation

17.5.1 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.

17.5.2 enum pit_interrupt_enable_t

Enumerator

kPIT_TimerInterruptEnable Timer interrupt enable.

17.5.3 enum pit_status_flags_t

Enumerator

kPIT_TimerFlag Timer flag.

17.6 Function Documentation

17.6.1 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

17.6.2 void PIT_Deinit (PIT_Type * *base*)

Parameters

<i>base</i>	PIT peripheral base address
-------------	-----------------------------

17.6.3 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 onfiguration structure.
---------------	--

17.6.4 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.

17.6.5 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 pit_interrupt_enable_t

17.6.6 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 pit_interrupt_enable_t

17.6.7 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](#)

17.6.8 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](#)

17.6.9 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 pit_status_flags_t

17.6.10 **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

17.6.11 **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

17.6.12 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.

17.6.13 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.

17.6.14 uint64_t PIT_GetLifetimeTimerCount (PIT_Type * *base*)

The lifetime timer is a 64-bit timer which chains timer 0 and timer 1 together. Timer 0 and 1 are chained by calling the PIT_SetTimerChainMode before using this timer. The period of lifetime timer is equal to the "period of timer 0 * period of timer 1". For the 64-bit value, the higher 32-bit has the value of timer 1, and the lower 32-bit has the value of timer 0.

Function Documentation

Parameters

<i>base</i>	PIT peripheral base address
-------------	-----------------------------

Returns

Current lifetime timer value

Chapter 18

PMC: Power Management Controller

18.1 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.

18.2 Data Structure Documentation

18.2.1 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.

18.2.2 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.

18.2.3 struct [pmc_bandgap_buffer_config_t](#)

Data Fields

- bool [enable](#)
Enable bandgap buffer.
- bool [enableInLowPowerMode](#)

Enable bandgap buffer in low-power mode.

18.2.3.0.0.33 Field Documentation

18.2.3.0.0.33.1 `bool pmc_bandgap_buffer_config_t::enable`

18.2.3.0.0.33.2 `bool pmc_bandgap_buffer_config_t::enableInLowPowerMode`

18.3 Macro Definition Documentation

18.3.1 `#define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))`

Version 2.0.0.

18.4 Enumeration Type Documentation

18.4.1 `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)

18.4.2 `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)

18.5 Function Documentation

18.5.1 `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.

18.5.2 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

18.5.3 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.
-------------	------------------------------

18.5.4 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.

18.5.5 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.

18.5.6 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.
-------------	------------------------------

18.5.7 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

18.5.8 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.

18.5.9 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.
-------------	------------------------------

18.5.10 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.

Chapter 19

PORT: Port Control and Interrupts

19.1 Overview

The MCUXpresso SDK provides a driver for the Port Control and Interrupts (PORT) module of MCU-Xpresso SDK devices.

19.2 Typical configuration use case

19.2.1 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);
```

19.2.2 I2C PORT Configuration

```
/* I2C pin PORT configuration */
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_pin_config_t`
PORT pin configuration structure. [More...](#)

Enumerations

- enum `_port_pull` {
 `kPORT_PullDisable` = 0U,
 `kPORT_PullDown` = 2U,
 `kPORT_PullUp` = 3U }
}

Typical configuration use case

- Internal resistor pull feature selection.*
 - enum `_port_slew_rate` {
 `kPORT_FastSlewRate` = 0U,
 `kPORT_SlowSlewRate` = 1U }
- Slew rate selection.*
 - 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_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.*

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.

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.

19.3 Data Structure Documentation

19.3.1 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 [driveStrength](#): 1
Fast/slow drive strength configure.
- uint16_t [mux](#): 3
Pin mux Configure.

19.4 Macro Definition Documentation

19.4.1 #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))

19.5 Enumeration Type Documentation

19.5.1 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.

Enumeration Type Documentation

19.5.2 enum _port_slew_rate

Enumerator

- kPORT_FastSlewRate* Fast slew rate is configured.
- kPORT_SlowSlewRate* Slow slew rate is configured.

19.5.3 enum _port_passive_filter_enable

Enumerator

- kPORT_PassiveFilterDisable* Passive input filter is disabled.
- kPORT_PassiveFilterEnable* Passive input filter is enabled.

19.5.4 enum _port_drive_strength

Enumerator

- kPORT_LowDriveStrength* Low-drive strength is configured.
- kPORT_HighDriveStrength* High-drive strength is configured.

19.5.5 enum port_mux_t

Enumerator

- kPORT_PinDisabledOrAnalog* Corresponding pin is disabled, but is used as an analog pin.
- 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.

19.5.6 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.

19.6 Function Documentation

19.6.1 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.

19.6.2 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 ,
```

Function Documentation

```
*      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.

19.6.3 static void PORT_SetPinMux (PORT_Type * *base*, uint32_t *pin*, port_mux_t *mux*) [inline], [static]

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">• kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function.• kPORT_MuxAsGpio : Set 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. : 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 : kPORT_PinDisabledOrAnalog). This function is recommended to use to reset the pin mux

19.6.4 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>	PORT pin interrupt configuration. <ul style="list-style-type: none"> • kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled. • kPORT_DMARisingEdge: DMA request on rising edge(if the DMA requests exit). • kPORT_DMAFallingEdge: DMA request on falling edge(if the DMA requests exit). • kPORT_DMAEitherEdge: DMA request on either edge(if the DMA requests exit). • #kPORT_FlagRisingEdge: Flag sets on rising edge(if the Flag states exit). • #kPORT_FlagFallingEdge: Flag sets on falling edge(if the Flag states exit). • #kPORT_FlagEitherEdge: Flag sets on either edge(if the Flag states exit). • 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. • #kPORT_ActiveHighTriggerOutputEnable: Enable active high-trigger output (if the trigger states exit). • #kPORT_ActiveLowTriggerOutputEnable: Enable active low-trigger output (if the trigger states exit).

19.6.5 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.

19.6.6 `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.

Chapter 20

RCM: Reset Control Module Driver

20.1 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_SourceLockup` = RCM_SRS1_LOCKUP_MASK << 8U,
 `kRCM_SourceSw` = RCM_SRS1_SW_MASK << 8U,
 `kRCM_SourceMdma` = RCM_SRS1_MDM_AP_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)
 Gets the reset source status which caused a previous reset.

Enumeration Type Documentation

- void [RCM_ConfigureResetPinFilter](#) (RCM_Type *base, const [rcm_reset_pin_filter_config_t](#) *config)
Configures the reset pin filter.

20.2 Data Structure Documentation

20.2.1 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.

20.2.1.0.0.34 Field Documentation

20.2.1.0.0.34.1 bool [rcm_reset_pin_filter_config_t::enableFilterInStop](#)

20.2.1.0.0.34.2 [rcm_run_wait_filter_mode_t](#) [rcm_reset_pin_filter_config_t::filterInRunWait](#)

20.2.1.0.0.34.3 uint8_t [rcm_reset_pin_filter_config_t::busClockFilterCount](#)

20.3 Macro Definition Documentation

20.3.1 #define [FSL_RCM_DRIVER_VERSION](#) (MAKE_VERSION(2, 0, 1))

20.4 Enumeration Type Documentation

20.4.1 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_SourceLockup Core lock up reset.
kRCM_SourceSw Software reset.
kRCM_SourceMdmmap MDM-AP system reset.
kRCM_SourceSackerr Parameter could get all reset flags.

20.4.2 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.

20.5 Function Documentation

20.5.1 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.

20.5.2 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.

Chapter 21

RTC: Real Time Clock

21.1 Overview

The MCUXpresso SDK provides a driver for the Real Time Clock (RTC) of MCUXpresso SDK devices.

21.2 Function groups

The RTC driver supports operating the module as a time counter.

21.2.1 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.

21.2.2 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.

21.2.3 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

21.2.4 Start & Stop timer

The function `RTC_StartTimer()` starts the RTC time counter.

The function `RTC_StopTimer()` stops the RTC time counter.

21.2.5 Status

Provides functions to get and clear the RTC status.

21.2.6 Interrupt

Provides functions to enable/disable RTC interrupts and get current enabled interrupts.

21.2.7 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.

21.2.8 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.

21.3 Typical use case

21.3.1 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 */
```



```

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_TIIE_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)
Un gates 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.

21.4 Data Structure Documentation

21.4.1 struct rtc_datetime_t

Data Fields

- uint16_t [year](#)

Data Structure Documentation

- `uint8_t month`
Range from 1970 to 2099.
- `uint8_t day`
Range from 1 to 12.
- `uint8_t hour`
Range from 1 to 31 (depending on month).
- `uint8_t minute`
Range from 0 to 23.
- `uint8_t second`
Range from 0 to 59.

21.4.1.0.0.35 Field Documentation

21.4.1.0.0.35.1 `uint16_t rtc_datetime_t::year`

21.4.1.0.0.35.2 `uint8_t rtc_datetime_t::month`

21.4.1.0.0.35.3 `uint8_t rtc_datetime_t::day`

21.4.1.0.0.35.4 `uint8_t rtc_datetime_t::hour`

21.4.1.0.0.35.5 `uint8_t rtc_datetime_t::minute`

21.4.1.0.0.35.6 `uint8_t rtc_datetime_t::second`

21.4.2 `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.

21.5 Enumeration Type Documentation

21.5.1 enum rtc_interrupt_enable_t

Enumerator

kRTC_TimeInvalidInterruptEnable Time invalid interrupt.
kRTC_TimeOverflowInterruptEnable Time overflow interrupt.
kRTC_AlarmInterruptEnable Alarm interrupt.
kRTC_SecondsInterruptEnable Seconds interrupt.

21.5.2 enum rtc_status_flags_t

Enumerator

kRTC_TimeInvalidFlag Time invalid flag.
kRTC_TimeOverflowFlag Time overflow flag.
kRTC_AlarmFlag Alarm flag.

21.5.3 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

21.6 Function Documentation

21.6.1 void RTC_Init (RTC_Type * *base*, const rtc_config_t * *config*)

This function issues a software reset if the timer invalid flag is set.

Note

This API should be called at the beginning of the application using the RTC driver.

Function Documentation

Parameters

<i>base</i>	RTC peripheral base address
<i>config</i>	Pointer to the user's RTC configuration structure.

21.6.2 static void RTC_Deinit (RTC_Type * *base*) [inline], [static]

Parameters

<i>base</i>	RTC peripheral base address
-------------	-----------------------------

21.6.3 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.
---------------	--

21.6.4 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

21.6.5 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.

21.6.6 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

21.6.7 void RTC_GetAlarm (RTC_Type * *base*, rtc_datetime_t * *datetime*)

Parameters

Function Documentation

<i>base</i>	RTC peripheral base address
<i>datetime</i>	Pointer to the structure where the alarm date and time details are stored.

21.6.8 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 rtc_interrupt_enable_t

21.6.9 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 rtc_interrupt_enable_t

21.6.10 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](#)

21.6.11 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](#)

21.6.12 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 rtc_status_flags_t

21.6.13 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
-------------	-----------------------------

21.6.14 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
-------------	-----------------------------

21.6.15 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 rtc_osc_cap_load_t

21.6.16 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
-------------	-----------------------------

Chapter 22

SAI: Serial Audio Interface

22.1 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.

22.2 Typical use case

22.2.1 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)
{
}

// ...
}
```

22.2.2 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` }
- Master 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 }
- The SAI interrupt enable flag.*
 - enum `_sai_dma_enable_t` { `kSAI_FIFOWarningDMAEnable` = I2S_TCSR_FWDE_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_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)
Resets the SAI Tx.
- void `SAI_RxReset` (I2S_Type *base)
Resets the SAI Rx.
- void `SAI_TxEnable` (I2S_Type *base, bool enable)

Typical use case

- *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)
Sends data using a blocking method.
- static void [SAI_WriteData](#) (I2S_Type *base, uint32_t channel, uint32_t data)
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.
- static uint32_t [SAI_ReadData](#) (I2S_Type *base, uint32_t channel)
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.

22.3 Data Structure Documentation

22.3.1 struct sai_config_t

Data Fields

- [sai_protocol_t protocol](#)
Audio bus protocol in SAI.
- [sai_sync_mode_t syncMode](#)
SAI sync mode, control Tx/Rx clock sync.

Data Structure Documentation

- 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.

22.3.2 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 [channel](#)
Data channel used in transfer.
- [sai_protocol_t](#) [protocol](#)
Which audio protocol used.

22.3.2.0.0.36 Field Documentation

22.3.2.0.0.36.1 uint8_t [sai_transfer_format_t::channel](#)

22.3.3 struct [sai_transfer_t](#)

Data Fields

- uint8_t * [data](#)
Data start address to transfer.
- size_t [dataSize](#)
Transfer size.

22.3.3.0.0.37 Field Documentation

22.3.3.0.0.37.1 `uint8_t* sai_transfer_t::data`

22.3.3.0.0.37.2 `size_t sai_transfer_t::dataSize`

22.3.4 `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.

22.4 Macro Definition Documentation

22.4.1 `#define SAI_XFER_QUEUE_SIZE (4)`

22.5 Enumeration Type Documentation

22.5.1 `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

22.5.2 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.

22.5.3 enum sai_master_slave_t

Enumerator

kSAI_Master Master mode.
kSAI_Slave Slave mode.

22.5.4 enum sai_mono_stereo_t

Enumerator

kSAI_Stereo Stereo sound.
kSAI_MonoLeft Only left channel have sound.
kSAI_MonoRight Only Right channel have sound.

22.5.5 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.

22.5.6 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.

22.5.7 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.

22.5.8 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.

22.5.9 enum _sai_dma_enable_t

Enumerator

kSAI_FIFOWarningDMAEnable FIFO warning caused by the DMA request.

22.5.10 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_FIFOWarningFlag FIFO warning flag.

22.5.11 enum sai_reset_type_t

Enumerator

kSAI_ResetTypeSoftware Software reset, reset the logic state.
kSAI_ResetTypeFIFO FIFO reset, reset the FIFO read and write pointer.
kSAI_ResetAll All reset.

Function Documentation

22.5.12 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.

22.5.13 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.

22.6 Function Documentation

22.6.1 void SAI_TxInit (I2S_Type * *base*, const sai_config_t * *config*)

Un-gates 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.
---------------	------------------------------

22.6.2 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.

22.6.3 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
---------------	---

22.6.4 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
---------------	---

22.6.5 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
-------------	------------------

22.6.6 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
-------------	------------------

22.6.7 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
-------------	------------------

22.6.8 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.
---------------	--

22.6.9 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.

22.6.10 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.

22.6.11 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"> • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

22.6.12 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.

**22.6.13 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">• kSAI_WordStartFlag• kSAI_SyncErrorFlag• kSAI_FIFOErrorFlag

**22.6.14 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">• kSAI_WordStartInterruptEnable• kSAI_SyncErrorInterruptEnable• kSAI_FIFOWarningInterruptEnable• kSAI_FIFORequestInterruptEnable• kSAI_FIFOErrorInterruptEnable

**22.6.15 static void SAI_RxEnableInterrupts (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"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

22.6.16 static void SAI_TxDisableInterrupts (l2s_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"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

Function Documentation

22.6.17 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">• kSAI_WordStartInterruptEnable• kSAI_SyncErrorInterruptEnable• kSAI_FIFOWarningInterruptEnable• kSAI_FIFORequestInterruptEnable• kSAI_FIFOErrorInterruptEnable

22.6.18 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">• kSAI_FIFOWarningDMAEnable• kSAI_FIFORequestDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

22.6.19 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">• kSAI_FIFOWarningDMAEnable• kSAI_FIFORequestDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

22.6.20 `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.

22.6.21 `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.

22.6.22 `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.
---------------------------	---

22.6.23 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.

22.6.24 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.

22.6.25 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.

22.6.26 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.

**22.6.27 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.

22.6.28 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

22.6.29 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.

22.6.30 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.

22.6.31 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.

22.6.32 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.

22.6.33 `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.

22.6.34 `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_NoTransferInProgress</i>	There is not a non-blocking transaction currently in progress.

22.6.35 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_NoTransferInProgress</i>	There is not a non-blocking transaction currently in progress.

22.6.36 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.

22.6.37 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.

22.6.38 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.

22.6.39 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.

22.7 SAI DMA Driver

22.7.1 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.

22.7.2 Data Structure Documentation

22.7.2.1 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.

22.7.2.1.0.38 Field Documentation

22.7.2.1.0.38.1 [sai_transfer_t](#) [sai_dma_handle_t::saiQueue](#)[[SAI_XFER_QUEUE_SIZE](#)]

22.7.2.1.0.38.2 [volatile uint8_t](#) [sai_dma_handle_t::queueUser](#)

22.7.3 Function Documentation

22.7.3.1 **`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.

22.7.3.2 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.

22.7.3.3 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.

22.7.3.4 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.

22.7.3.5 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.

22.7.3.6 **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.

22.7.3.7 **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.

22.7.3.8 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.

22.7.3.9 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.

22.7.3.10 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_NoTransferInProgress</i>	There is not a non-blocking transaction currently in progress.

22.8 SAI eDMA Driver

22.8.1 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.

22.8.2 Data Structure Documentation

22.8.2.1 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.

22.8.2.1.0.39 Field Documentation

22.8.2.1.0.39.1 `uint8_t sai_edma_handle_t::nbytes`

22.8.2.1.0.39.2 `edma_tcd_t sai_edma_handle_t::tcd[SAI_XFER_QUEUE_SIZE+1U]`

22.8.2.1.0.39.3 `sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

22.8.2.1.0.39.4 `volatile uint8_t sai_edma_handle_t::queueUser`

22.8.3 Function Documentation

22.8.3.1 `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.

22.8.3.2 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.

22.8.3.3 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>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 argument is invalid.

22.8.3.4 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>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 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.

22.8.3.5 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.

22.8.3.6 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_GetReceiveRemaining-Bytes 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.

22.8.3.7 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.

22.8.3.8 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.

22.8.3.9 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.

22.8.3.10 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_NoTransferInProgress</i>	There is no non-blocking transaction in progress.

Chapter 23

SIM: System Integration Module Driver

23.1 Overview

The MCUXpresso SDK provides a peripheral driver for the System Integration Module (SIM) of MCU-Xpresso 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.

23.2 Data Structure Documentation

23.2.1 struct sim_uid_t

Data Fields

- uint32_t [MH](#)
UIDMH.
- uint32_t [ML](#)
UIDML.
- uint32_t [L](#)
UIDL.

23.2.1.0.0.40 Field Documentation

23.2.1.0.0.40.1 uint32_t sim_uid_t::MH

23.2.1.0.0.40.2 uint32_t sim_uid_t::ML

23.2.1.0.0.40.3 uint32_t sim_uid_t::L

23.3 Enumeration Type Documentation

23.3.1 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.

23.3.2 enum _sim_flash_mode

Enumerator

kSIM_FlashDisableInWait Disable flash in wait mode.

kSIM_FlashDisable Disable flash in normal mode.

23.4 Function Documentation

23.4.1 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_UsbVoltRegEnableInLow-Power);`

Function Documentation

Parameters

<i>mask</i>	USB voltage regulator enable setting.
-------------	---------------------------------------

23.4.2 void SIM_GetUniqueld (sim_uid_t * *uid*)

Parameters

<i>uid</i>	Pointer to the structure to save the UID value.
------------	---

23.4.3 static void SIM_SetFlashMode (uint8_t *mode*) [inline], [static]

Parameters

<i>mode</i>	The mode to set; see _sim_flash_mode for mode details.
-------------	--

Chapter 24

SLCD: Segment LCD Driver

24.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Segment LCD (SLCD) module of MCU-Xpresso SDK devices. The SLCD module is a CMOS charge pump voltage inverter that is designed for low voltage and low-power operation. SLCD is designed to generate the appropriate waveforms to drive multiplexed numeric, alphanumeric, or custom segment LCD panels. SLCD also has several timing and control settings that can be software-configured depending on the application's requirements. Timing and control consists of registers and control logic for the following:

1. LCD frame frequency
2. Duty cycle selection
3. Front plane/back plane selection and enabling
4. Blink modes and frequency
5. Operation in low-power modes

After the SLCD general initialization, the [SLCD_SetBackPlanePhase\(\)](#), [SLCD_SetFrontPlaneSegments\(\)](#), and [SLCD_SetFrontPlaneOnePhase\(\)](#) are used to set the special back/front Plane to make SLCD display correctly. Then, the independent display control APIs, [SLCD_StartDisplay\(\)](#) and [SLCD_StopDisplay\(\)](#), start and stop the SLCD display.

The [SLCD_StartBlinkMode\(\)](#) and [SLCD_StopBlinkMode\(\)](#) are provided for the runtime special blink mode control. To get the SLCD fault detection result, call the [SLCD_GetFaultDetectCounter\(\)](#).

24.2 Typical use case

24.2.1 SLCD Initialization operation

```
slcd_config_t configure = 0;
slcd_clock_config_t clkConfig =
{
    kSLCD_AlternateClk1,
    kSLCD_AltClkDivFactor1,
    kSLCD_ClkPrescaler00
#ifdef FSL_FEATURE_SLCD_HAS_FAST_FRAME_RATE
    ,
    false
#endif
};
SLCD_GetDefaultConfig(&configure);
configure.clkConfig      = &clkConfig;
configure.loadAdjust     = kSLCD_LowLoadOrIntermediateClkSrc;
configure.dutyCycle      = kSLCD_1Div4DutyCycle;
configure.slcdlowPinEnabled = 0x1a44;
configure.backPlaneLowPin = 0x0822;
configure.faultConfig    = NULL;

SLCD_Init(base, &configure);
```

Typical use case

```
SLCD_SetBackPlanePhase(base, 1, kSLCD_PhaseAActivate);
SLCD_SetBackPlanePhase(base, 5, kSLCD_PhaseBActivate);
SLCD_SetBackPlanePhase(base, 11, kSLCD_PhaseCActivate);

SLCD_SetFrontPlaneSegments(base, 0, (
    kSLCD_PhaseAActivate | kSLCD_PhaseBActivate));
SLCD_SetFrontPlaneSegments(base, 9, (
    kSLCD_PhaseBActivate | kSLCD_PhaseCActivate));

SLCD_StartDisplay(base);
```

Data Structures

- struct `slcd_fault_detect_config_t`
SLCD fault frame detection configuration structure. [More...](#)
- struct `slcd_clock_config_t`
SLCD clock configuration structure. [More...](#)
- struct `slcd_config_t`
SLCD configuration structure. [More...](#)

Enumerations

- enum `slcd_power_supply_option_t` {
 `kSLCD_InternalVil3UseChargePump`,
 `kSLCD_ExternalVil3UseResistorBiasNetwork`,
 `kSLCD_ExtetanlVil3UseChargePump`,
 `kSLCD_InternalVil1UseChargePump` }
SLCD power supply option.
- enum `slcd_regulated_voltage_trim_t` {
 `kSLCD_RegulatedVolatgeTrim00` = 0U,
 `kSLCD_RegulatedVolatgeTrim01`,
 `kSLCD_RegulatedVolatgeTrim02`,
 `kSLCD_RegulatedVolatgeTrim03`,
 `kSLCD_RegulatedVolatgeTrim04`,
 `kSLCD_RegulatedVolatgeTrim05`,
 `kSLCD_RegulatedVolatgeTrim06`,
 `kSLCD_RegulatedVolatgeTrim07`,
 `kSLCD_RegulatedVolatgeTrim08`,
 `kSLCD_RegulatedVolatgeTrim09`,
 `kSLCD_RegulatedVolatgeTrim10`,
 `kSLCD_RegulatedVolatgeTrim11`,
 `kSLCD_RegulatedVolatgeTrim12`,
 `kSLCD_RegulatedVolatgeTrim13`,
 `kSLCD_RegulatedVolatgeTrim14`,
 `kSLCD_RegulatedVolatgeTrim15` }
SLCD regulated voltage trim parameter, be used to meet the desired contrast.
- enum `slcd_load_adjust_t` {
 `kSLCD_LowLoadOrFastestClkSrc` = 0U,
 `kSLCD_LowLoadOrIntermediateClkSrc`,
 `kSLCD_HighLoadOrIntermediateClkSrc`,

`kSLCD_HighLoadOrSlowestClkSrc }`

SLCD load adjust to handle different LCD glass capacitance or configure the LCD charge pump clock source.

- enum `slcd_clock_src_t` {
`kSLCD_DefaultClk` = 0U,
`kSLCD_AlternateClk1` = 1U,
`kSLCD_AlternateClk2` = 3U }

SLCD clock source.

- enum `slcd_alt_clock_div_t` {
`kSLCD_AltClkDivFactor1` = 0U,
`kSLCD_AltClkDivFactor64`,
`kSLCD_AltClkDivFactor256`,
`kSLCD_AltClkDivFactor512` }

SLCD alternate clock divider.

- enum `slcd_clock_prescaler_t` {
`kSLCD_ClkPrescaler00` = 0U,
`kSLCD_ClkPrescaler01`,
`kSLCD_ClkPrescaler02`,
`kSLCD_ClkPrescaler03`,
`kSLCD_ClkPrescaler04`,
`kSLCD_ClkPrescaler05`,
`kSLCD_ClkPrescaler06`,
`kSLCD_ClkPrescaler07` }

SLCD clock prescaler to generate frame frequency.

- enum `slcd_duty_cycle_t` {
`kSLCD_1Div1DutyCycle` = 0U,
`kSLCD_1Div2DutyCycle`,
`kSLCD_1Div3DutyCycle`,
`kSLCD_1Div4DutyCycle`,
`kSLCD_1Div5DutyCycle`,
`kSLCD_1Div6DutyCycle`,
`kSLCD_1Div7DutyCycle`,
`kSLCD_1Div8DutyCycle` }

SLCD duty cycle.

- enum `slcd_phase_type_t` {
`kSLCD_NoPhaseActivate` = 0x00U,
`kSLCD_PhaseAActivate` = 0x01U,
`kSLCD_PhaseBActivate` = 0x02U,
`kSLCD_PhaseCActivate` = 0x04U,
`kSLCD_PhaseDActivate` = 0x08U,
`kSLCD_PhaseEActivate` = 0x10U,
`kSLCD_PhaseFActivate` = 0x20U,
`kSLCD_PhaseGActivate` = 0x40U,
`kSLCD_PhaseHActivate` = 0x80U }

SLCD segment phase type.

- enum `slcd_phase_index_t` {

Typical use case

```
kSLCD_PhaseAIndex = 0x0U,  
kSLCD_PhaseBIndex = 0x1U,  
kSLCD_PhaseCIndex = 0x2U,  
kSLCD_PhaseDIndex = 0x3U,  
kSLCD_PhaseEIndex = 0x4U,  
kSLCD_PhaseFIndex = 0x5U,  
kSLCD_PhaseGIndex = 0x6U,  
kSLCD_PhaseHIndex = 0x7U }
```

SLCD segment phase bit index.

- enum `slcd_display_mode_t` {
 kSLCD_NormalMode = 0U,
 kSLCD_AlternateMode,
 kSLCD_BlankMode }

SLCD display mode.

- enum `slcd_blink_mode_t` {
 kSLCD_BlankDisplayBlink = 0U,
 kSLCD_AltDisplayBlink }

SLCD blink mode.

- enum `slcd_blink_rate_t` {
 kSLCD_BlinkRate00 = 0U,
 kSLCD_BlinkRate01,
 kSLCD_BlinkRate02,
 kSLCD_BlinkRate03,
 kSLCD_BlinkRate04,
 kSLCD_BlinkRate05,
 kSLCD_BlinkRate06,
 kSLCD_BlinkRate07 }

SLCD blink rate.

- enum `slcd_fault_detect_clock_prescaler_t` {
 kSLCD_FaultSampleFreqDivider1 = 0U,
 kSLCD_FaultSampleFreqDivider2,
 kSLCD_FaultSampleFreqDivider4,
 kSLCD_FaultSampleFreqDivider8,
 kSLCD_FaultSampleFreqDivider16,
 kSLCD_FaultSampleFreqDivider32,
 kSLCD_FaultSampleFreqDivider64,
 kSLCD_FaultSampleFreqDivider128 }

SLCD fault detect clock prescaler.

- enum `slcd_fault_detect_sample_window_width_t` {
 kSLCD_FaultDetectWindowWidth4SampleClk = 0U,
 kSLCD_FaultDetectWindowWidth8SampleClk,
 kSLCD_FaultDetectWindowWidth16SampleClk,
 kSLCD_FaultDetectWindowWidth32SampleClk,
 kSLCD_FaultDetectWindowWidth64SampleClk,
 kSLCD_FaultDetectWindowWidth128SampleClk,
 kSLCD_FaultDetectWindowWidth256SampleClk,

`kSLCD_FaultDetectWindowWidth512SampleClk }`

SLCD fault detect sample window width.

- enum `slcd_interrupt_enable_t` { `kSLCD_FaultDetectCompleteInterrupt = 1U` }

SLCD interrupt source.

- enum `slcd_lowpower_behavior` {
`kSLCD_EnabledInWaitStop = 0,`
`kSLCD_EnabledInWaitOnly,`
`kSLCD_EnabledInStopOnly,`
`kSLCD_DisabledInWaitStop` }

SLCD behavior in low power mode.

Driver version

- #define `FSL_SLCD_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 1)`)
SLCD driver version 2.0.1.

Initialization and deinitialization

- void `SLCD_Init` (`LCD_Type *base`, `slcd_config_t *configure`)
Initializes the SLCD, ungates the module clock, initializes the power setting, enables all used plane pins, and sets with interrupt and work mode with the configuration.
- void `SLCD_Deinit` (`LCD_Type *base`)
Deinitializes the SLCD module, gates the module clock, disables an interrupt, and displays the SLCD.
- void `SLCD_GetDefaultConfig` (`slcd_config_t *configure`)
Gets the SLCD default configuration structure.

Plane Setting and Display Control

- static void `SLCD_StartDisplay` (`LCD_Type *base`)
Enables the SLCD controller, starts generation, and displays the front plane and back plane waveform.
- static void `SLCD_StopDisplay` (`LCD_Type *base`)
Stops the SLCD controller.
- void `SLCD_StartBlinkMode` (`LCD_Type *base`, `slcd_blink_mode_t mode`, `slcd_blink_rate_t rate`)
Starts the SLCD blink mode.
- static void `SLCD_StopBlinkMode` (`LCD_Type *base`)
Stops the SLCD blink mode.
- static void `SLCD_SetBackPlanePhase` (`LCD_Type *base`, `uint32_t pinIndx`, `slcd_phase_type_t phase`)
Sets the SLCD back plane pin phase.
- static void `SLCD_SetFrontPlaneSegments` (`LCD_Type *base`, `uint32_t pinIndx`, `uint8_t operation`)
Sets the SLCD front plane segment operation for a front plane pin.
- static void `SLCD_SetFrontPlaneOnePhase` (`LCD_Type *base`, `uint32_t pinIndx`, `slcd_phase_index_t phaseIndx`, `bool enable`)
Sets one SLCD front plane pin for one phase.
- static void `SLCD_EnablePadSafeState` (`LCD_Type *base`, `bool enable`)
Enables/disables the SLCD pad safe state.
- static `uint32_t SLCD_GetFaultDetectCounter` (`LCD_Type *base`)
Gets the SLCD fault detect counter.

Interrupts.

- void [SLCD_EnableInterrupts](#) (LCD_Type *base, uint32_t mask)
Enables the SLCD interrupt.
- void [SLCD_DisableInterrupts](#) (LCD_Type *base, uint32_t mask)
Disables the SLCD interrupt.
- uint32_t [SLCD_GetInterruptStatus](#) (LCD_Type *base)
Gets the SLCD interrupt status flag.
- void [SLCD_ClearInterruptStatus](#) (LCD_Type *base, uint32_t mask)
Clears the SLCD interrupt events status flag.

24.3 Data Structure Documentation

24.3.1 struct slcd_fault_detect_config_t

Data Fields

- bool [faultDetectIntEnable](#)
Fault frame detection interrupt enable flag.
- bool [faultDetectBackPlaneEnable](#)
True means the pin id fault detected is back plane otherwise front plane.
- uint8_t [faultDetectPinIndex](#)
Fault detected pin id from 0 to 63.
- [slcd_fault_detect_clock_prescaler_t](#) [faultPrescaler](#)
Fault detect clock prescaler.
- [slcd_fault_detect_sample_window_width_t](#) [width](#)
Fault detect sample window width.

24.3.1.0.0.41 Field Documentation

24.3.1.0.0.41.1 bool [slcd_fault_detect_config_t::faultDetectIntEnable](#)

24.3.1.0.0.41.2 bool [slcd_fault_detect_config_t::faultDetectBackPlaneEnable](#)

24.3.1.0.0.41.3 uint8_t [slcd_fault_detect_config_t::faultDetectPinIndex](#)

24.3.1.0.0.41.4 [slcd_fault_detect_clock_prescaler_t](#) [slcd_fault_detect_config_t::faultPrescaler](#)

24.3.1.0.0.41.5 [slcd_fault_detect_sample_window_width_t](#) [slcd_fault_detect_config_t::width](#)

24.3.2 struct slcd_clock_config_t

Data Fields

- [slcd_clock_src_t](#) [clkSource](#)
Clock source.
- [slcd_alt_clock_div_t](#) [altClkDivider](#)
The divider to divide the alternate clock used for alternate clock source.
- [slcd_clock_prescaler_t](#) [clkPrescaler](#)
Clock prescaler.

- bool [fastFrameRateEnable](#)
Fast frame rate enable flag.

24.3.2.0.0.42 Field Documentation

24.3.2.0.0.42.1 [slcd_clock_src_t](#) [slcd_clock_config_t::clkSource](#)

"slcd_clock_src_t" is recommended to be used. The SLCD is optimized to operate using a 32.768kHz clock input.

24.3.2.0.0.42.2 [slcd_alt_clock_div_t](#) [slcd_clock_config_t::altClkDivider](#)

24.3.2.0.0.42.3 [slcd_clock_prescaler_t](#) [slcd_clock_config_t::clkPrescaler](#)

24.3.2.0.0.42.4 bool [slcd_clock_config_t::fastFrameRateEnable](#)

24.3.3 struct [slcd_config_t](#)

Data Fields

- [slcd_power_supply_option_t](#) [powerSupply](#)
Power supply option.
- [slcd_regulated_voltage_trim_t](#) [voltageTrim](#)
Regulated voltage trim used for the internal regulator VIREG to adjust to facilitate contrast control.
- [slcd_clock_config_t](#) * [clkConfig](#)
Clock configure.
- [slcd_display_mode_t](#) [displayMode](#)
SLCD display mode.
- [slcd_load_adjust_t](#) [loadAdjust](#)
Load adjust to handle glass capacitance.
- [slcd_duty_cycle_t](#) [dutyCycle](#)
Duty cycle.
- [slcd_lowpower_behavior](#) [lowPowerBehavior](#)
SLCD behavior in low power mode.
- uint32_t [slcdLowPinEnabled](#)
Setting enabled SLCD pin 0 ~ pin 31.
- uint32_t [slcdHighPinEnabled](#)
Setting enabled SLCD pin 32 ~ pin 63.
- uint32_t [backPlaneLowPin](#)
Setting back plane pin 0 ~ pin 31.
- uint32_t [backPlaneHighPin](#)
Setting back plane pin 32 ~ pin 63.
- [slcd_fault_detect_config_t](#) * [faultConfig](#)
Fault frame detection configure.

Enumeration Type Documentation

24.3.3.0.0.43 Field Documentation

24.3.3.0.0.43.1 `slcd_power_supply_option_t slcd_config_t::powerSupply`

24.3.3.0.0.43.2 `slcd_regulated_voltage_trim_t slcd_config_t::voltageTrim`

24.3.3.0.0.43.3 `slcd_clock_config_t* slcd_config_t::clkConfig`

24.3.3.0.0.43.4 `slcd_display_mode_t slcd_config_t::displayMode`

24.3.3.0.0.43.5 `slcd_load_adjust_t slcd_config_t::loadAdjust`

24.3.3.0.0.43.6 `slcd_duty_cycle_t slcd_config_t::dutyCycle`

24.3.3.0.0.43.7 `slcd_lowpower_behavior slcd_config_t::lowPowerBehavior`

24.3.3.0.0.43.8 `uint32_t slcd_config_t::slcdLowPinEnabled`

Setting bit *n* to 1 means enable pin *n*.

24.3.3.0.0.43.9 `uint32_t slcd_config_t::slcdHighPinEnabled`

Setting bit *n* to 1 means enable pin (*n* + 32).

24.3.3.0.0.43.10 `uint32_t slcd_config_t::backPlaneLowPin`

Setting bit *n* to 1 means setting pin *n* as back plane. It should never have the same bit setting as the frontPlane Pin.

24.3.3.0.0.43.11 `uint32_t slcd_config_t::backPlaneHighPin`

Setting bit *n* to 1 means setting pin (*n* + 32) as back plane. It should never have the same bit setting as the frontPlane Pin.

24.3.3.0.0.43.12 `slcd_fault_detect_config_t* slcd_config_t::faultConfig`

If not requirement, set to NULL.

24.4 Macro Definition Documentation

24.4.1 `#define FSL_SLCD_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

24.5 Enumeration Type Documentation

24.5.1 `enum slcd_power_supply_option_t`

Enumerator

kSLCD_InternalVll3UseChargePump VLL3 connected to VDD internally, charge pump is used to generate VLL1 and VLL2.

kSLCD_ExternalVll3UseResistorBiasNetwork VLL3 is driven externally and resistor bias network is used to generate VLL1 and VLL2.

kSLCD_ExteranlVll3UseChargePump VLL3 is driven externally and charge pump is used to generate VLL1 and VLL2.

kSLCD_InternalVll1UseChargePump VIREG is connected to VLL1 internally and charge pump is used to generate VLL2 and VLL3.

24.5.2 enum slcd_regulated_voltage_trim_t

Enumerator

<i>kSLCD_RegulatedVolatgeTrim00</i>	Increase the voltage to 0.91 V.
<i>kSLCD_RegulatedVolatgeTrim01</i>	Increase the voltage to 1.01 V.
<i>kSLCD_RegulatedVolatgeTrim02</i>	Increase the voltage to 0.96 V.
<i>kSLCD_RegulatedVolatgeTrim03</i>	Increase the voltage to 1.06 V.
<i>kSLCD_RegulatedVolatgeTrim04</i>	Increase the voltage to 0.93 V.
<i>kSLCD_RegulatedVolatgeTrim05</i>	Increase the voltage to 1.02 V.
<i>kSLCD_RegulatedVolatgeTrim06</i>	Increase the voltage to 0.98 V.
<i>kSLCD_RegulatedVolatgeTrim07</i>	Increase the voltage to 1.08 V.
<i>kSLCD_RegulatedVolatgeTrim08</i>	Increase the voltage to 0.92 V.
<i>kSLCD_RegulatedVolatgeTrim09</i>	Increase the voltage to 1.02 V.
<i>kSLCD_RegulatedVolatgeTrim10</i>	Increase the voltage to 0.97 V.
<i>kSLCD_RegulatedVolatgeTrim11</i>	Increase the voltage to 1.07 V.
<i>kSLCD_RegulatedVolatgeTrim12</i>	Increase the voltage to 0.94 V.
<i>kSLCD_RegulatedVolatgeTrim13</i>	Increase the voltage to 1.05 V.
<i>kSLCD_RegulatedVolatgeTrim14</i>	Increase the voltage to 0.99 V.
<i>kSLCD_RegulatedVolatgeTrim15</i>	Increase the voltage to 1.09 V.

24.5.3 enum slcd_load_adjust_t

Adjust the LCD glass capacitance if resistor bias network is enabled: ***kSLCD_LowLoadOrFastestClkSrc*** - Low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) ***kSLCD_LowLoadOrIntermediateClkSrc*** - low load (LCD glass capacitance 2000pF or lower. LCD or GPIO function can be used on VLL1,VLL2,Vcap1 and Vcap2 pins) ***kSLCD_HighLoadOrIntermediateClkSrc*** - high load (LCD glass capacitance 8000pF or lower. LCD or GPIO function can be used on Vcap1 and Vcap2 pins) ***kSLCD_HighLoadOrSlowestClkSrc*** - high load (LCD glass capacitance 8000pF or lower LCD or GPIO function can be used on Vcap1 and Vcap2 pins) Adjust clock for charge pump if charge pump is enabled: ***kSLCD_LowLoadOrFastestClkSrc*** - Fasten clock source (LCD glass capacitance 8000pF or 4000pF or lower if Fast Frame Rate is set) ***kSLCD_LowLoadOrIntermediateClkSrc*** - Intermediate clock source (LCD glass capacitance 4000pF or 2000pF or lower if Fast Frame Rate is set) ***kSLCD_HighLoadOrIntermediateClkSrc*** - Intermediate clock source

Enumeration Type Documentation

(LCD glass capacitance 2000pF or 1000pF or lower if Fast Frame Rate is set) *kSLCD_HighLoadOrSlowestClkSrc* - slowest clock source (LCD glass capacitance 1000pF or 500pF or lower if Fast Frame Rate is set)

Enumerator

kSLCD_LowLoadOrFastestClkSrc Adjust in low load or selects fastest clock.
kSLCD_LowLoadOrIntermediateClkSrc Adjust in low load or selects intermediate clock.
kSLCD_HighLoadOrIntermediateClkSrc Adjust in high load or selects intermediate clock.
kSLCD_HighLoadOrSlowestClkSrc Adjust in high load or selects slowest clock.

24.5.4 enum slcd_clock_src_t

Enumerator

kSLCD_DefaultClk Select default clock ERCLK32K.
kSLCD_AlternateClk1 Select alternate clock source 1 : MCGIRCLK.
kSLCD_AlternateClk2 Select alternate clock source 2 : OSCERCLK.

24.5.5 enum slcd_alt_clock_div_t

Enumerator

kSLCD_AltClkDivFactor1 No divide for alternate clock.
kSLCD_AltClkDivFactor64 Divide alternate clock with factor 64.
kSLCD_AltClkDivFactor256 Divide alternate clock with factor 256.
kSLCD_AltClkDivFactor512 Divide alternate clock with factor 512.

24.5.6 enum slcd_clock_prescaler_t

Enumerator

kSLCD_ClkPrescaler00 Prescaler 0.
kSLCD_ClkPrescaler01 Prescaler 1.
kSLCD_ClkPrescaler02 Prescaler 2.
kSLCD_ClkPrescaler03 Prescaler 3.
kSLCD_ClkPrescaler04 Prescaler 4.
kSLCD_ClkPrescaler05 Prescaler 5.
kSLCD_ClkPrescaler06 Prescaler 6.
kSLCD_ClkPrescaler07 Prescaler 7.

24.5.7 enum slcd_duty_cycle_t

Enumerator

kSLCD_1Div1DutyCycle LCD use 1 BP 1/1 duty cycle.
kSLCD_1Div2DutyCycle LCD use 2 BP 1/2 duty cycle.
kSLCD_1Div3DutyCycle LCD use 3 BP 1/3 duty cycle.
kSLCD_1Div4DutyCycle LCD use 4 BP 1/4 duty cycle.
kSLCD_1Div5DutyCycle LCD use 5 BP 1/5 duty cycle.
kSLCD_1Div6DutyCycle LCD use 6 BP 1/6 duty cycle.
kSLCD_1Div7DutyCycle LCD use 7 BP 1/7 duty cycle.
kSLCD_1Div8DutyCycle LCD use 8 BP 1/8 duty cycle.

24.5.8 enum slcd_phase_type_t

Enumerator

kSLCD_NoPhaseActivate LCD waveform no phase activates.
kSLCD_PhaseAActivate LCD waveform phase A activates.
kSLCD_PhaseBActivate LCD waveform phase B activates.
kSLCD_PhaseCAActivate LCD waveform phase C activates.
kSLCD_PhaseDAActivate LCD waveform phase D activates.
kSLCD_PhaseEActivate LCD waveform phase E activates.
kSLCD_PhaseFAActivate LCD waveform phase F activates.
kSLCD_PhaseGAActivate LCD waveform phase G activates.
kSLCD_PhaseHActivate LCD waveform phase H activates.

24.5.9 enum slcd_phase_index_t

Enumerator

kSLCD_PhaseAIndex LCD phase A bit index.
kSLCD_PhaseBIndex LCD phase B bit index.
kSLCD_PhaseCIndex LCD phase C bit index.
kSLCD_PhaseDIndex LCD phase D bit index.
kSLCD_PhaseEIndex LCD phase E bit index.
kSLCD_PhaseFIndex LCD phase F bit index.
kSLCD_PhaseGIndex LCD phase G bit index.
kSLCD_PhaseHIndex LCD phase H bit index.

24.5.10 enum slcd_display_mode_t

Enumerator

kSLCD_NormalMode LCD Normal display mode.

kSLCD_AlternateMode LCD Alternate display mode. For four back planes or less.

kSLCD_BlankMode LCD Blank display mode.

24.5.11 enum slcd_blink_mode_t

Enumerator

kSLCD_BlankDisplayBlink Display blank during the blink period.

kSLCD_AltDisplayBlink Display alternate display during the blink period if duty cycle is lower than 5.

24.5.12 enum slcd_blink_rate_t

Enumerator

kSLCD_BlinkRate00 SLCD blink rate is LCD clock/ $((2^{12}))$.

kSLCD_BlinkRate01 SLCD blink rate is LCD clock/ $((2^{13}))$.

kSLCD_BlinkRate02 SLCD blink rate is LCD clock/ $((2^{14}))$.

kSLCD_BlinkRate03 SLCD blink rate is LCD clock/ $((2^{15}))$.

kSLCD_BlinkRate04 SLCD blink rate is LCD clock/ $((2^{16}))$.

kSLCD_BlinkRate05 SLCD blink rate is LCD clock/ $((2^{17}))$.

kSLCD_BlinkRate06 SLCD blink rate is LCD clock/ $((2^{18}))$.

kSLCD_BlinkRate07 SLCD blink rate is LCD clock/ $((2^{19}))$.

24.5.13 enum slcd_fault_detect_clock_prescaler_t

Enumerator

kSLCD_FaultSampleFreqDivider1 Fault detect sample clock frequency is 1/1 bus clock.

kSLCD_FaultSampleFreqDivider2 Fault detect sample clock frequency is 1/2 bus clock.

kSLCD_FaultSampleFreqDivider4 Fault detect sample clock frequency is 1/4 bus clock.

kSLCD_FaultSampleFreqDivider8 Fault detect sample clock frequency is 1/8 bus clock.

kSLCD_FaultSampleFreqDivider16 Fault detect sample clock frequency is 1/16 bus clock.

kSLCD_FaultSampleFreqDivider32 Fault detect sample clock frequency is 1/32 bus clock.

kSLCD_FaultSampleFreqDivider64 Fault detect sample clock frequency is 1/64 bus clock.

kSLCD_FaultSampleFreqDivider128 Fault detect sample clock frequency is 1/128 bus clock.

24.5.14 enum slcd_fault_detect_sample_window_width_t

Enumerator

- kSLCD_FaultDetectWindowWidth4SampleClk* Sample window width is 4 sample clock cycles.
- kSLCD_FaultDetectWindowWidth8SampleClk* Sample window width is 8 sample clock cycles.
- kSLCD_FaultDetectWindowWidth16SampleClk* Sample window width is 16 sample clock cycles.
- kSLCD_FaultDetectWindowWidth32SampleClk* Sample window width is 32 sample clock cycles.
- kSLCD_FaultDetectWindowWidth64SampleClk* Sample window width is 64 sample clock cycles.
- kSLCD_FaultDetectWindowWidth128SampleClk* Sample window width is 128 sample clock cycles.
- kSLCD_FaultDetectWindowWidth256SampleClk* Sample window width is 256 sample clock cycles.
- kSLCD_FaultDetectWindowWidth512SampleClk* Sample window width is 512 sample clock cycles.

24.5.15 enum slcd_interrupt_enable_t

Enumerator

- kSLCD_FaultDetectCompleteInterrupt* SLCD fault detection complete interrupt source.

24.5.16 enum slcd_lowpower_behavior

Enumerator

- kSLCD_EnabledInWaitStop* SLCD works in wait and stop mode.
- kSLCD_EnabledInWaitOnly* SLCD works in wait mode and is disabled in stop mode.
- kSLCD_EnabledInStopOnly* SLCD works in stop mode and is disabled in wait mode.
- kSLCD_DisabledInWaitStop* SLCD is disabled in stop mode and wait mode.

24.6 Function Documentation

24.6.1 void SLCD_Init (LCD_Type * *base*, slcd_config_t * *configure*)

Parameters

Function Documentation

<i>base</i>	SLCD peripheral base address.
<i>configure</i>	SLCD configuration pointer. For the configuration structure, many parameters have the default setting and the <code>SLCD_Getdefaultconfig()</code> is provided to get them. Use it verified for their applications. The others have no default settings, such as "clk-Config", and must be provided by the application before calling the SLCD_Init() API.

24.6.2 void SLCD_Deinit (LCD_Type * *base*)

Parameters

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

24.6.3 void SLCD_GetDefaultConfig (slcd_config_t * *configure*)

The purpose of this API is to get default parameters of the configuration structure for the [SLCD_Init\(\)](#). Use these initialized parameters unchanged in [SLCD_Init\(\)](#) or modify fields of the structure before the calling [SLCD_Init\(\)](#). All default parameters of the configure structuration are listed.

```
config.displayMode      = kSLCD_NormalMode; // SLCD normal mode
config.powerSupply      = kSLCD_InternalV113UseChargePump; // Use charge
                        pump internal VLL3
config.voltageTrim      = kSLCD_RegulatedVolatgeTrim00;
config.lowPowerBehavior = kSLCD_EnabledInWaitStop; // Work on low power mode
config.interruptSrc     = 0; // No interrupt source is enabled
config.faultConfig      = NULL; // Fault detection is disabled
config.frameFreqIntEnable = false;
```

Parameters

<i>configure</i>	The SLCD configuration structure pointer.
------------------	---

24.6.4 static void SLCD_StartDisplay (LCD_Type * *base*) [inline], [static]

Parameters

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

24.6.5 static void SLCD_StopDisplay (LCD_Type * *base*) [inline], [static]

There is no waveform generator and all enabled pins only output a low value.

Parameters

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

24.6.6 void SLCD_StartBlinkMode (LCD_Type * *base*, slcd_blink_mode_t *mode*, slcd_blink_rate_t *rate*)

Parameters

<i>base</i>	SLCD peripheral base address.
<i>mode</i>	SLCD blink mode.
<i>rate</i>	SLCD blink rate.

24.6.7 static void SLCD_StopBlinkMode (LCD_Type * *base*) [inline], [static]

Parameters

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

24.6.8 static void SLCD_SetBackPlanePhase (LCD_Type * *base*, uint32_t *pinIdx*, slcd_phase_type_t *phase*) [inline], [static]

This function sets the SLCD back plane pin phase. "kSLCD_PhaseXActivate" setting means the phase X is active for the back plane pin. "kSLCD_NoPhaseActivate" setting means there is no phase active for the back plane pin. For example, set the back plane pin 20 for phase A.

```
* SLCD_SetBackPlanePhase(LCD, 20, kSLCD_PhaseAActivate);
*
```

Function Documentation

Parameters

<i>base</i>	SLCD peripheral base address.
<i>pinIndx</i>	SLCD back plane pin index. Range from 0 to 63.
<i>phase</i>	The phase activates for the back plane pin.

24.6.9 static void SLCD_SetFrontPlaneSegments (LCD_Type * *base*, uint32_t *pinIndx*, uint8_t *operation*) [inline], [static]

This function sets the SLCD front plane segment on or off operation. Each bit turns on or off the segments associated with the front plane pin in the following pattern: HGFEDCBA (most significant bit controls segment H and least significant bit controls segment A). For example, turn on the front plane pin 20 for phase B and phase C.

```
* SLCD_SetFrontPlaneSegments(LCD, 20, (  
    kSLCD_PhaseBActivate | kSLCD_PhaseCActivate));  
*
```

Parameters

<i>base</i>	SLCD peripheral base address.
<i>pinIndx</i>	SLCD back plane pin index. Range from 0 to 63.
<i>operation</i>	The operation for the segment on the front plane pin. This is a logical OR of the enumeration :: slcd_phase_type_t.

24.6.10 static void SLCD_SetFrontPlaneOnePhase (LCD_Type * *base*, uint32_t *pinIndx*, slcd_phase_index_t *phaseIndx*, bool *enable*) [inline], [static]

This function can be used to set one phase on or off for the front plane pin. It can be call many times to set the plane pin for different phase indexes. For example, turn on the front plane pin 20 for phase B and phase C.

```
* SLCD_SetFrontPlaneOnePhase(LCD, 20,  
    kSLCD_PhaseBIndex, true);  
* SLCD_SetFrontPlaneOnePhase(LCD, 20,  
    kSLCD_PhaseCIndex, true);  
*
```


Parameters

<i>base</i>	SLCD peripheral base address.
<i>pinIndx</i>	SLCD back plane pin index. Range from 0 to 63.
<i>phaseIndx</i>	The phase bit index slcd_phase_index_t .
<i>enable</i>	True to turn on the segment for phaseIndx phase false to turn off the segment for phaseIndx phase.

24.6.11 static void SLCD_EnablePadSafeState (LCD_Type * *base*, bool *enable*) [inline], [static]

Forces the safe state on the LCD pad controls. All LCD front plane and backplane functions are disabled.

Parameters

<i>base</i>	SLCD peripheral base address.
<i>enable</i>	True enable, false disable.

24.6.12 static uint32_t SLCD_GetFaultDetectCounter (LCD_Type * *base*) [inline], [static]

This function gets the number of samples inside the fault detection sample window.

Parameters

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

Returns

The fault detect counter. The maximum return value is 255. If the maximum 255 returns, the overflow may happen. Reconfigure the fault detect sample window and fault detect clock prescaler for proper sampling.

24.6.13 void SLCD_EnableInterrupts (LCD_Type * *base*, uint32_t *mask*)

For example, to enable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
* SLCD_EnableInterrupts(LCD,  
  kSLCD_FaultDetectCompleteInterrupt | kSLCD_FrameFreqInterrupt);  
*
```

Function Documentation

Parameters

<i>base</i>	SLCD peripheral base address.
<i>mask</i>	SLCD interrupts to enable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

24.6.14 void SLCD_DisableInterrupts (LCD_Type * *base*, uint32_t *mask*)

For example, to disable fault detect complete interrupt and frame frequency interrupt, for FSL_FEATURE_SLCD_HAS_FRAME_FREQUENCY_INTERRUPT enabled case, do the following.

```
* SLCD_DisableInterrupts(LCD,  
* kSLCD_FaultDetectCompleteInterrupt | kSLCD_FrameFreqInterrupt);  
*
```

Parameters

<i>base</i>	SLCD peripheral base address.
<i>mask</i>	SLCD interrupts to disable. This is a logical OR of the enumeration :: slcd_interrupt_enable_t.

24.6.15 uint32_t SLCD_GetInterruptStatus (LCD_Type * *base*)

Parameters

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

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration :: slcd_interrupt_enable_t.

24.6.16 void SLCD_ClearInterruptStatus (LCD_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	SLCD peripheral base address.
<i>mask</i>	SLCD interrupt source to be cleared. This is the logical OR of members of the enumeration :: <code>slcd_interrupt_enable_t</code> .

Chapter 25

SMC: System Mode Controller Driver

25.1 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.

25.2 Typical use case

25.2.1 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_AllowPowerModeVlpr` = `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_PowerStateVlprw` = `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)
Recovers after wake up from stop modes.
- static void [SMC_PreEnterWaitModes](#) (void)
Prepares to enter wait modes.
- static void [SMC_PostExitWaitModes](#) (void)
Recovers 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)
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.

25.3 Data Structure Documentation

25.3.1 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.

25.4 Macro Definition Documentation

25.4.1 #define [FSL_SMC_DRIVER_VERSION](#) (MAKE_VERSION(2, 0, 3))

25.5 Enumeration Type Documentation

25.5.1 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.

25.5.2 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

25.5.3 enum smc_run_mode_t

Enumerator

kSMC_RunNormal Normal RUN mode.
kSMC_RunVlpr Very-low-power RUN mode.

25.5.4 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.

25.5.5 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.

25.5.6 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.

25.5.7 enum _smc_status

Enumerator

kStatus_SMC_StopAbort Entering Stop mode is abort.

25.6 Function Documentation

25.6.1 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.

25.6.2 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.

25.6.3 void SMC_PreEnterStopModes (void)

This function should be called before entering STOP/VLPS/LLS/VLLS modes.

25.6.4 void SMC_PostExitStopModes (void)

This function should be called after wake up from STOP/VLPS/LLS/VLLS modes. It is used with [SMC_PreEnterStopModes](#).

25.6.5 static void SMC_PreEnterWaitModes (void) [inline], [static]

This function should be called before entering WAIT/VLPW modes.

25.6.6 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](#).

25.6.7 status_t SMC_SetPowerModeRun (SMC_Type * *base*)

Function Documentation

Parameters

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

Returns

SMC configuration error code.

25.6.8 **status_t SMC_SetPowerModeWait (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

25.6.9 **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.

25.6.10 **status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

25.6.11 **status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

25.6.12 **status_t SMC_SetPowerModeVlps (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

25.6.13 **status_t SMC_SetPowerModeLls (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

25.6.14 **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 26

SPI: Serial Peripheral Interface Driver

26.1 Overview

Modules

- [SPI DMA Driver](#)
- [SPI Driver](#)
- [SPI FreeRTOS driver](#)

26.2 SPI Driver

26.2.1 Overview

SPI driver includes functional APIs and transactional APIs.

Functional APIs are feature/property target low level APIs. Functional APIs can be used for SPI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the SPI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the peripheral base address as the first parameter. SPI functional operation groups provide the functional API set.

Transactional APIs are transaction 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 `spi_handle_t` as the first parameter. Initialize the handle by calling the [SPI_MasterTransferCreateHandle\(\)](#) or [SPI_SlaveTransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer. This means that the functions [SPI_MasterTransferNonBlocking\(\)](#) and [SPI_SlaveTransferNonBlocking\(\)](#) set up the interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_SPI_Idle` status.

26.2.2 Typical use case

26.2.2.1 SPI master transfer using an interrupt method

```
#define BUFFER_LEN (64)
spi_master_handle_t spiHandle;
spi_master_config_t masterConfig;
spi_transfer_t xfer;
volatile bool isFinished = false;

const uint8_t sendData[BUFFER_LEN] = [.....];
uint8_t receiveBuff[BUFFER_LEN];

void SPI_UserCallback(SPI_Type *base, spi_master_handle_t *handle, status_t status, void *userData)
{
    isFinished = true;
}

void main(void)
{
    //...

    SPI_MasterGetDefaultConfig(&masterConfig);

    SPI_MasterInit(SPI0, &masterConfig);
    SPI_MasterTransferCreateHandle(SPI0, &spiHandle, SPI_UserCallback, NULL);

    // Prepare to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
    xfer.dataSize = BUFFER_LEN;

    // Send out.
    SPI_MasterTransferNonBlocking(SPI0, &spiHandle, &xfer);
```



```

    // Wait send finished.
    while (!isFinished)
    {
    }

    // ...
}

```

26.2.2.2 SPI Send/receive using a DMA method

```

#define BUFFER_LEN (64)
spi_dma_handle_t spiHandle;
dma_handle_t g_spiTxDmaHandle;
dma_handle_t g_spiRxDmaHandle;
spi_config_t masterConfig;
spi_transfer_t xfer;
volatile bool isFinished;
uint8_t sendData[BUFFER_LEN] = ...;
uint8_t receiveBuff[BUFFER_LEN];

void SPI_UserCallback(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)
{
    isFinished = true;
}

void main(void)
{
    //...

    SPI_MasterGetDefaultConfig(&masterConfig);
    SPI_MasterInit(SPI0, &masterConfig);

    // Sets up the DMA.
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, SPI_TX_DMA_CHANNEL, SPI_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SPI_TX_DMA_CHANNEL);
    DMAMUX_SetSource(DMAMUX0, SPI_RX_DMA_CHANNEL, SPI_RX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SPI_RX_DMA_CHANNEL);

    DMA_Init(DMA0);

    /* Creates the DMA handle. */
    DMA_CreateHandle(&g_spiTxDmaHandle, DMA0, SPI_TX_DMA_CHANNEL);
    DMA_CreateHandle(&g_spiRxDmaHandle, DMA0, SPI_RX_DMA_CHANNEL);

    SPI_MasterTransferCreateHandleDMA(SPI0, spiHandle, &g_spiTxDmaHandle,
        &g_spiRxDmaHandle, SPI_UserCallback, NULL);

    // Prepares to send.
    xfer.txData = sendData;
    xfer.rxData = receiveBuff;
    xfer.dataSize = BUFFER_LEN;

    // Sends out.
    SPI_MasterTransferDMA(SPI0, &spiHandle, &xfer);

    // Waits for send to complete.
    while (!isFinished)
    {
    }

    // ...
}

```

SPI Driver

Data Structures

- struct [spi_master_config_t](#)
SPI master user configure structure. [More...](#)
- struct [spi_slave_config_t](#)
SPI slave user configure structure. [More...](#)
- struct [spi_transfer_t](#)
SPI transfer structure. [More...](#)
- struct [spi_master_handle_t](#)
SPI transfer handle structure. [More...](#)

Macros

- #define [SPI_DUMMYDATA](#) (0xFFU)
SPI dummy transfer data, the data is sent while txBuff is NULL.

Typedefs

- typedef [spi_master_handle_t](#) [spi_slave_handle_t](#)
Slave handle is the same with master handle.
- typedef void(* [spi_master_callback_t](#))(SPI_Type *base, [spi_master_handle_t](#) *handle, status_t status, void *userData)
SPI master callback for finished transmit.
- typedef void(* [spi_slave_callback_t](#))(SPI_Type *base, [spi_slave_handle_t](#) *handle, status_t status, void *userData)
SPI master callback for finished transmit.

Enumerations

- enum [_spi_status](#) {
 [kStatus_SPI_Busy](#) = MAKE_STATUS(kStatusGroup_SPI, 0),
 [kStatus_SPI_Idle](#) = MAKE_STATUS(kStatusGroup_SPI, 1),
 [kStatus_SPI_Error](#) = MAKE_STATUS(kStatusGroup_SPI, 2) }
Return status for the SPI driver.
- enum [spi_clock_polarity_t](#) {
 [kSPI_ClockPolarityActiveHigh](#) = 0x0U,
 [kSPI_ClockPolarityActiveLow](#) }
SPI clock polarity configuration.
- enum [spi_clock_phase_t](#) {
 [kSPI_ClockPhaseFirstEdge](#) = 0x0U,
 [kSPI_ClockPhaseSecondEdge](#) }
SPI clock phase configuration.
- enum [spi_shift_direction_t](#) {
 [kSPI_MsbFirst](#) = 0x0U,
 [kSPI_LsbFirst](#) }

- SPI data shifter direction options.*

 - enum `spi_ss_output_mode_t` {
`kSPI_SlaveSelectAsGpio` = 0x0U,
`kSPI_SlaveSelectFaultInput` = 0x2U,
`kSPI_SlaveSelectAutomaticOutput` = 0x3U }
- SPI slave select output mode options.*

 - enum `spi_pin_mode_t` {
`kSPI_PinModeNormal` = 0x0U,
`kSPI_PinModeInput` = 0x1U,
`kSPI_PinModeOutput` = 0x3U }
- SPI pin mode options.*

 - enum `spi_data_bitcount_mode_t` {
`kSPI_8BitMode` = 0x0U,
`kSPI_16BitMode` }
- SPI data length mode options.*

 - enum `_spi_interrupt_enable` {
`kSPI_RxFullAndModfInterruptEnable` = 0x1U,
`kSPI_TxEmptyInterruptEnable` = 0x2U,
`kSPI_MatchInterruptEnable` = 0x4U,
`kSPI_RxFifoNearFullInterruptEnable` = 0x8U,
`kSPI_TxFifoNearEmptyInterruptEnable` = 0x10U }
- SPI interrupt sources.*

 - enum `_spi_flags` {
`kSPI_RxBufferFullFlag` = SPI_S_SPRF_MASK,
`kSPI_MatchFlag` = SPI_S_SPMF_MASK,
`kSPI_TxBufferEmptyFlag` = SPI_S_SPTEF_MASK,
`kSPI_ModeFaultFlag` = SPI_S_MODF_MASK,
`kSPI_RxFifoNearFullFlag` = SPI_S_RNFULLF_MASK,
`kSPI_TxFifoNearEmptyFlag` = SPI_S_TNEAREF_MASK,
`kSPI_TxFifoFullFlag` = SPI_S_TXFULLF_MASK,
`kSPI_RxFifoEmptyFlag` = SPI_S_RFIFOEF_MASK,
`kSPI_TxFifoError` = SPI_CI_TXFERR_MASK << 8U,
`kSPI_RxFifoError` = SPI_CI_RXFERR_MASK << 8U,
`kSPI_TxOverflow` = SPI_CI_TXFOF_MASK << 8U,
`kSPI_RxOverflow` = SPI_CI_RXFOF_MASK << 8U }
- SPI status flags.*

 - enum `spi_wlc_interrupt_t` {
`kSPI_RxFifoFullClearInterrupt` = SPI_CI_SPRFCI_MASK,
`kSPI_TxFifoEmptyClearInterrupt` = SPI_CI_SPTEFCI_MASK,
`kSPI_RxNearFullClearInterrupt` = SPI_CI_RNFULLFCI_MASK,
`kSPI_TxNearEmptyClearInterrupt` = SPI_CI_TNEAREFCI_MASK }
- SPI FIFO write-1-to-clear interrupt flags.*

 - enum `spi_txfifo_watermark_t` {
`kSPI_TxFifoOneFourthEmpty` = 0,
`kSPI_TxFifoOneHalfEmpty` = 1 }
- SPI TX FIFO watermark settings.*

 - enum `spi_rxfifo_watermark_t` {

SPI Driver

```
kSPI_RxFifoThreeFourthsFull = 0,  
kSPI_RxFifoOneHalfFull = 1 }
```

SPI RX FIFO watermark settings.

- enum `_spi_dma_enable_t` {
 `kSPI_TxDmaEnable` = `SPI_C2_TXDMAE_MASK`,
 `kSPI_RxDmaEnable` = `SPI_C2_RXDMAE_MASK`,
 `kSPI_DmaAllEnable` = (`SPI_C2_TXDMAE_MASK` | `SPI_C2_RXDMAE_MASK`) }
 SPI DMA source.

Driver version

- #define `FSL_SPI_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 3))
 SPI driver version 2.0.3.

Initialization and deinitialization

- void `SPI_MasterGetDefaultConfig` (`spi_master_config_t` *config)
 Sets the SPI master configuration structure to default values.
- void `SPI_MasterInit` (`SPI_Type` *base, const `spi_master_config_t` *config, `uint32_t` srcClock_Hz)
 Initializes the SPI with master configuration.
- void `SPI_SlaveGetDefaultConfig` (`spi_slave_config_t` *config)
 Sets the SPI slave configuration structure to default values.
- void `SPI_SlaveInit` (`SPI_Type` *base, const `spi_slave_config_t` *config)
 Initializes the SPI with slave configuration.
- void `SPI_Deinit` (`SPI_Type` *base)
 De-initializes the SPI.
- static void `SPI_Enable` (`SPI_Type` *base, bool enable)
 Enables or disables the SPI.

Status

- `uint32_t` `SPI_GetStatusFlags` (`SPI_Type` *base)
 Gets the status flag.
- static void `SPI_ClearInterrupt` (`SPI_Type` *base, `uint32_t` mask)
 Clear the interrupt if enable INCTLR.

Interrupts

- void `SPI_EnableInterrupts` (`SPI_Type` *base, `uint32_t` mask)
 Enables the interrupt for the SPI.
- void `SPI_DisableInterrupts` (`SPI_Type` *base, `uint32_t` mask)
 Disables the interrupt for the SPI.

DMA Control

- static void [SPI_EnableDMA](#) (SPI_Type *base, uint32_t mask, bool enable)
Enables the DMA source for SPI.
- static uint32_t [SPI_GetDataRegisterAddress](#) (SPI_Type *base)
Gets the SPI tx/rx data register address.

Bus Operations

- void [SPI_MasterSetBaudRate](#) (SPI_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
Sets the baud rate for SPI transfer.
- static void [SPI_SetMatchData](#) (SPI_Type *base, uint32_t matchData)
Sets the match data for SPI.
- void [SPI_EnableFIFO](#) (SPI_Type *base, bool enable)
Enables or disables the FIFO if there is a FIFO.
- void [SPI_WriteBlocking](#) (SPI_Type *base, uint8_t *buffer, size_t size)
Sends a buffer of data bytes using a blocking method.
- void [SPI_WriteData](#) (SPI_Type *base, uint16_t data)
Writes a data into the SPI data register.
- uint16_t [SPI_ReadData](#) (SPI_Type *base)
Gets a data from the SPI data register.

Transactional

- void [SPI_MasterTransferCreateHandle](#) (SPI_Type *base, spi_master_handle_t *handle, [spi_master_callback_t](#) callback, void *userData)
Initializes the SPI master handle.
- status_t [SPI_MasterTransferBlocking](#) (SPI_Type *base, [spi_transfer_t](#) *xfer)
Transfers a block of data using a polling method.
- status_t [SPI_MasterTransferNonBlocking](#) (SPI_Type *base, spi_master_handle_t *handle, [spi_transfer_t](#) *xfer)
Performs a non-blocking SPI interrupt transfer.
- status_t [SPI_MasterTransferGetCount](#) (SPI_Type *base, spi_master_handle_t *handle, size_t *count)
Gets the bytes of the SPI interrupt transferred.
- void [SPI_MasterTransferAbort](#) (SPI_Type *base, spi_master_handle_t *handle)
Aborts an SPI transfer using interrupt.
- void [SPI_MasterTransferHandleIRQ](#) (SPI_Type *base, spi_master_handle_t *handle)
Interrupts the handler for the SPI.
- void [SPI_SlaveTransferCreateHandle](#) (SPI_Type *base, [spi_slave_handle_t](#) *handle, [spi_slave_callback_t](#) callback, void *userData)
Initializes the SPI slave handle.
- static status_t [SPI_SlaveTransferNonBlocking](#) (SPI_Type *base, [spi_slave_handle_t](#) *handle, [spi_transfer_t](#) *xfer)
Performs a non-blocking SPI slave interrupt transfer.
- static status_t [SPI_SlaveTransferGetCount](#) (SPI_Type *base, [spi_slave_handle_t](#) *handle, size_t *count)

SPI Driver

- *Gets the bytes of the SPI interrupt transferred.*
- static void [SPI_SlaveTransferAbort](#) (SPI_Type *base, [spi_slave_handle_t](#) *handle)
Aborts an SPI slave transfer using interrupt.
- void [SPI_SlaveTransferHandleIRQ](#) (SPI_Type *base, [spi_slave_handle_t](#) *handle)
Interrupts a handler for the SPI slave.

26.2.3 Data Structure Documentation

26.2.3.1 struct spi_master_config_t

Data Fields

- bool [enableMaster](#)
Enable SPI at initialization time.
- bool [enableStopInWaitMode](#)
SPI stop in wait mode.
- [spi_clock_polarity_t](#) polarity
Clock polarity.
- [spi_clock_phase_t](#) phase
Clock phase.
- [spi_shift_direction_t](#) direction
MSB or LSB.
- [spi_data_bitcount_mode_t](#) dataMode
8bit or 16bit mode
- [spi_txfifo_watermark_t](#) txWatermark
Tx watermark settings.
- [spi_rxfifo_watermark_t](#) rxWatermark
Rx watermark settings.
- [spi_ss_output_mode_t](#) outputMode
SS pin setting.
- [spi_pin_mode_t](#) pinMode
SPI pin mode select.
- uint32_t [baudRate_Bps](#)
Baud Rate for SPI in Hz.

26.2.3.2 struct spi_slave_config_t

Data Fields

- bool [enableSlave](#)
Enable SPI at initialization time.
- bool [enableStopInWaitMode](#)
SPI stop in wait mode.
- [spi_clock_polarity_t](#) polarity
Clock polarity.
- [spi_clock_phase_t](#) phase
Clock phase.
- [spi_shift_direction_t](#) direction

- *MSB or LSB.*
- [spi_data_bitcount_mode_t](#) *dataMode*
8bit or 16bit mode
- [spi_txfifo_watermark_t](#) *txWatermark*
Tx watermark settings.
- [spi_rxfifo_watermark_t](#) *rxWatermark*
Rx watermark settings.

26.2.3.3 struct spi_transfer_t

Data Fields

- [uint8_t](#) * [txData](#)
Send buffer.
- [uint8_t](#) * [rxData](#)
Receive buffer.
- [size_t](#) [dataSize](#)
Transfer bytes.
- [uint32_t](#) [flags](#)
SPI control flag, useless to SPI.

26.2.3.3.0.44 Field Documentation

26.2.3.3.0.44.1 [uint32_t spi_transfer_t::flags](#)

26.2.3.4 struct _spi_master_handle

Data Fields

- [uint8_t](#) *volatile [txData](#)
Transfer buffer.
- [uint8_t](#) *volatile [rxData](#)
Receive buffer.
- volatile [size_t](#) [txRemainingBytes](#)
Send data remaining in bytes.
- volatile [size_t](#) [rxRemainingBytes](#)
Receive data remaining in bytes.
- volatile [uint32_t](#) [state](#)
SPI internal state.
- [size_t](#) [transferSize](#)
Bytes to be transferred.
- [uint8_t](#) [bytePerFrame](#)
SPI mode, 2bytes or 1byte in a frame.
- [uint8_t](#) [watermark](#)
Watermark value for SPI transfer.
- [spi_master_callback_t](#) [callback](#)
SPI callback.
- void * [userData](#)
Callback parameter.

SPI Driver

26.2.4 Macro Definition Documentation

26.2.4.1 **#define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))**

26.2.4.2 **#define SPI_DUMMYDATA (0xFFU)**

26.2.5 Enumeration Type Documentation

26.2.5.1 enum _spi_status

Enumerator

kStatus_SPI_Busy SPI bus is busy.

kStatus_SPI_Idle SPI is idle.

kStatus_SPI_Error SPI error.

26.2.5.2 enum spi_clock_polarity_t

Enumerator

kSPI_ClockPolarityActiveHigh Active-high SPI clock (idles low).

kSPI_ClockPolarityActiveLow Active-low SPI clock (idles high).

26.2.5.3 enum spi_clock_phase_t

Enumerator

kSPI_ClockPhaseFirstEdge First edge on SPSCCK occurs at the middle of the first cycle of a data transfer.

kSPI_ClockPhaseSecondEdge First edge on SPSCCK occurs at the start of the first cycle of a data transfer.

26.2.5.4 enum spi_shift_direction_t

Enumerator

kSPI_MsbFirst Data transfers start with most significant bit.

kSPI_LsbFirst Data transfers start with least significant bit.

26.2.5.5 enum spi_ss_output_mode_t

Enumerator

- kSPI_SlaveSelectAsGpio* Slave select pin configured as GPIO.
- kSPI_SlaveSelectFaultInput* Slave select pin configured for fault detection.
- kSPI_SlaveSelectAutomaticOutput* Slave select pin configured for automatic SPI output.

26.2.5.6 enum spi_pin_mode_t

Enumerator

- kSPI_PinModeNormal* Pins operate in normal, single-direction mode.
- kSPI_PinModeInput* Bidirectional mode. Master: MOSI pin is input; Slave: MISO pin is input.
- kSPI_PinModeOutput* Bidirectional mode. Master: MOSI pin is output; Slave: MISO pin is output.

26.2.5.7 enum spi_data_bitcount_mode_t

Enumerator

- kSPI_8BitMode* 8-bit data transmission mode
- kSPI_16BitMode* 16-bit data transmission mode

26.2.5.8 enum _spi_interrupt_enable

Enumerator

- kSPI_RxFullAndModfInterruptEnable* Receive buffer full (SPRF) and mode fault (MODF) interrupt.
- kSPI_TxEmptyInterruptEnable* Transmit buffer empty interrupt.
- kSPI_MatchInterruptEnable* Match interrupt.
- kSPI_RxFifoNearFullInterruptEnable* Receive FIFO nearly full interrupt.
- kSPI_TxFifoNearEmptyInterruptEnable* Transmit FIFO nearly empty interrupt.

26.2.5.9 enum _spi_flags

Enumerator

- kSPI_RxBufferFullFlag* Read buffer full flag.
- kSPI_MatchFlag* Match flag.
- kSPI_TxBufferEmptyFlag* Transmit buffer empty flag.
- kSPI_ModeFaultFlag* Mode fault flag.
- kSPI_RxFifoNearFullFlag* Rx FIFO near full.

SPI Driver

kSPI_TxFifoNearEmptyFlag Tx FIFO near empty.

kSPI_TxFifoFullFlag Tx FIFO full.

kSPI_RxFifoEmptyFlag Rx FIFO empty.

kSPI_TxFifoError Tx FIFO error.

kSPI_RxFifoError Rx FIFO error.

kSPI_TxOverflow Tx FIFO Overflow.

kSPI_RxOverflow Rx FIFO Overflow.

26.2.5.10 enum spi_w1c_interrupt_t

Enumerator

kSPI_RxFifoFullClearInterrupt Receive FIFO full interrupt.

kSPI_TxFifoEmptyClearInterrupt Transmit FIFO empty interrupt.

kSPI_RxNearFullClearInterrupt Receive FIFO nearly full interrupt.

kSPI_TxNearEmptyClearInterrupt Transmit FIFO nearly empty interrupt.

26.2.5.11 enum spi_txfifo_watermark_t

Enumerator

kSPI_TxFifoOneFourthEmpty SPI tx watermark at 1/4 FIFO size.

kSPI_TxFifoOneHalfEmpty SPI tx watermark at 1/2 FIFO size.

26.2.5.12 enum spi_rxfifo_watermark_t

Enumerator

kSPI_RxFifoThreeFourthsFull SPI rx watermark at 3/4 FIFO size.

kSPI_RxFifoOneHalfFull SPI rx watermark at 1/2 FIFO size.

26.2.5.13 enum _spi_dma_enable_t

Enumerator

kSPI_TxDmaEnable Tx DMA request source.

kSPI_RxDmaEnable Rx DMA request source.

kSPI_DmaAllEnable All DMA request source.

26.2.6 Function Documentation

26.2.6.1 void SPI_MasterGetDefaultConfig (spi_master_config_t * *config*)

The purpose of this API is to get the configuration structure initialized for use in [SPI_MasterInit\(\)](#). User may use the initialized structure unchanged in [SPI_MasterInit\(\)](#), or modify some fields of the structure before calling [SPI_MasterInit\(\)](#). After calling this API, the master is ready to transfer. Example:

```
spi_master_config_t config;
SPI_MasterGetDefaultConfig(&config);
```

Parameters

<i>config</i>	pointer to master config structure
---------------	------------------------------------

26.2.6.2 void SPI_MasterInit (SPI_Type * *base*, const spi_master_config_t * *config*, uint32_t *srcClock_Hz*)

The configuration structure can be filled by user from scratch, or be set with default values by [SPI_MasterGetDefaultConfig\(\)](#). After calling this API, the slave is ready to transfer. Example

```
spi_master_config_t config = {
    .baudRate_Bps = 400000,
    ...
};
SPI_MasterInit(SPI0, &config);
```

Parameters

<i>base</i>	SPI base pointer
<i>config</i>	pointer to master configuration structure
<i>srcClock_Hz</i>	Source clock frequency.

26.2.6.3 void SPI_SlaveGetDefaultConfig (spi_slave_config_t * *config*)

The purpose of this API is to get the configuration structure initialized for use in [SPI_SlaveInit\(\)](#). Modify some fields of the structure before calling [SPI_SlaveInit\(\)](#). Example:

```
spi_slave_config_t config;
SPI_SlaveGetDefaultConfig(&config);
```

SPI Driver

Parameters

<i>config</i>	pointer to slave configuration structure
---------------	--

26.2.6.4 void SPI_SlaveInit (SPI_Type * *base*, const spi_slave_config_t * *config*)

The configuration structure can be filled by user from scratch or be set with default values by [SPI_Slave-GetDefaultConfig\(\)](#). After calling this API, the slave is ready to transfer. Example

```
spi_slave_config_t config = {  
.polarity = kSPIClockPolarity_ActiveHigh;  
.phase = kSPIClockPhase_FirstEdge;  
.direction = kSPIMsbFirst;  
...  
};  
SPI_MasterInit(SPI0, &config);
```

Parameters

<i>base</i>	SPI base pointer
<i>config</i>	pointer to master configuration structure

26.2.6.5 void SPI_Deinit (SPI_Type * *base*)

Calling this API resets the SPI module, gates the SPI clock. The SPI module can't work unless calling the SPI_MasterInit/SPI_SlaveInit to initialize module.

Parameters

<i>base</i>	SPI base pointer
-------------	------------------

26.2.6.6 static void SPI_Enable (SPI_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SPI base pointer
<i>enable</i>	pass true to enable module, false to disable module

26.2.6.7 uint32_t SPI_GetStatusFlags (SPI_Type * *base*)

Parameters

<i>base</i>	SPI base pointer
-------------	------------------

Returns

SPI Status, use status flag to AND [_spi_flags](#) could get the related status.

26.2.6.8 static void SPI_ClearInterrupt (SPI_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SPI base pointer
<i>interrupt</i>	Interrupt need to be cleared The parameter could be any combination of the following values: <ul style="list-style-type: none"> • kSPIRxFifoFullClearInt • kSPITxFifoEmptyClearInt • kSPIRxFifoNearFullClearInt • kSPITxFifoNearEmptyClearInt

26.2.6.9 void SPI_EnableInterrupts (SPI_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	SPI base pointer
<i>mask</i>	SPI interrupt source. The parameter can be any combination of the following values: <ul style="list-style-type: none"> • kSPI_RxFifoFullAndModfInterruptEnable • kSPI_TxFifoEmptyInterruptEnable • kSPI_MatchInterruptEnable • kSPI_RxFifoNearFullInterruptEnable • kSPI_TxFifoNearEmptyInterruptEnable

26.2.6.10 void SPI_DisableInterrupts (SPI_Type * *base*, uint32_t *mask*)

SPI Driver

Parameters

<i>base</i>	SPI base pointer
<i>mask</i>	SPI interrupt source. The parameter can be any combination of the following values: <ul style="list-style-type: none">• kSPI_RxFullAndModfInterruptEnable• kSPI_TxEmptyInterruptEnable• kSPI_MatchInterruptEnable• kSPI_RxFifoNearFullInterruptEnable• kSPI_TxFifoNearEmptyInterruptEnable

26.2.6.11 static void SPI_EnableDMA (SPI_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SPI base pointer
<i>source</i>	SPI DMA source.
<i>enable</i>	True means enable DMA, false means disable DMA

26.2.6.12 static uint32_t SPI_GetDataRegisterAddress (SPI_Type * *base*) [inline], [static]

This API is used to provide a transfer address for the SPI DMA transfer configuration.

Parameters

<i>base</i>	SPI base pointer
-------------	------------------

Returns

data register address

26.2.6.13 void SPI_MasterSetBaudRate (SPI_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

This is only used in master.

Parameters

<i>base</i>	SPI base pointer
<i>baudRate_Bps</i>	baud rate needed in Hz.
<i>srcClock_Hz</i>	SPI source clock frequency in Hz.

26.2.6.14 static void SPI_SetMatchData (SPI_Type * *base*, uint32_t *matchData*) [inline], [static]

The match data is a hardware comparison value. When the value received in the SPI receive data buffer equals the hardware comparison value, the SPI Match Flag in the S register (S[SPMF]) sets. This can also generate an interrupt if the enable bit sets.

Parameters

<i>base</i>	SPI base pointer
<i>matchData</i>	Match data.

26.2.6.15 void SPI_EnableFIFO (SPI_Type * *base*, bool *enable*)

Parameters

<i>base</i>	SPI base pointer
<i>enable</i>	True means enable FIFO, false means disable FIFO.

26.2.6.16 void SPI_WriteBlocking (SPI_Type * *base*, uint8_t * *buffer*, size_t *size*)

Note

This function blocks via polling until all bytes have been sent.

Parameters

<i>base</i>	SPI base pointer
-------------	------------------

SPI Driver

<i>buffer</i>	The data bytes to send
<i>size</i>	The number of data bytes to send

26.2.6.17 void SPI_WriteData (SPI_Type * *base*, uint16_t *data*)

Parameters

<i>base</i>	SPI base pointer
<i>data</i>	needs to be write.

26.2.6.18 uint16_t SPI_ReadData (SPI_Type * *base*)

Parameters

<i>base</i>	SPI base pointer
-------------	------------------

Returns

Data in the register.

26.2.6.19 void SPI_MasterTransferCreateHandle (SPI_Type * *base*, spi_master_handle_t * *handle*, spi_master_callback_t *callback*, void * *userData*)

This function initializes the SPI master handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.
<i>callback</i>	Callback function.
<i>userData</i>	User data.

26.2.6.20 status_t SPI_MasterTransferBlocking (SPI_Type * *base*, spi_transfer_t * *xfer*)

Parameters

<i>base</i>	SPI base pointer
<i>xfer</i>	pointer to spi_xfer_config_t structure

Return values

<i>kStatus_Success</i>	Successfully start a transfer.
<i>kStatus_InvalidArgument</i>	Input argument is invalid.

26.2.6.21 status_t SPI_MasterTransferNonBlocking (SPI_Type * *base*, spi_master_handle_t * *handle*, spi_transfer_t * *xfer*)

Note

The API immediately returns after transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.

If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_master_handle_t structure which stores the transfer state
<i>xfer</i>	pointer to spi_xfer_config_t structure

Return values

<i>kStatus_Success</i>	Successfully start a transfer.
<i>kStatus_InvalidArgument</i>	Input argument is invalid.
<i>kStatus_SPI_Busy</i>	SPI is not idle, is running another transfer.

26.2.6.22 status_t SPI_MasterTransferGetCount (SPI_Type * *base*, spi_master_handle_t * *handle*, size_t * *count*)

Parameters

SPI Driver

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to SPI transfer handle, this should be a static variable.
<i>count</i>	Transferred bytes of SPI master.

Return values

<i>kStatus_SPI_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is not a non-blocking transaction currently in progress.

26.2.6.23 void SPI_MasterTransferAbort (SPI_Type * *base*, spi_master_handle_t * *handle*)

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to SPI transfer handle, this should be a static variable.

26.2.6.24 void SPI_MasterTransferHandleIRQ (SPI_Type * *base*, spi_master_handle_t * *handle*)

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_master_handle_t structure which stores the transfer state.

26.2.6.25 void SPI_SlaveTransferCreateHandle (SPI_Type * *base*, spi_slave_handle_t * *handle*, spi_slave_callback_t *callback*, void * *userData*)

This function initializes the SPI slave handle which can be used for other SPI slave transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.
<i>callback</i>	Callback function.
<i>userData</i>	User data.

26.2.6.26 static status_t SPI_SlaveTransferNonBlocking (SPI_Type * *base*, spi_slave_handle_t * *handle*, spi_transfer_t * *xfer*) [inline], [static]

Note

The API returns immediately after the transfer initialization is finished. Call SPI_GetStatusIRQ() to get the transfer status.

If SPI transfer data frame size is 16 bits, the transfer size cannot be an odd number.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_master_handle_t structure which stores the transfer state
<i>xfer</i>	pointer to spi_xfer_config_t structure

Return values

<i>kStatus_Success</i>	Successfully start a transfer.
<i>kStatus_InvalidArgument</i>	Input argument is invalid.
<i>kStatus_SPI_Busy</i>	SPI is not idle, is running another transfer.

26.2.6.27 static status_t SPI_SlaveTransferGetCount (SPI_Type * *base*, spi_slave_handle_t * *handle*, size_t * *count*) [inline], [static]

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to SPI transfer handle, this should be a static variable.

SPI Driver

<i>count</i>	Transferred bytes of SPI slave.
--------------	---------------------------------

Return values

<i>kStatus_SPI_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is not a non-blocking transaction currently in progress.

26.2.6.28 static void SPI_SlaveTransferAbort (SPI_Type * *base*, spi_slave_handle_t * *handle*) [inline], [static]

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	Pointer to SPI transfer handle, this should be a static variable.

26.2.6.29 void SPI_SlaveTransferHandleIRQ (SPI_Type * *base*, spi_slave_handle_t * *handle*)

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	pointer to spi_slave_handle_t structure which stores the transfer state

26.3 SPI DMA Driver

26.3.1 Overview

This section describes the programming interface of the SPI DMA driver.

Data Structures

- struct [spi_dma_handle_t](#)
SPI DMA transfer handle, users should not touch the content of the handle. [More...](#)

Typedefs

- typedef void(* [spi_dma_callback_t](#))(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)
SPI DMA callback called at the end of transfer.

DMA Transactional

- void [SPI_MasterTransferCreateHandleDMA](#) (SPI_Type *base, spi_dma_handle_t *handle, [spi_dma_callback_t](#) callback, void *userData, [dma_handle_t](#) *txHandle, [dma_handle_t](#) *rxHandle)
Initialize the SPI master DMA handle.
- status_t [SPI_MasterTransferDMA](#) (SPI_Type *base, spi_dma_handle_t *handle, [spi_transfer_t](#) *xfer)
Perform a non-blocking SPI transfer using DMA.
- void [SPI_MasterTransferAbortDMA](#) (SPI_Type *base, spi_dma_handle_t *handle)
Abort a SPI transfer using DMA.
- status_t [SPI_MasterTransferGetCountDMA](#) (SPI_Type *base, spi_dma_handle_t *handle, size_t *count)
Get the transferred bytes for SPI slave DMA.
- static void [SPI_SlaveTransferCreateHandleDMA](#) (SPI_Type *base, spi_dma_handle_t *handle, [spi_dma_callback_t](#) callback, void *userData, [dma_handle_t](#) *txHandle, [dma_handle_t](#) *rxHandle)
Initialize the SPI slave DMA handle.
- static status_t [SPI_SlaveTransferDMA](#) (SPI_Type *base, spi_dma_handle_t *handle, [spi_transfer_t](#) *xfer)
Perform a non-blocking SPI transfer using DMA.
- static void [SPI_SlaveTransferAbortDMA](#) (SPI_Type *base, spi_dma_handle_t *handle)
Abort a SPI transfer using DMA.
- static status_t [SPI_SlaveTransferGetCountDMA](#) (SPI_Type *base, spi_dma_handle_t *handle, size_t *count)
Get the transferred bytes for SPI slave DMA.

26.3.2 Data Structure Documentation

26.3.2.1 struct _spi_dma_handle

Data Fields

- bool [txInProgress](#)
Send transfer finished.
- bool [rxInProgress](#)
Receive transfer finished.
- [dma_handle_t](#) * [txHandle](#)
DMA handler for SPI send.
- [dma_handle_t](#) * [rxHandle](#)
DMA handler for SPI receive.
- uint8_t [bytesPerFrame](#)
Bytes in a frame for SPI transfer.
- [spi_dma_callback_t](#) [callback](#)
Callback for SPI DMA transfer.
- void * [userData](#)
User Data for SPI DMA callback.
- uint32_t [state](#)
Internal state of SPI DMA transfer.
- size_t [transferSize](#)
Bytes need to be transfer.

26.3.3 Typedef Documentation

26.3.3.1 **typedef void(* spi_dma_callback_t)(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)**

26.3.4 Function Documentation

26.3.4.1 **void SPI_MasterTransferCreateHandleDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, spi_dma_callback_t *callback*, void * *userData*, dma_handle_t * *txHandle*, dma_handle_t * *rxHandle*)**

This function initializes the SPI master DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.
<i>callback</i>	User callback function called at the end of a transfer.
<i>userData</i>	User data for callback.
<i>txHandle</i>	DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
<i>rxHandle</i>	DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

26.3.4.2 **status_t SPI_MasterTransferDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, spi_transfer_t * *xfer*)**

Note

This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.
<i>xfer</i>	Pointer to dma transfer structure.

Return values

<i>kStatus_Success</i>	Successfully start a transfer.
<i>kStatus_InvalidArgument</i>	Input argument is invalid.
<i>kStatus_SPI_Busy</i>	SPI is not idle, is running another transfer.

26.3.4.3 **void SPI_MasterTransferAbortDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*)**

Parameters

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

SPI DMA Driver

<i>handle</i>	SPI DMA handle pointer.
---------------	-------------------------

26.3.4.4 `status_t SPI_MasterTransferGetCountDMA (SPI_Type * base, spi_dma_handle_t * handle, size_t * count)`

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.
<i>count</i>	Transferred bytes.

Return values

<i>kStatus_SPI_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is not a non-blocking transaction currently in progress.

26.3.4.5 `static void SPI_SlaveTransferCreateHandleDMA (SPI_Type * base, spi_dma_handle_t * handle, spi_dma_callback_t callback, void * userData, dma_handle_t * txHandle, dma_handle_t * rxHandle) [inline], [static]`

This function initializes the SPI slave DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI handle pointer.
<i>callback</i>	User callback function called at the end of a transfer.
<i>userData</i>	User data for callback.
<i>txHandle</i>	DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
<i>rxHandle</i>	DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

26.3.4.6 `static status_t SPI_SlaveTransferDMA (SPI_Type * base, spi_dma_handle_t * handle, spi_transfer_t * xfer) [inline], [static]`

Note

This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.
<i>xfer</i>	Pointer to dma transfer structure.

Return values

<i>kStatus_Success</i>	Successfully start a transfer.
<i>kStatus_InvalidArgument</i>	Input argument is invalid.
<i>kStatus_SPI_Busy</i>	SPI is not idle, is running another transfer.

26.3.4.7 static void SPI_SlaveTransferAbortDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*) [inline], [static]

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.

26.3.4.8 static status_t SPI_SlaveTransferGetCountDMA (SPI_Type * *base*, spi_dma_handle_t * *handle*, size_t * *count*) [inline], [static]

Parameters

<i>base</i>	SPI peripheral base address.
<i>handle</i>	SPI DMA handle pointer.
<i>count</i>	Transferred bytes.

Return values

SPI DMA Driver

<i>kStatus_SPI_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is not a non-blocking transaction currently in progress.

26.4 SPI FreeRTOS driver

26.4.1 Overview

This section describes the programming interface of the SPI FreeRTOS driver.

SPI RTOS Operation

- `status_t SPI_RTOS_Init (spi_rtos_handle_t *handle, SPI_Type *base, const spi_master_config_t *masterConfig, uint32_t srcClock_Hz)`
Initializes SPI.
- `status_t SPI_RTOS_Deinit (spi_rtos_handle_t *handle)`
Deinitializes the SPI.
- `status_t SPI_RTOS_Transfer (spi_rtos_handle_t *handle, spi_transfer_t *transfer)`
Performs SPI transfer.

26.4.2 Function Documentation

26.4.2.1 `status_t SPI_RTOS_Init (spi_rtos_handle_t * handle, SPI_Type * base, const spi_master_config_t * masterConfig, uint32_t srcClock_Hz)`

This function initializes the SPI module and related RTOS context.

Parameters

<i>handle</i>	The RTOS SPI handle, the pointer to an allocated space for RTOS context.
<i>base</i>	The pointer base address of the SPI instance to initialize.
<i>masterConfig</i>	Configuration structure to set-up SPI in master mode.
<i>srcClock_Hz</i>	Frequency of input clock of the SPI module.

Returns

status of the operation.

26.4.2.2 `status_t SPI_RTOS_Deinit (spi_rtos_handle_t * handle)`

This function deinitializes the SPI module and related RTOS context.

SPI FreeRTOS driver

Parameters

<i>handle</i>	The RTOS SPI handle.
---------------	----------------------

26.4.2.3 **status_t SPI_RTOS_Transfer (spi_rtos_handle_t * *handle*, spi_transfer_t * *transfer*)**

This function performs an SPI transfer according to data given in the transfer structure.

Parameters

<i>handle</i>	The RTOS SPI handle.
<i>transfer</i>	Structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 27

TPM: Timer PWM Module

27.1 Overview

The MCUXpresso SDK provides a driver for the Timer PWM Module (TPM) of MCUXpresso SDK devices.

The TPM driver supports the generation of PWM signals, input capture, and output compare modes. On some SoCs, the driver supports the generation of combined PWM signals, dual-edge capture, and quadrature decoder modes. The driver also supports configuring each of the TPM fault inputs. The fault input is available only on some SoCs.

The function [TPM_Init\(\)](#) initializes the TPM with a specified configurations. The function [TPM_GetDefaultConfig\(\)](#) gets the default configurations. On some SoCs, the initialization function issues a software reset to reset the TPM internal logic. The initialization function configures the TPM's behavior when it receives a trigger input and its operation in doze and debug modes.

The function [TPM_Deinit\(\)](#) disables the TPM counter and turns off the module clock.

The function [TPM_SetupPwm\(\)](#) sets up TPM channels for the PWM output. The function can set up the PWM signal properties for multiple channels. Each channel has its own [tpm_chnl_pwm_signal_param_t](#) structure that is used to specify the output signals duty cycle and level-mode. However, the same PWM period and PWM mode is applied to all channels requesting a PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 where 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle). When generating a combined PWM signal, the channel number passed refers to a channel pair number, for example 0 refers to channel 0 and 1, 1 refers to channels 2 and 3.

The function [TPM_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular TPM channel.

The function [TPM_UpdateChnlEdgeLevelSelect\(\)](#) updates the level select bits of a particular TPM channel. This can be used to disable the PWM output when making changes to the PWM signal.

The function [TPM_SetupInputCapture\(\)](#) sets up a TPM channel for input capture. The user can specify the capture edge.

The function [TPM_SetupDualEdgeCapture\(\)](#) can be used to measure the pulse width of a signal. This is available only for certain SoCs. A channel pair is used during the capture with the input signal coming through a channel that can be configured. The user can specify the capture edge for each channel and any filter value to be used when processing the input signal.

The function [TPM_SetupOutputCompare\(\)](#) sets up a TPM channel for output comparison. The user can specify the channel output on a successful comparison and a comparison value.

The function [TPM_SetupQuadDecode\(\)](#) sets up TPM channels 0 and 1 for quad decode, which is available only for certain SoCs. The user can specify the quad decode mode, polarity, and filter properties for each

Typical use case

input signal.

The function `TPM_SetupFault()` sets up the properties for each fault, which is available only for certain SoCs. 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.

Provides functions to get and clear the TPM status.

Provides functions to enable/disable TPM interrupts and get current enabled interrupts.

27.2 Typical use case

27.2.1 PWM output

Output the PWM signal on 2 TPM channels with different duty cycles. Periodically update the PWM signal duty cycle.

```
int main(void)
{
    bool brightnessUp = true; /* Indicates whether the LED is brighter or dimmer. */
    tpm_config_t tpmInfo;
    uint8_t updatedDutycycle = 0U;
    tpm_chnl_pwm_signal_param_t tpmParam[2];

    /* Configures the TPM parameters with frequency 24 kHz. */
    tpmParam[0].chnlNumber = (tpm_chnl_t)BOARD_FIRST_TPM_CHANNEL;
    tpmParam[0].level = kTPM_LowTrue;
    tpmParam[0].dutyCyclePercent = 0U;

    tpmParam[1].chnlNumber = (tpm_chnl_t)BOARD_SECOND_TPM_CHANNEL;
    tpmParam[1].level = kTPM_LowTrue;
    tpmParam[1].dutyCyclePercent = 0U;

    /* Board pin, clock, and debug console initialization. */
    BOARD_InitHardware();

    TPM_GetDefaultConfig(&tpmInfo);
    /* Initializes the TPM module. */
    TPM_Init(BOARD_TPM_BASEADDR, &tpmInfo);

    TPM_SetupPwm(BOARD_TPM_BASEADDR, tpmParam, 2U,
                 kTPM_EdgeAlignedPwm, 24000U, TPM_SOURCE_CLOCK);
    TPM_StartTimer(BOARD_TPM_BASEADDR, kTPM_SystemClock);
    while (1)
    {
        /* Delays to see the change of LED brightness. */
        delay();

        if (brightnessUp)
        {
            /* Increases a duty cycle until it reaches a limited value. */
            if (++updatedDutycycle == 100U)
            {
                brightnessUp = false;
            }
        }
        else
        {
            /* Decreases a duty cycle until it reaches a limited value. */
            if (--updatedDutycycle == 0U)
            {
                brightnessUp = true;
            }
        }
    }
}
```

```

    }
}
/* Starts PWM mode with an updated duty cycle. */
TPM_UpdatePwmDutycycle(BOARD_TPM_BASEADDR, (
tpm_chnl_t)BOARD_FIRST_TPM_CHANNEL, kTPM_EdgeAlignedPwm,
    updatedDutycycle);
TPM_UpdatePwmDutycycle(BOARD_TPM_BASEADDR, (
tpm_chnl_t)BOARD_SECOND_TPM_CHANNEL, kTPM_EdgeAlignedPwm,
    updatedDutycycle);
}
}

```

Data Structures

- struct `tpm_chnl_pwm_signal_param_t`
Options to configure a TPM channel's PWM signal. [More...](#)
- struct `tpm_config_t`
TPM config structure. [More...](#)

Enumerations

- enum `tpm_chnl_t` {
kTPM_Chnl_0 = 0U,
kTPM_Chnl_1,
kTPM_Chnl_2,
kTPM_Chnl_3,
kTPM_Chnl_4,
kTPM_Chnl_5,
kTPM_Chnl_6,
kTPM_Chnl_7 }
List of TPM channels.
- enum `tpm_pwm_mode_t` {
kTPM_EdgeAlignedPwm = 0U,
kTPM_CenterAlignedPwm }
TPM PWM operation modes.
- enum `tpm_pwm_level_select_t` {
kTPM_NoPwmSignal = 0U,
kTPM_LowTrue,
kTPM_HighTrue }
TPM PWM output pulse mode: high-true, low-true or no output.
- enum `tpm_trigger_select_t`
Trigger options available.
- enum `tpm_output_compare_mode_t` {
kTPM_NoOutputSignal = (1U << TPM_CnSC_MSA_SHIFT),
kTPM_ToggleOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_S-
HIFT)),
kTPM_ClearOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_SH-
IFT)),
kTPM_SetOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (3U << TPM_CnSC_ELSA_SHIF-
T)),
kTPM_HighPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_-

Typical use case

```
SHIFT)),  
kTPM_LowPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_S-  
HIFT)) }
```

TPM output compare modes.

- enum `tpm_input_capture_edge_t` {
 `kTPM_RisingEdge` = (1U << TPM_CnSC_ELSA_SHIFT),
 `kTPM_FallingEdge` = (2U << TPM_CnSC_ELSA_SHIFT),
 `kTPM_RiseAndFallEdge` = (3U << TPM_CnSC_ELSA_SHIFT) }

TPM input capture edge.

- enum `tpm_clock_source_t` {
 `kTPM_SystemClock` = 1U,
 `kTPM_ExternalClock` }

TPM clock source selection.

- enum `tpm_clock_prescale_t` {
 `kTPM_Prescale_Divide_1` = 0U,
 `kTPM_Prescale_Divide_2`,
 `kTPM_Prescale_Divide_4`,
 `kTPM_Prescale_Divide_8`,
 `kTPM_Prescale_Divide_16`,
 `kTPM_Prescale_Divide_32`,
 `kTPM_Prescale_Divide_64`,
 `kTPM_Prescale_Divide_128` }

TPM prescale value selection for the clock source.

- enum `tpm_interrupt_enable_t` {
 `kTPM_Chnl0InterruptEnable` = (1U << 0),
 `kTPM_Chnl1InterruptEnable` = (1U << 1),
 `kTPM_Chnl2InterruptEnable` = (1U << 2),
 `kTPM_Chnl3InterruptEnable` = (1U << 3),
 `kTPM_Chnl4InterruptEnable` = (1U << 4),
 `kTPM_Chnl5InterruptEnable` = (1U << 5),
 `kTPM_Chnl6InterruptEnable` = (1U << 6),
 `kTPM_Chnl7InterruptEnable` = (1U << 7),
 `kTPM_TimeOverflowInterruptEnable` = (1U << 8) }

List of TPM interrupts.

- enum `tpm_status_flags_t` {
 `kTPM_Chnl0Flag` = (1U << 0),
 `kTPM_Chnl1Flag` = (1U << 1),
 `kTPM_Chnl2Flag` = (1U << 2),
 `kTPM_Chnl3Flag` = (1U << 3),
 `kTPM_Chnl4Flag` = (1U << 4),
 `kTPM_Chnl5Flag` = (1U << 5),
 `kTPM_Chnl6Flag` = (1U << 6),
 `kTPM_Chnl7Flag` = (1U << 7),
 `kTPM_TimeOverflowFlag` = (1U << 8) }

List of TPM flags.

Driver version

- #define `FSL_TPM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 2)`)
Version 2.0.2.

Initialization and deinitialization

- void `TPM_Init` (`TPM_Type *base`, const `tpm_config_t *config`)
Ungates the TPM clock and configures the peripheral for basic operation.
- void `TPM_Deinit` (`TPM_Type *base`)
Stops the counter and gates the TPM clock.
- void `TPM_GetDefaultConfig` (`tpm_config_t *config`)
Fill in the TPM config struct with the default settings.

Channel mode operations

- status_t `TPM_SetupPwm` (`TPM_Type *base`, const `tpm_chnl_pwm_signal_param_t *chnlParams`, `uint8_t numOfChnls`, `tpm_pwm_mode_t mode`, `uint32_t pwmFreq_Hz`, `uint32_t srcClock_Hz`)
Configures the PWM signal parameters.
- void `TPM_UpdatePwmDutycycle` (`TPM_Type *base`, `tpm_chnl_t chnlNumber`, `tpm_pwm_mode_t currentPwmMode`, `uint8_t dutyCyclePercent`)
Update the duty cycle of an active PWM signal.
- void `TPM_UpdateChnlEdgeLevelSelect` (`TPM_Type *base`, `tpm_chnl_t chnlNumber`, `uint8_t level`)
Update the edge level selection for a channel.
- void `TPM_SetupInputCapture` (`TPM_Type *base`, `tpm_chnl_t chnlNumber`, `tpm_input_capture_edge_t captureMode`)
Enables capturing an input signal on the channel using the function parameters.
- void `TPM_SetupOutputCompare` (`TPM_Type *base`, `tpm_chnl_t chnlNumber`, `tpm_output_compare_mode_t compareMode`, `uint32_t compareValue`)
Configures the TPM to generate timed pulses.

Interrupt Interface

- void `TPM_EnableInterrupts` (`TPM_Type *base`, `uint32_t mask`)
Enables the selected TPM interrupts.
- void `TPM_DisableInterrupts` (`TPM_Type *base`, `uint32_t mask`)
Disables the selected TPM interrupts.
- `uint32_t` `TPM_GetEnabledInterrupts` (`TPM_Type *base`)
Gets the enabled TPM interrupts.

Status Interface

- static `uint32_t` `TPM_GetStatusFlags` (`TPM_Type *base`)
Gets the TPM status flags.
- static void `TPM_ClearStatusFlags` (`TPM_Type *base`, `uint32_t mask`)
Clears the TPM status flags.

Read and write the timer period

- static void `TPM_SetTimerPeriod` (`TPM_Type *base`, `uint32_t ticks`)

Data Structure Documentation

- Sets the timer period in units of ticks.*
- static uint32_t [TPM_GetCurrentTimerCount](#) (TPM_Type *base)
Reads the current timer counting value.

Timer Start and Stop

- static void [TPM_StartTimer](#) (TPM_Type *base, [tpm_clock_source_t](#) clockSource)
Starts the TPM counter.
- static void [TPM_StopTimer](#) (TPM_Type *base)
Stops the TPM counter.

27.3 Data Structure Documentation

27.3.1 struct tpm_chnl_pwm_signal_param_t

Data Fields

- [tpm_chnl_t](#) chnlNumber
TPM channel to configure.
- [tpm_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)...

27.3.1.0.0.45 Field Documentation

27.3.1.0.0.45.1 tpm_chnl_t tpm_chnl_pwm_signal_param_t::chnlNumber

In combined mode (available in some SoC's, this represents the channel pair number

27.3.1.0.0.45.2 uint8_t tpm_chnl_pwm_signal_param_t::dutyCyclePercent

100=always active signal (100% duty cycle)

27.3.2 struct tpm_config_t

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the [TPM_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

- [tpm_clock_prescale_t](#) prescale
Select TPM clock prescale value.
- bool [useGlobalTimeBase](#)

- true: Use of an external global time base is enabled; false: disabled*
- [tpm_trigger_select_t triggerSelect](#)
Input trigger to use for controlling the counter operation.
- [bool enableDoze](#)
true: TPM counter is paused in doze mode; false: TPM counter continues in doze mode
- [bool enableDebugMode](#)
true: TPM counter continues in debug mode; false: TPM counter is paused in debug mode
- [bool enableReloadOnTrigger](#)
true: TPM counter is reloaded on trigger; false: TPM counter not reloaded
- [bool enableStopOnOverflow](#)
true: TPM counter stops after overflow; false: TPM counter continues running after overflow
- [bool enableStartOnTrigger](#)
true: TPM counter only starts when a trigger is detected; false: TPM counter starts immediately

27.4 Enumeration Type Documentation

27.4.1 enum tpm_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

- kTPM_Chnl_0*** TPM channel number 0.
- kTPM_Chnl_1*** TPM channel number 1.
- kTPM_Chnl_2*** TPM channel number 2.
- kTPM_Chnl_3*** TPM channel number 3.
- kTPM_Chnl_4*** TPM channel number 4.
- kTPM_Chnl_5*** TPM channel number 5.
- kTPM_Chnl_6*** TPM channel number 6.
- kTPM_Chnl_7*** TPM channel number 7.

27.4.2 enum tpm_pwm_mode_t

Enumerator

- kTPM_EdgeAlignedPwm*** Edge aligned PWM.
- kTPM_CenterAlignedPwm*** Center aligned PWM.

27.4.3 enum tpm_pwm_level_select_t

Enumerator

- kTPM_NoPwmSignal*** No PWM output on pin.

Enumeration Type Documentation

kTPM_LowTrue Low true pulses.

kTPM_HighTrue High true pulses.

27.4.4 enum tpm_trigger_select_t

This is used for both internal & external trigger sources (external option available in certain SoC's)

Note

The actual trigger options available is SoC-specific.

27.4.5 enum tpm_output_compare_mode_t

Enumerator

kTPM_NoOutputSignal No channel output when counter reaches CnV.

kTPM_ToggleOnMatch Toggle output.

kTPM_ClearOnMatch Clear output.

kTPM_SetOnMatch Set output.

kTPM_HighPulseOutput Pulse output high.

kTPM_LowPulseOutput Pulse output low.

27.4.6 enum tpm_input_capture_edge_t

Enumerator

kTPM_RisingEdge Capture on rising edge only.

kTPM_FallingEdge Capture on falling edge only.

kTPM_RiseAndFallEdge Capture on rising or falling edge.

27.4.7 enum tpm_clock_source_t

Enumerator

kTPM_SystemClock System clock.

kTPM_ExternalClock External clock.

27.4.8 enum tpm_clock_prescale_t

Enumerator

kTPM_Prescale_Divide_1 Divide by 1.
kTPM_Prescale_Divide_2 Divide by 2.
kTPM_Prescale_Divide_4 Divide by 4.
kTPM_Prescale_Divide_8 Divide by 8.
kTPM_Prescale_Divide_16 Divide by 16.
kTPM_Prescale_Divide_32 Divide by 32.
kTPM_Prescale_Divide_64 Divide by 64.
kTPM_Prescale_Divide_128 Divide by 128.

27.4.9 enum tpm_interrupt_enable_t

Enumerator

kTPM_Chnl0InterruptEnable Channel 0 interrupt.
kTPM_Chnl1InterruptEnable Channel 1 interrupt.
kTPM_Chnl2InterruptEnable Channel 2 interrupt.
kTPM_Chnl3InterruptEnable Channel 3 interrupt.
kTPM_Chnl4InterruptEnable Channel 4 interrupt.
kTPM_Chnl5InterruptEnable Channel 5 interrupt.
kTPM_Chnl6InterruptEnable Channel 6 interrupt.
kTPM_Chnl7InterruptEnable Channel 7 interrupt.
kTPM_TimeOverflowInterruptEnable Time overflow interrupt.

27.4.10 enum tpm_status_flags_t

Enumerator

kTPM_Chnl0Flag Channel 0 flag.
kTPM_Chnl1Flag Channel 1 flag.
kTPM_Chnl2Flag Channel 2 flag.
kTPM_Chnl3Flag Channel 3 flag.
kTPM_Chnl4Flag Channel 4 flag.
kTPM_Chnl5Flag Channel 5 flag.
kTPM_Chnl6Flag Channel 6 flag.
kTPM_Chnl7Flag Channel 7 flag.
kTPM_TimeOverflowFlag Time overflow flag.

27.5 Function Documentation

27.5.1 void TPM_Init (TPM_Type * *base*, const tpm_config_t * *config*)

Function Documentation

Note

This API should be called at the beginning of the application using the TPM driver.

Parameters

<i>base</i>	TPM peripheral base address
<i>config</i>	Pointer to user's TPM config structure.

27.5.2 void TPM_Deinit (TPM_Type * *base*)

Parameters

<i>base</i>	TPM peripheral base address
-------------	-----------------------------

27.5.3 void TPM_GetDefaultConfig (tpm_config_t * *config*)

The default values are:

```
* config->prescale = kTPM_Prescale_Divide_1;
* config->useGlobalTimeBase = false;
* config->dozeEnable = false;
* config->dbgMode = false;
* config->enableReloadOnTrigger = false;
* config->enableStopOnOverflow = false;
* config->enableStartOnTrigger = false;
*#if FSL_FEATURE_TPM_HAS_PAUSE_COUNTER_ON_TRIGGER
* config->enablePauseOnTrigger = false;
*#endif
* config->triggerSelect = kTPM_Trigger_Select_0;
*#if FSL_FEATURE_TPM_HAS_EXTERNAL_TRIGGER_SELECTION
* config->triggerSource = kTPM_TriggerSource_External;
*#endif
*
```

Parameters

<i>config</i>	Pointer to user's TPM config structure.
---------------	---

27.5.4 status_t TPM_SetupPwm (TPM_Type * *base*, const tpm_chnl_pwm_signal_param_t * *chnlParams*, uint8_t *numOfChnls*, tpm_pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz*)

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

<i>base</i>	TPM 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 tpm_pwm_mode_t
<i>pwmFreq_Hz</i>	PWM signal frequency in Hz
<i>srcClock_Hz</i>	TPM counter clock in Hz

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

27.5.5 void TPM_UpdatePwmDutycycle (TPM_Type * *base*, tpm_chnl_t *chnlNumber*, tpm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent*)

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel 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, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

27.5.6 void TPM_UpdateChnlEdgeLevelSelect (TPM_Type * *base*, tpm_chnl_t *chnlNumber*, uint8_t *level*)

Parameters

<i>base</i>	TPM peripheral base address
-------------	-----------------------------

Function Documentation

<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 appropriate SoC reference manual for details about this field.

27.5.7 void TPM_SetupInputCapture (TPM_Type * *base*, tpm_chnl_t *chnlNumber*, tpm_input_capture_edge_t *captureMode*)

When the edge specified in the *captureMode* argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number
<i>captureMode</i>	Specifies which edge to capture

27.5.8 void TPM_SetupOutputCompare (TPM_Type * *base*, tpm_chnl_t *chnlNumber*, tpm_output_compare_mode_t *compareMode*, uint32_t *compareValue*)

When the TPM 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>	TPM 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.

27.5.9 void TPM_EnableInterrupts (TPM_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	TPM peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t

27.5.10 void TPM_DisableInterrupts (TPM_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	TPM peripheral base address
<i>mask</i>	The interrupts to disable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t

27.5.11 uint32_t TPM_GetEnabledInterrupts (TPM_Type * *base*)

Parameters

<i>base</i>	TPM peripheral base address
-------------	-----------------------------

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [tpm_interrupt_enable_t](#)

27.5.12 static uint32_t TPM_GetStatusFlags (TPM_Type * *base*) [inline], [static]

Parameters

<i>base</i>	TPM peripheral base address
-------------	-----------------------------

Returns

The status flags. This is the logical OR of members of the enumeration [tpm_status_flags_t](#)

27.5.13 static void TPM_ClearStatusFlags (TPM_Type * *base*, uint32_t *mask*) [inline], [static]

Function Documentation

Parameters

<i>base</i>	TPM peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration tpm_status_flags_t

27.5.14 static void TPM_SetTimerPeriod (TPM_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 TPM module as a timer. Do not mix usage of this API with TPM'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>	TPM peripheral base address
<i>ticks</i>	A timer period in units of ticks, which should be equal or greater than 1.

27.5.15 static uint32_t TPM_GetCurrentTimerCount (TPM_Type * *base*) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note

Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

<i>base</i>	TPM peripheral base address
-------------	-----------------------------

Returns

The current counter value in ticks

27.5.16 `static void TPM_StartTimer (TPM_Type * base, tpm_clock_source_t
clockSource) [inline], [static]`

Function Documentation

Parameters

<i>base</i>	TPM peripheral base address
<i>clockSource</i>	TPM clock source; once clock source is set the counter will start running

27.5.17 static void TPM_StopTimer (TPM_Type * *base*) [inline], [static]

Parameters

<i>base</i>	TPM peripheral base address
-------------	-----------------------------



Chapter 28

TSI: Touch Sensing Input

28.1 Overview

Modules

- [TSIv4 Driver](#)

28.2 TSIv4 Driver

28.2.1 Overview

The MCUXpresso SDK provides driver for the Touch Sensing Input (TSI) module of MCUXpresso SDK devices.

28.2.2 Typical use case

28.2.2.1 TSI Operation

```
TSI_Init(TSI0);
TSI_Configure(TSI0, &user_config);
TSI_SetMeasuredChannelNumber(TSI0, channelMask);
TSI_EnableInterrupts(TSI0, kTSI_GlobalInterruptEnable |
    kTSI_EndOfScanInterruptEnable);

TSI_EnableSoftwareTriggerScan(TSI0);
TSI_EnableModule(TSI0);
while(1)
{
    TSI_StartSoftwareTrigger(TSI0);
    TSI_GetCounter(TSI0);
}
```

Data Structures

- struct `tsi_calibration_data_t`
TSI calibration data storage. [More...](#)
- struct `tsi_config_t`
TSI configuration structure. [More...](#)

Macros

- #define `ALL_FLAGS_MASK` (TSI_GENCS_EOSF_MASK | TSI_GENCS_OUTRGF_MASK)
TSI status flags macro collection.
- #define `TSI_V4_EXTCHRG_RESISTOR_BIT_SHIFT` TSI_GENCS_EXTCHRG_SHIFT
resistor bit shift in EXTCHRG bit-field
- #define `TSI_V4_EXTCHRG_FILTER_BITS_SHIFT` (1U + TSI_GENCS_EXTCHRG_SHIFT)
filter bits shift in EXTCHRG bit-field
- #define `TSI_V4_EXTCHRG_RESISTOR_BIT_CLEAR` ((uint32_t)((~(ALL_FLAGS_MASK | TSI_GENCS_EXTCHRG_MASK)) | (3U << TSI_V4_EXTCHRG_FILTER_BITS_SHIFT)))
macro of clearing the resistor bit in EXTCHRG bit-field
- #define `TSI_V4_EXTCHRG_FILTER_BITS_CLEAR` ((uint32_t)((~(ALL_FLAGS_MASK | TSI_GENCS_EXTCHRG_MASK)) | (1U << TSI_V4_EXTCHRG_RESISTOR_BIT_SHIFT)))
macro of clearing the filter bits in EXTCHRG bit-field

Enumerations

- enum `tsi_n_consecutive_scans_t` {
`kTSI_ConsecutiveScansNumber_1time` = 0U,
`kTSI_ConsecutiveScansNumber_2time` = 1U,
`kTSI_ConsecutiveScansNumber_3time` = 2U,
`kTSI_ConsecutiveScansNumber_4time` = 3U,
`kTSI_ConsecutiveScansNumber_5time` = 4U,
`kTSI_ConsecutiveScansNumber_6time` = 5U,
`kTSI_ConsecutiveScansNumber_7time` = 6U,
`kTSI_ConsecutiveScansNumber_8time` = 7U,
`kTSI_ConsecutiveScansNumber_9time` = 8U,
`kTSI_ConsecutiveScansNumber_10time` = 9U,
`kTSI_ConsecutiveScansNumber_11time` = 10U,
`kTSI_ConsecutiveScansNumber_12time` = 11U,
`kTSI_ConsecutiveScansNumber_13time` = 12U,
`kTSI_ConsecutiveScansNumber_14time` = 13U,
`kTSI_ConsecutiveScansNumber_15time` = 14U,
`kTSI_ConsecutiveScansNumber_16time` = 15U,
`kTSI_ConsecutiveScansNumber_17time` = 16U,
`kTSI_ConsecutiveScansNumber_18time` = 17U,
`kTSI_ConsecutiveScansNumber_19time` = 18U,
`kTSI_ConsecutiveScansNumber_20time` = 19U,
`kTSI_ConsecutiveScansNumber_21time` = 20U,
`kTSI_ConsecutiveScansNumber_22time` = 21U,
`kTSI_ConsecutiveScansNumber_23time` = 22U,
`kTSI_ConsecutiveScansNumber_24time` = 23U,
`kTSI_ConsecutiveScansNumber_25time` = 24U,
`kTSI_ConsecutiveScansNumber_26time` = 25U,
`kTSI_ConsecutiveScansNumber_27time` = 26U,
`kTSI_ConsecutiveScansNumber_28time` = 27U,
`kTSI_ConsecutiveScansNumber_29time` = 28U,
`kTSI_ConsecutiveScansNumber_30time` = 29U,
`kTSI_ConsecutiveScansNumber_31time` = 30U,
`kTSI_ConsecutiveScansNumber_32time` = 31U }
TSI number of scan intervals for each electrode.
- enum `tsi_electrode_osc_prescaler_t` {
`kTSI_ElecOscPrescaler_1div` = 0U,
`kTSI_ElecOscPrescaler_2div` = 1U,
`kTSI_ElecOscPrescaler_4div` = 2U,
`kTSI_ElecOscPrescaler_8div` = 3U,
`kTSI_ElecOscPrescaler_16div` = 4U,
`kTSI_ElecOscPrescaler_32div` = 5U,
`kTSI_ElecOscPrescaler_64div` = 6U,
`kTSI_ElecOscPrescaler_128div` = 7U }

- TSI electrode oscillator prescaler.*
 - enum `tsi_analog_mode_t` {
 `kTSI_AnalogModeSel_Capacitive` = 0U,
 `kTSI_AnalogModeSel_NoiseNoFreqLim` = 4U,
 `kTSI_AnalogModeSel_NoiseFreqLim` = 8U,
 `kTSI_AnalogModeSel_AutoNoise` = 12U }
- TSI analog mode select.*
 - enum `tsi_reference_osc_charge_current_t` {
 `kTSI_RefOscChargeCurrent_500nA` = 0U,
 `kTSI_RefOscChargeCurrent_1uA` = 1U,
 `kTSI_RefOscChargeCurrent_2uA` = 2U,
 `kTSI_RefOscChargeCurrent_4uA` = 3U,
 `kTSI_RefOscChargeCurrent_8uA` = 4U,
 `kTSI_RefOscChargeCurrent_16uA` = 5U,
 `kTSI_RefOscChargeCurrent_32uA` = 6U,
 `kTSI_RefOscChargeCurrent_64uA` = 7U }
- TSI Reference oscillator charge and discharge current select.*
 - enum `tsi_osc_voltage_rails_t` {
 `kTSI_OscVolRailsOption_0` = 0U,
 `kTSI_OscVolRailsOption_1` = 1U,
 `kTSI_OscVolRailsOption_2` = 2U,
 `kTSI_OscVolRailsOption_3` = 3U }
- TSI oscilator's voltage rails.*
 - enum `tsi_external_osc_charge_current_t` {
 `kTSI_ExtOscChargeCurrent_500nA` = 0U,
 `kTSI_ExtOscChargeCurrent_1uA` = 1U,
 `kTSI_ExtOscChargeCurrent_2uA` = 2U,
 `kTSI_ExtOscChargeCurrent_4uA` = 3U,
 `kTSI_ExtOscChargeCurrent_8uA` = 4U,
 `kTSI_ExtOscChargeCurrent_16uA` = 5U,
 `kTSI_ExtOscChargeCurrent_32uA` = 6U,
 `kTSI_ExtOscChargeCurrent_64uA` = 7U }
- TSI External oscillator charge and discharge current select.*
 - enum `tsi_series_resistor_t` {
 `kTSI_SeriesResistance_32k` = 0U,
 `kTSI_SeriesResistance_187k` = 1U }
- TSI series resistance RS value select.*
 - enum `tsi_filter_bits_t` {
 `kTSI_FilterBits_3` = 0U,
 `kTSI_FilterBits_2` = 1U,
 `kTSI_FilterBits_1` = 2U,
 `kTSI_FilterBits_0` = 3U }
- TSI series filter bits select.*
 - enum `tsi_status_flags_t` {
 `kTSI_EndOfScanFlag` = TSI_GENCS_EOSF_MASK,
 `kTSI_OutOfRangeFlag` = TSI_GENCS_OUTRGF_MASK }

- TSI status flags.*
- enum `tsi_interrupt_enable_t` {
`kTSI_GlobalInterruptEnable` = 1U,
`kTSI_OutOfRangeInterruptEnable` = 2U,
`kTSI_EndOfScanInterruptEnable` = 4U }
- TSI feature interrupt source.*

Functions

- void `TSI_Init` (TSI_Type *base, const `tsi_config_t` *config)
Initializes hardware.
- void `TSI_Deinit` (TSI_Type *base)
De-initializes hardware.
- void `TSI_GetNormalModeDefaultConfig` (`tsi_config_t` *userConfig)
Gets the TSI normal mode user configuration structure.
- void `TSI_GetLowPowerModeDefaultConfig` (`tsi_config_t` *userConfig)
Gets the TSI low power mode default user configuration structure.
- void `TSI_Calibrate` (TSI_Type *base, `tsi_calibration_data_t` *calBuff)
Hardware calibration.
- void `TSI_EnableInterrupts` (TSI_Type *base, uint32_t mask)
Enables the TSI interrupt requests.
- void `TSI_DisableInterrupts` (TSI_Type *base, uint32_t mask)
Disables the TSI interrupt requests.
- static uint32_t `TSI_GetStatusFlags` (TSI_Type *base)
Gets an interrupt flag.
- void `TSI_ClearStatusFlags` (TSI_Type *base, uint32_t mask)
Clears the interrupt flag.
- static uint32_t `TSI_GetScanTriggerMode` (TSI_Type *base)
Gets the TSI scan trigger mode.
- static bool `TSI_IsScanInProgress` (TSI_Type *base)
Gets the scan in progress flag.
- static void `TSI_SetElectrodeOSCPrescaler` (TSI_Type *base, `tsi_electrode_osc_prescaler_t` prescaler)
Sets the prescaler.
- static void `TSI_SetNumberOfScans` (TSI_Type *base, `tsi_n_consecutive_scans_t` number)
Sets the number of scans (NSCN).
- static void `TSI_EnableModule` (TSI_Type *base, bool enable)
Enables/disables the TSI module.
- static void `TSI_EnableLowPower` (TSI_Type *base, bool enable)
Sets the TSI low power STOP mode as enabled or disabled.
- static void `TSI_EnableHardwareTriggerScan` (TSI_Type *base, bool enable)
Enables/disables the hardware trigger scan.
- static void `TSI_StartSoftwareTrigger` (TSI_Type *base)
Starts a software trigger measurement (triggers a new measurement).
- static void `TSI_SetMeasuredChannelNumber` (TSI_Type *base, uint8_t channel)
Sets the the measured channel number.
- static uint8_t `TSI_GetMeasuredChannelNumber` (TSI_Type *base)
Gets the current measured channel number.
- static void `TSI_EnableDmaTransfer` (TSI_Type *base, bool enable)

TSIv4 Driver

- Enables/disables the DMA transfer.*
- static uint16_t **TSI_GetCounter** (TSI_Type *base)
Gets the conversion counter value.
- static void **TSI_SetLowThreshold** (TSI_Type *base, uint16_t low_threshold)
Sets the TSI wake-up channel low threshold.
- static void **TSI_SetHighThreshold** (TSI_Type *base, uint16_t high_threshold)
Sets the TSI wake-up channel high threshold.
- static void **TSI_SetAnalogMode** (TSI_Type *base, tsi_analog_mode_t mode)
Sets the analog mode of the TSI module.
- static uint8_t **TSI_GetNoiseModeResult** (TSI_Type *base)
Gets the noise mode result of the TSI module.
- static void **TSI_SetReferenceChargeCurrent** (TSI_Type *base, tsi_reference_osc_charge_current_t current)
Sets the reference oscillator charge current.
- static void **TSI_SetElectrodeChargeCurrent** (TSI_Type *base, tsi_external_osc_charge_current_t current)
Sets the external electrode charge current.
- static void **TSI_SetOscVoltageRails** (TSI_Type *base, tsi_osc_voltage_rails_t dvolt)
Sets the oscillator's voltage rails.
- static void **TSI_SetElectrodeSeriesResistor** (TSI_Type *base, tsi_series_resistor_t resistor)
Sets the electrode series resistance value in EXTCHRG[0] bit.
- static void **TSI_SetFilterBits** (TSI_Type *base, tsi_filter_bits_t filter)
Sets the electrode filter bits value in EXTCHRG[2:1] bits.

Driver version

- #define **FSL_TSI_DRIVER_VERSION** (MAKE_VERSION(2, 1, 2))
TSI driver version.

28.2.3 Data Structure Documentation

28.2.3.1 struct tsi_calibration_data_t

Data Fields

- uint16_t **calibratedData** [FSL_FEATURE_TSI_CHANNEL_COUNT]
TSI calibration data storage buffer.

28.2.3.2 struct tsi_config_t

This structure contains the settings for the most common TSI configurations including the TSI module charge currents, number of scans, thresholds, and so on.

Data Fields

- uint16_t **thresh**

- *High threshold.*
uint16_t [thresl](#)
- *Low threshold.*
tsi_electrode_osc_prescaler_t [prescaler](#)
- *Prescaler.*
tsi_external_osc_charge_current_t [extchrg](#)
- *Electrode charge current.*
tsi_reference_osc_charge_current_t [refchrg](#)
- *Reference charge current.*
tsi_n_consecutive_scans_t [nscn](#)
- *Number of scans.*
tsi_analog_mode_t [mode](#)
- *TSI mode of operation.*
tsi_osc_voltage_rails_t [dvolt](#)
- *Oscillator's voltage rails.*
tsi_series_resistor_t [resistor](#)
- *Series resistance value.*
tsi_filter_bits_t [filter](#)
- *Noise mode filter bits.*

28.2.3.2.0.46 Field Documentation

28.2.3.2.0.46.1 uint16_t [tsi_config_t::thresh](#)

28.2.3.2.0.46.2 uint16_t [tsi_config_t::thresl](#)

28.2.3.2.0.46.3 tsi_n_consecutive_scans_t [tsi_config_t::nscn](#)

28.2.3.2.0.46.4 tsi_analog_mode_t [tsi_config_t::mode](#)

28.2.3.2.0.46.5 tsi_osc_voltage_rails_t [tsi_config_t::dvolt](#)

28.2.4 Enumeration Type Documentation

28.2.4.1 enum [tsi_n_consecutive_scans_t](#)

These constants define the tsi number of consecutive scans in a TSI instance for each electrode.

Enumerator

<i>kTSI_ConsecutiveScansNumber_1time</i>	Once per electrode.
<i>kTSI_ConsecutiveScansNumber_2time</i>	Twice per electrode.
<i>kTSI_ConsecutiveScansNumber_3time</i>	3 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_4time</i>	4 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_5time</i>	5 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_6time</i>	6 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_7time</i>	7 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_8time</i>	8 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_9time</i>	9 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_10time</i>	10 times consecutive scan

<i>kTSI_ConsecutiveScansNumber_11time</i>	11 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_12time</i>	12 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_13time</i>	13 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_14time</i>	14 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_15time</i>	15 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_16time</i>	16 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_17time</i>	17 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_18time</i>	18 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_19time</i>	19 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_20time</i>	20 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_21time</i>	21 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_22time</i>	22 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_23time</i>	23 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_24time</i>	24 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_25time</i>	25 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_26time</i>	26 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_27time</i>	27 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_28time</i>	28 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_29time</i>	29 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_30time</i>	30 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_31time</i>	31 times consecutive scan
<i>kTSI_ConsecutiveScansNumber_32time</i>	32 times consecutive scan

28.2.4.2 enum tsi_electrode_osc_prescaler_t

These constants define the TSI electrode oscillator prescaler in a TSI instance.

Enumerator

<i>kTSI_ElecOscPrescaler_1div</i>	Electrode oscillator frequency divided by 1.
<i>kTSI_ElecOscPrescaler_2div</i>	Electrode oscillator frequency divided by 2.
<i>kTSI_ElecOscPrescaler_4div</i>	Electrode oscillator frequency divided by 4.
<i>kTSI_ElecOscPrescaler_8div</i>	Electrode oscillator frequency divided by 8.
<i>kTSI_ElecOscPrescaler_16div</i>	Electrode oscillator frequency divided by 16.
<i>kTSI_ElecOscPrescaler_32div</i>	Electrode oscillator frequency divided by 32.
<i>kTSI_ElecOscPrescaler_64div</i>	Electrode oscillator frequency divided by 64.
<i>kTSI_ElecOscPrescaler_128div</i>	Electrode oscillator frequency divided by 128.

28.2.4.3 enum tsi_analog_mode_t

Set up TSI analog modes in a TSI instance.

Enumerator

<i>kTSI_AnalogModeSel_Capacitive</i>	Active TSI capacitive sensing mode.
--------------------------------------	-------------------------------------

kTSI_AnalogModeSel_NoiseNoFreqLim Single threshold noise detection mode with no freq. limitation.

kTSI_AnalogModeSel_NoiseFreqLim Single threshold noise detection mode with freq. limitation.

kTSI_AnalogModeSel_AutoNoise Active TSI analog in automatic noise detection mode.

28.2.4.4 enum tsi_reference_osc_charge_current_t

These constants define the TSI Reference oscillator charge current select in a TSI (REFCHRG) instance.

Enumerator

kTSI_RefOscChargeCurrent_500nA Reference oscillator charge current is 500 μ A.

kTSI_RefOscChargeCurrent_1uA Reference oscillator charge current is 1 μ A.

kTSI_RefOscChargeCurrent_2uA Reference oscillator charge current is 2 μ A.

kTSI_RefOscChargeCurrent_4uA Reference oscillator charge current is 4 μ A.

kTSI_RefOscChargeCurrent_8uA Reference oscillator charge current is 8 μ A.

kTSI_RefOscChargeCurrent_16uA Reference oscillator charge current is 16 μ A.

kTSI_RefOscChargeCurrent_32uA Reference oscillator charge current is 32 μ A.

kTSI_RefOscChargeCurrent_64uA Reference oscillator charge current is 64 μ A.

28.2.4.5 enum tsi_osc_voltage_rails_t

These bits indicate the oscillator's voltage rails.

Enumerator

kTSI_OscVolRailsOption_0 DVOLT value option 0, the value may differ on different platforms.

kTSI_OscVolRailsOption_1 DVOLT value option 1, the value may differ on different platforms.

kTSI_OscVolRailsOption_2 DVOLT value option 2, the value may differ on different platforms.

kTSI_OscVolRailsOption_3 DVOLT value option 3, the value may differ on different platforms.

28.2.4.6 enum tsi_external_osc_charge_current_t

These bits indicate the electrode oscillator charge and discharge current value in TSI (EXTCHRG) instance.

Enumerator

kTSI_ExtOscChargeCurrent_500nA External oscillator charge current is 500 μ A.

kTSI_ExtOscChargeCurrent_1uA External oscillator charge current is 1 μ A.

kTSI_ExtOscChargeCurrent_2uA External oscillator charge current is 2 μ A.

kTSI_ExtOscChargeCurrent_4uA External oscillator charge current is 4 μ A.

TSIv4 Driver

kTSI_ExtOscChargeCurrent_8uA External oscillator charge current is 8 μ A.
kTSI_ExtOscChargeCurrent_16uA External oscillator charge current is 16 μ A.
kTSI_ExtOscChargeCurrent_32uA External oscillator charge current is 32 μ A.
kTSI_ExtOscChargeCurrent_64uA External oscillator charge current is 64 μ A.

28.2.4.7 enum tsi_series_resistor_t

These bits indicate the electrode RS series resistance for the noise mode in TSI (EXTCHRG) instance.

Enumerator

kTSI_SeriesResistance_32k Series Resistance is 32 kilo ohms.
kTSI_SeriesResistance_187k Series Resistance is 18 7 kilo ohms.

28.2.4.8 enum tsi_filter_bits_t

These bits indicate the count of the filter bits in TSI noise mode EXTCHRG[2:1] bits

Enumerator

kTSI_FilterBits_3 3 filter bits, 8 peaks increments the cnt+1
kTSI_FilterBits_2 2 filter bits, 4 peaks increments the cnt+1
kTSI_FilterBits_1 1 filter bits, 2 peaks increments the cnt+1
kTSI_FilterBits_0 no filter bits, 1 peak increments the cnt+1

28.2.4.9 enum tsi_status_flags_t

Enumerator

kTSI_EndOfScanFlag End-Of-Scan flag.
kTSI_OutOfRangeFlag Out-Of-Range flag.

28.2.4.10 enum tsi_interrupt_enable_t

Enumerator

kTSI_GlobalInterruptEnable TSI module global interrupt.
kTSI_OutOfRangeInterruptEnable Out-Of-Range interrupt.
kTSI_EndOfScanInterruptEnable End-Of-Scan interrupt.

28.2.5 Function Documentation

28.2.5.1 void TSI_Init (TSI_Type * *base*, const tsi_config_t * *config*)

Initializes the peripheral to the targeted state specified by parameter configuration, such as sets prescalers, number of scans, clocks, delta voltage series resistor, filter bits, reference, and electrode charge current and threshold.

Parameters

<i>base</i>	TSI peripheral base address.
<i>config</i>	Pointer to TSI module configuration structure.

Returns

none

28.2.5.2 void TSI_Deinit (TSI_Type * *base*)

De-initializes the peripheral to default state.

Parameters

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

Returns

none

28.2.5.3 void TSI_GetNormalModeDefaultConfig (tsi_config_t * *userConfig*)

This interface sets userConfig structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to these values:

```
userConfig->prescaler = kTSI_ElecOscPrescaler_2div;
userConfig->extchrg = kTSI_ExtOscChargeCurrent_500nA;
userConfig->refchrg = kTSI_RefOscChargeCurrent_4uA;
userConfig->nscn = kTSI_ConsecutiveScansNumber_10time;
userConfig->mode = kTSI_AnalogModeSel_Capacitive;
userConfig->dvolt = kTSI_OscVolRailsOption_0;
userConfig->thresh = 0U;
userConfig->thresl = 0U;
```

TSIv4 Driver

Parameters

<i>userConfig</i>	Pointer to the TSI user configuration structure.
-------------------	--

28.2.5.4 void TSI_GetLowPowerModeDefaultConfig (tsi_config_t * *userConfig*)

This interface sets userConfig structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to these values:

```
userConfig->prescaler = kTSI_ElecOscPrescaler_2div;  
userConfig->extchrg = kTSI_ExtOscChargeCurrent_500nA;  
userConfig->refchrg = kTSI_RefOscChargeCurrent_4uA;  
userConfig->nscn = kTSI_ConsecutiveScansNumber_10time;  
userConfig->mode = kTSI_AnalogModeSel_Capacitive;  
userConfig->dvolt = kTSI_OscVolRailsOption_0;  
userConfig->thresh = 400U;  
userConfig->thresl = 0U;
```

Parameters

<i>userConfig</i>	Pointer to the TSI user configuration structure.
-------------------	--

28.2.5.5 void TSI_Calibrate (TSI_Type * *base*, tsi_calibration_data_t * *calBuff*)

Calibrates the peripheral to fetch the initial counter value of the enabled electrodes. This API is mostly used at initial application setup. Call this function after the [TSI_Init](#) API and use the calibrated counter values to set up applications (such as to determine under which counter value we can confirm a touch event occurs).

Parameters

<i>base</i>	TSI peripheral base address.
<i>calBuff</i>	Data buffer that store the calibrated counter value.

Returns

none

28.2.5.6 void TSI_EnableInterrupts (TSI_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	TSI peripheral base address.
<i>mask</i>	interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"> • kTSI_GlobalInterruptEnable • kTSI_EndOfScanInterruptEnable • kTSI_OutOfRangeInterruptEnable

28.2.5.7 void TSI_DisableInterrupts (TSI_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	TSI peripheral base address.
<i>mask</i>	interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"> • kTSI_GlobalInterruptEnable • kTSI_EndOfScanInterruptEnable • kTSI_OutOfRangeInterruptEnable

28.2.5.8 static uint32_t TSI_GetStatusFlags (TSI_Type * *base*) [inline], [static]

This function gets the TSI interrupt flags.

Parameters

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

Returns

The mask of these status flags combination.

28.2.5.9 void TSI_ClearStatusFlags (TSI_Type * *base*, uint32_t *mask*)

This function clears the TSI interrupt flag, automatically cleared flags can't be cleared by this function.

TSIv4 Driver

Parameters

<i>base</i>	TSI peripheral base address.
<i>mask</i>	The status flags to clear.

28.2.5.10 `static uint32_t TSI_GetScanTriggerMode (TSI_Type * base) [inline], [static]`

Parameters

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

Returns

Scan trigger mode.

28.2.5.11 `static bool TSI_IsScanInProgress (TSI_Type * base) [inline], [static]`

Parameters

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

Returns

True - scan is in progress. False - scan is not in progress.

28.2.5.12 `static void TSI_SetElectrodeOSCPrescaler (TSI_Type * base,
tsi_electrode_osc_prescaler_t prescaler) [inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>prescaler</i>	Prescaler value.

Returns

none.

28.2.5.13 `static void TSI_SetNumberOfScans (TSI_Type * base,
tsi_n_consecutive_scans_t number) [inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>number</i>	Number of scans.

Returns

none.

28.2.5.14 static void TSI_EnableModule (TSI_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose whether to enable or disable module; <ul style="list-style-type: none"> • true Enable TSI module; • false Disable TSI module;

Returns

none.

28.2.5.15 static void TSI_EnableLowPower (TSI_Type * *base*, bool *enable*) [inline], [static]

This enables the TSI module function in low power modes.

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose to enable or disable STOP mode. <ul style="list-style-type: none"> • true Enable module in STOP mode; • false Disable module in STOP mode;

Returns

none.

28.2.5.16 `static void TSI_EnableHardwareTriggerScan (TSI_Type * base, bool enable)`
`[inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose to enable hardware trigger or software trigger scan. <ul style="list-style-type: none"> • true Enable hardware trigger scan; • false Enable software trigger scan;

Returns

none.

28.2.5.17 `static void TSI_StartSoftwareTrigger (TSI_Type * base) [inline], [static]`

Parameters

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

Returns

none.

28.2.5.18 `static void TSI_SetMeasuredChannelNumber (TSI_Type * base, uint8_t channel) [inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>channel</i>	Channel number 0 ... 15.

Returns

none.

28.2.5.19 `static uint8_t TSI_GetMeasuredChannelNumber (TSI_Type * base) [inline], [static]`

TSIv4 Driver

Parameters

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

Returns

uint8_t Channel number 0 ... 15.

28.2.5.20 static void TSI_EnableDmaTransfer (TSI_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose to enable DMA transfer or not. <ul style="list-style-type: none">• true Enable DMA transfer;• false Disable DMA transfer;

Returns

none.

28.2.5.21 static uint16_t TSI_GetCounter (TSI_Type * *base*) [inline], [static]

Parameters

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

Returns

Accumulated scan counter value ticked by the reference clock.

28.2.5.22 static void TSI_SetLowThreshold (TSI_Type * *base*, uint16_t *low_threshold*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>low_threshold</i>	Low counter threshold.

Returns

none.

28.2.5.23 `static void TSI_SetHighThreshold (TSI_Type * base, uint16_t high_threshold)`
[inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>high_threshold</i>	High counter threshold.

Returns

none.

28.2.5.24 `static void TSI_SetAnalogMode (TSI_Type * base, tsi_analog_mode_t mode)`
[inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>mode</i>	Mode value.

Returns

none.

28.2.5.25 `static uint8_t TSI_GetNoiseModeResult (TSI_Type * base)` **[inline],**
[static]

TSIv4 Driver

Parameters

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

Returns

Value of the GENCS[MODE] bit-fields.

28.2.5.26 `static void TSI_SetReferenceChargeCurrent (TSI_Type * base,
tsi_reference_osc_charge_current_t current) [inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>current</i>	The reference oscillator charge current.

Returns

none.

28.2.5.27 `static void TSI_SetElectrodeChargeCurrent (TSI_Type * base,
tsi_external_osc_charge_current_t current) [inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>current</i>	External electrode charge current.

Returns

none.

28.2.5.28 `static void TSI_SetOscVoltageRails (TSI_Type * base, tsi_osc_voltage_rails_t
dvolt) [inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>dvolt</i>	The voltage rails.

Returns

none.

28.2.5.29 `static void TSI_SetElectrodeSeriesResistor (TSI_Type * base,
tsi_series_resistor_t resistor) [inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>resistor</i>	Series resistance.

Returns

none.

28.2.5.30 `static void TSI_SetFilterBits (TSI_Type * base, tsi_filter_bits_t filter)
[inline], [static]`

Parameters

<i>base</i>	TSI peripheral base address.
<i>filter</i>	Series resistance.

Returns

none.



Chapter 29

UART: Universal Asynchronous Receiver/Transmitter Driver

29.1 Overview

Modules

- [UART DMA Driver](#)
- [UART Driver](#)
- [UART FreeRTOS Driver](#)
- [UART eDMA Driver](#)

29.2 UART Driver

29.2.1 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_RxRingBufferOverflow`. 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.

29.2.2 Typical use case

29.2.2.1 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);
}

```

29.2.2.2 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)
{
}

// ...
}
```

29.2.2.3 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)
    {
    }
}

// ...
}

```

29.2.2.4 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_uartTxDmaHandle, DMA0, UART_TX_DMA_CHANNEL);
DMA_CreateHandle(&g_uartRxDmaHandle, DMA0, UART_RX_DMA_CHANNEL);

UART_TransferCreateHandleDMA(UART1, &g_uartHandle, UART_UserCallback, NULL,
    &g_uartTxDmaHandle, &g_uartRxDmaHandle);

// 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_RxRingBufferOverflow` = MAKE_STATUS(kStatusGroup_UART, 8),
 - `kStatus_UART_RxHardwareOverflow` = 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_LBKDIE_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_RxOverflowInterruptEnable` = (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) }

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 }
    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.
- 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.

UART Driver

- 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.

29.2.3 Data Structure Documentation

29.2.3.1 struct uart_config_t

Data Fields

- uint32_t [baudRate_Bps](#)
UART baud rate.
- [uart_parity_mode_t](#) [parityMode](#)
Parity mode, disabled (default), even, odd.
- [uart_stop_bit_count_t](#) [stopBitCount](#)
Number of stop bits, 1 stop bit (default) or 2 stop bits.
- bool [enableTx](#)
Enable TX.
- bool [enableRx](#)
Enable RX.

29.2.3.2 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.

29.2.3.2.0.47 Field Documentation

29.2.3.2.0.47.1 uint8_t* uart_transfer_t::data

29.2.3.2.0.47.2 size_t uart_transfer_t::dataSize

29.2.3.3 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

29.2.3.3.0.48 Field Documentation

- 29.2.3.3.0.48.1 `uint8_t* volatile uart_handle_t::txData`
- 29.2.3.3.0.48.2 `volatile size_t uart_handle_t::txDataSize`
- 29.2.3.3.0.48.3 `size_t uart_handle_t::txDataSizeAll`
- 29.2.3.3.0.48.4 `uint8_t* volatile uart_handle_t::rxData`
- 29.2.3.3.0.48.5 `volatile size_t uart_handle_t::rxDataSize`
- 29.2.3.3.0.48.6 `size_t uart_handle_t::rxDataSizeAll`
- 29.2.3.3.0.48.7 `uint8_t* uart_handle_t::rxRingBuffer`
- 29.2.3.3.0.48.8 `size_t uart_handle_t::rxRingBufferSize`
- 29.2.3.3.0.48.9 `volatile uint16_t uart_handle_t::rxRingBufferHead`
- 29.2.3.3.0.48.10 `volatile uint16_t uart_handle_t::rxRingBufferTail`
- 29.2.3.3.0.48.11 `uart_transfer_callback_t uart_handle_t::callback`
- 29.2.3.3.0.48.12 `void* uart_handle_t::userData`
- 29.2.3.3.0.48.13 `volatile uint8_t uart_handle_t::txState`

29.2.4 Macro Definition Documentation

- 29.2.4.1 `#define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))`

29.2.5 Typedef Documentation

- 29.2.5.1 `typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)`

29.2.6 Enumeration Type Documentation

29.2.6.1 `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_RxRingBufferOverflow UART RX software ring buffer overrun.
kStatus_UART_RxHardwareOverflow 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.

29.2.6.2 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.

29.2.6.3 enum uart_stop_bit_count_t

Enumerator

kUART_OneStopBit One stop bit.
kUART_TwoStopBit Two stop bits.

29.2.6.4 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_RxOverflowInterruptEnable Receiver overrun interrupt.
kUART_NoiseErrorInterruptEnable Noise error flag interrupt.
kUART_FramingErrorInterruptEnable Framing error flag interrupt.
kUART_ParityErrorInterruptEnable Parity error flag interrupt.

UART Driver

29.2.6.5 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.

29.2.7 Function Documentation

29.2.7.1 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

29.2.7.2 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.
-------------	-------------------------------

29.2.7.3 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.
---------------	-------------------------------------

29.2.7.4 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, 200000000U);
*
```

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.

29.2.7.5 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](#).

29.2.7.6 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 _uart_flags .

Return values

<i>kStatus_UART_Flag- CannotClearManually</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.

29.2.7.7 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 _uart_interrupt_enable .

29.2.7.8 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 _uart_interrupt_enable .

29.2.7.9 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](#).

29.2.7.10 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.

29.2.7.11 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.

29.2.7.12 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.

29.2.7.13 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.

29.2.7.14 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.

29.2.7.15 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.

29.2.7.16 **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.

29.2.7.17 **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.

29.2.7.18 **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.

29.2.7.19 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.

29.2.7.20 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.

29.2.7.21 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.

29.2.7.22 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_TransmissionCompleteFlag` 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 uart_transfer_t .

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.

29.2.7.23 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.

29.2.7.24 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 count;

29.2.7.25 **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 [kStatus_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 uart_transfer_t .
<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.

29.2.7.26 **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.

29.2.7.27 **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> ;

29.2.7.28 **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.

29.2.7.29 **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.

29.3 UART DMA Driver

29.3.1 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.

29.3.2 Data Structure Documentation

29.3.2.1 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.

29.3.2.1.0.49 Field Documentation

29.3.2.1.0.49.1 `UART_Type* uart_dma_handle_t::base`

29.3.2.1.0.49.2 `uart_dma_transfer_callback_t uart_dma_handle_t::callback`

29.3.2.1.0.49.3 `void* uart_dma_handle_t::userData`

29.3.2.1.0.49.4 `size_t uart_dma_handle_t::rxDataSizeAll`

29.3.2.1.0.49.5 `size_t uart_dma_handle_t::txDataSizeAll`

29.3.2.1.0.49.6 `dma_handle_t* uart_dma_handle_t::txDmaHandle`

29.3.2.1.0.49.7 `dma_handle_t* uart_dma_handle_t::rxDmaHandle`

29.3.2.1.0.49.8 `volatile uint8_t uart_dma_handle_t::txState`

29.3.3 Typedef Documentation

29.3.3.1 `typedef void(* uart_dma_transfer_callback_t)(UART_Type *base,
uart_dma_handle_t *handle, status_t status, void *userData)`

29.3.4 Function Documentation

29.3.4.1 `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.

29.3.4.2 **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 uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

29.3.4.3 **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 uart_transfer_t .

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.

29.3.4.4 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.

29.3.4.5 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.

29.3.4.6 `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_NoTransferInProgress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

29.3.4.7 **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_NoTransferInProgress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

29.4 UART eDMA Driver

29.4.1 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.

29.4.2 Data Structure Documentation

29.4.2.1 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.*

29.4.2.1.0.50 Field Documentation

29.4.2.1.0.50.1 `uart_edma_transfer_callback_t uart_edma_handle_t::callback`

29.4.2.1.0.50.2 `void* uart_edma_handle_t::userData`

29.4.2.1.0.50.3 `size_t uart_edma_handle_t::rxDataSizeAll`

29.4.2.1.0.50.4 `size_t uart_edma_handle_t::txDataSizeAll`

29.4.2.1.0.50.5 `edma_handle_t* uart_edma_handle_t::txEdmaHandle`

29.4.2.1.0.50.6 `edma_handle_t* uart_edma_handle_t::rxEdmaHandle`

29.4.2.1.0.50.7 `uint8_t uart_edma_handle_t::nbytes`

29.4.2.1.0.50.8 `volatile uint8_t uart_edma_handle_t::txState`

29.4.3 Typedef Documentation

29.4.3.1 `typedef void(* uart_edma_transfer_callback_t)(UART_Type *base,
uart_edma_handle_t *handle, status_t status, void *userData)`

29.4.4 Function Documentation

29.4.4.1 `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.

29.4.4.2 `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 uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

29.4.4.3 `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 uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_RxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

29.4.4.4 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.

29.4.4.5 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.

29.4.4.6 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_NoTransferInProgress</i>	No send in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

29.4.4.7 **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_NoTransferInProgress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter <code>count</code> ;

29.5 UART FreeRTOS Driver

29.5.1 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.

29.5.2 Data Structure Documentation

29.5.2.1 struct `uart_rtos_config_t`

Data Fields

- `UART_Type` * `base`
UART base address.
- `uint32_t` `srcclk`
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.

29.5.3 Function Documentation

29.5.3.1 `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.

29.5.3.2 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.
---------------	-----------------------

29.5.3.3 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.

29.5.3.4 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 30 Clock Driver

30.1 Overview

The MCUXpresso SDK provides APIs for MCUXpresso SDK devices' clock operation.

30.2 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.

30.3 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 [SPI_CLOCKS](#)
Clock ip name array for SPI.
- #define [SLCD_CLOCKS](#)
Clock ip name array for SLCD.
- #define [PIT_CLOCKS](#)
Clock ip name array for PIT.
- #define [PORT_CLOCKS](#)
Clock ip name array for PORT.
- #define [TSI_CLOCKS](#)
Clock ip name array for TSI.
- #define [DAC_CLOCKS](#)
Clock ip name array for DAC.
- #define [LPTMR_CLOCKS](#)
Clock ip name array for LPTMR.
- #define [ADC16_CLOCKS](#)
Clock ip name array for ADC16.
- #define [DMA_CLOCKS](#)
Clock ip name array for DMA.
- #define [UART0_CLOCKS](#)
Clock ip name array for LPSCI/UART0.
- #define [UART_CLOCKS](#)
Clock ip name array for UART.
- #define [TPM_CLOCKS](#)
Clock ip name array for TPM.
- #define [I2C_CLOCKS](#)
Clock ip name array for I2C.
- #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_FlashClk`,
`kCLOCK_PllFltSelClk`,
`kCLOCK_Er32kClk`,
`kCLOCK_Osc0ErClk`,
`kCLOCK_McgFixedFreqClk`,
`kCLOCK_McgInternalRefClk`,
`kCLOCK_McgFltClk`,
`kCLOCK_McgPll0Clk`,
`kCLOCK_McgExtPllClk`,
`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` }
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_EREFS0_MASK` }
OSCERCLK enable mode.
- enum `mcg_fl_src_t` {
`kMCG_FltSrcExternal`,
`kMCG_FltSrcInternal` }
MCG FLL reference clock source select.
- enum `mcg_irc_mode_t` {
`kMCG_IrcSlow`,
`kMCG_IrcFast` }
MCG internal reference clock select.
- enum `mcg_dmx32_t` {

External clock frequency

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 }

MCG PLLCS select.

- 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_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)
Disable the clock for specific IP.
- static void `CLOCK_SetEr32kClock` (`uint32_t` src)
Set ERCLK32K source.
- static void `CLOCK_SetTpmClock` (`uint32_t` src)
Set TPM clock source.
- static void `CLOCK_SetLpsci0Clock` (`uint32_t` src)
Set LPSCI0 (UART0) 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` 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.

External 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_GetFlashClkFreq](#) (void)
Get the flash clock frequency.
- uint32_t [CLOCK_GetPllFllSelClkFreq](#) (void)
Get the output clock frequency selected by SIM[PLL_FLLSEL].
- 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 fcr-div)
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_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, oscr_config_t const *config)
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(*fllStableDelay)(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 sytem 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.*

30.4 Data Structure Documentation

30.4.1 struct sim_clock_config_t

Data Fields

- uint8_t **pllFllSel**
PLL/FLL selection.
- uint8_t **er32kSrc**
ERCLK32K source selection.
- uint32_t **clkdiv1**
SIM_CLKDIV1.

30.4.1.0.0.51 Field Documentation**30.4.1.0.0.51.1** `uint8_t sim_clock_config_t::pllFilSel`**30.4.1.0.0.51.2** `uint8_t sim_clock_config_t::er32kSrc`**30.4.1.0.0.51.3** `uint32_t sim_clock_config_t::clkdiv1`**30.4.2 struct oscr_config_t****Data Fields**

- `uint8_t enableMode`
OSCERCLK enable mode.

30.4.2.0.0.52 Field Documentation**30.4.2.0.0.52.1** `uint8_t oscr_config_t::enableMode`OR'ed value of `_oscer_enable_mode`.**30.4.3 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 oscrConfig`
Configuration for OSCERCLK.

Data Structure Documentation

30.4.3.0.0.53 Field Documentation

30.4.3.0.0.53.1 `uint32_t osc_config_t::freq`

30.4.3.0.0.53.2 `uint8_t osc_config_t::capLoad`

30.4.3.0.0.53.3 `osc_mode_t osc_config_t::workMode`

30.4.3.0.0.53.4 `oscer_config_t osc_config_t::oscerConfig`

30.4.4 `struct mcg_pll_config_t`

Data Fields

- `uint8_t enableMode`
Enable mode.
- `uint8_t prdiv`
Reference divider PRDIV.
- `uint8_t vdiv`
VCO divider VDIV.

30.4.4.0.0.54 Field Documentation

30.4.4.0.0.54.1 `uint8_t mcg_pll_config_t::enableMode`

OR'ed value of `_mcg_pll_enable_mode`.

30.4.4.0.0.54.2 `uint8_t mcg_pll_config_t::prdiv`

30.4.4.0.0.54.3 `uint8_t mcg_pll_config_t::vdiv`

30.4.5 `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_pll_config_t pll0Config`
MCGPLL0CLK configuration.

30.4.5.0.0.55 Field Documentation

30.4.5.0.0.55.1 `mcg_mode_t mcg_config_t::mcgMode`

30.4.5.0.0.55.2 `uint8_t mcg_config_t::ircclkEnableMode`

30.4.5.0.0.55.3 `mcg_irc_mode_t mcg_config_t::ircs`

30.4.5.0.0.55.4 `uint8_t mcg_config_t::fcrdiv`

30.4.5.0.0.55.5 `uint8_t mcg_config_t::frdiv`

30.4.5.0.0.55.6 `mcg_drs_t mcg_config_t::drs`

30.4.5.0.0.55.7 `mcg_dmx32_t mcg_config_t::dmx32`

30.4.5.0.0.55.8 `mcg_pll_config_t mcg_config_t::pll0Config`

30.5 Macro Definition Documentation

30.5.1 `#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 MCGIRCLK 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.

30.5.2 `#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

Macro Definition Documentation

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.

30.5.3 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 2, 1))

30.5.4 #define DMAMUX_CLOCKS

Value:

```
{
    \
    kCLOCK_Dmamux0 \
}
```

30.5.5 #define RTC_CLOCKS

Value:

```
{
    \
    kCLOCK_Rtc0 \
}
```

30.5.6 #define SAI_CLOCKS

Value:

```
{
    \
    kCLOCK_Sai0 \
}
```

30.5.7 #define SPI_CLOCKS

Value:

```
{
    \
    kCLOCK_Spi0, kCLOCK_Spi1 \
}
```

30.5.8 #define SLCD_CLOCKS

Value:

```
{  
    \kCLOCK_Slcd0 \  
}
```

30.5.9 #define PIT_CLOCKS

Value:

```
{  
    \kCLOCK_Pit0 \  
}
```

30.5.10 #define PORT_CLOCKS

Value:

```
{  
    \kCLOCK_PortA, kCLOCK_PortB, kCLOCK_PortC, kCLOCK_PortD, kCLOCK_PortE \  
}
```

30.5.11 #define TSI_CLOCKS

Value:

```
{  
    \kCLOCK_Tsi0 \  
}
```

30.5.12 #define DAC_CLOCKS

Value:

```
{  
    \kCLOCK_Dac0 \  
}
```

30.5.13 #define LPTMR_CLOCKS

Value:

```
{  
    \kCLOCK_Lptmr0 \  
}
```

30.5.14 #define ADC16_CLOCKS

Value:

```
{  
    \kCLOCK_Adc0 \  
}
```

30.5.15 #define DMA_CLOCKS

Value:

```
{  
    \kCLOCK_Dma0 \  
}
```

30.5.16 #define UART0_CLOCKS

Value:

```
{  
    \kCLOCK_Uart0 \  
}
```

30.5.17 #define UART_CLOCKS

Value:

```
{  
    \kCLOCK_IpInvalid, kCLOCK_Uart1, kCLOCK_Uart2 \  
}
```


30.5.18 #define TPM_CLOCKS

Value:

```
{
    kCLOCK_Tpm0, kCLOCK_Tpm1, kCLOCK_Tpm2 \
}
```

30.5.19 #define I2C_CLOCKS

Value:

```
{
    kCLOCK_I2c0, kCLOCK_I2c1 \
}
```

30.5.20 #define FTF_CLOCKS

Value:

```
{
    kCLOCK_Ftf0 \
}
```

30.5.21 #define CMP_CLOCKS

Value:

```
{
    kCLOCK_Cmp0 \
}
```

30.5.22 #define SYS_CLK kCLOCK_CoreSysClk

30.6 Enumeration Type Documentation

30.6.1 enum clock_name_t

Enumerator

kCLOCK_CoreSysClk Core/system clock.

Enumeration Type Documentation

kCLOCK_PlatClk Platform clock.
kCLOCK_BusClk Bus clock.
kCLOCK_FlashClk Flash clock.
kCLOCK_PllFlSelClk The clock after SIM[PLLFLSEL].
kCLOCK_Er32kClk External reference 32K clock (ERCLK32K)
kCLOCK_Osc0ErClk OSC0 external reference clock (OSC0ERCLK)
kCLOCK_McgFixedFreqClk MCG fixed frequency clock (MCGFFCLK)
kCLOCK_McgInternalRefClk MCG internal reference clock (MCGIRCLK)
kCLOCK_McgFlClk MCGFLLCLK.
kCLOCK_McgPll0Clk MCGPLL0CLK.
kCLOCK_McgExtPllClk EXT_PLLCLK.
kCLOCK_LpoClk LPO clock.

30.6.2 enum clock_usb_src_t

Enumerator

kCLOCK_UsbSrcPll0 Use PLL0.
kCLOCK_UsbSrcExt Use USB_CLKIN.

30.6.3 enum clock_ip_name_t

30.6.4 enum osc_mode_t

Enumerator

kOSC_ModeExt Use an external clock.
kOSC_ModeOscLowPower Oscillator low power.
kOSC_ModeOscHighGain Oscillator high gain.

30.6.5 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

30.6.6 enum _oscer_enable_mode

Enumerator

kOSC_ErClkEnable Enable.

kOSC_ErClkEnableInStop Enable in stop mode.

30.6.7 enum mcg_fl_src_t

Enumerator

kMCG_FlSrcExternal External reference clock is selected.

kMCG_FlSrcInternal The slow internal reference clock is selected.

30.6.8 enum mcg_irc_mode_t

Enumerator

kMCG_IrcSlow Slow internal reference clock selected.

kMCG_IrcFast Fast internal reference clock selected.

30.6.9 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.

30.6.10 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.

Enumeration Type Documentation

30.6.11 enum mcg_pll_ref_src_t

Enumerator

kMCG_PllRefOsc0 Selects OSC0 as PLL reference clock.

kMCG_PllRefOsc1 Selects OSC1 as PLL reference clock.

30.6.12 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.

30.6.13 enum mcg_atm_select_t

Enumerator

kMCG_AtmSel32k 32 kHz Internal Reference Clock selected

kMCG_AtmSel4m 4 MHz Internal Reference Clock selected

30.6.14 enum mcg_oscsel_t

Enumerator

kMCG_OscselOsc Selects System Oscillator (OSCCLK)

kMCG_OscselRtc Selects 32 kHz RTC Oscillator.

30.6.15 enum mcg_pll_clk_select_t

Enumerator

kMCG_PllClkSelPll0 PLL0 output clock is selected.

30.6.16 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.

30.6.17 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.

30.6.18 enum _mcg_status_flags_t

Enumerator

kMCG_Osc0LostFlag OSC0 lost.

kMCG_Osc0InitFlag OSC0 crystal initialized.

kMCG_Pll0LostFlag PLL0 lost.

kMCG_Pll0LockFlag PLL0 locked.

30.6.19 enum _mcg_irclk_enable_mode

Enumerator

kMCG_IrclkEnable MCGIRCLK enable.

kMCG_IrclkEnableInStop MCGIRCLK enable in stop mode.

30.6.20 enum _mcg_pll_enable_mode

Enumerator

kMCG_PllEnableIndependent 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_PllEnableInStop MCGPLLCLK enable in STOP mode.

Function Documentation

30.6.21 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.

30.7 Function Documentation

30.7.1 static void CLOCK_EnableClock (clock_ip_name_t *name*) [inline], [static]

Parameters

<i>name</i>	Which clock to enable, see clock_ip_name_t .
-------------	--

30.7.2 static void CLOCK_DisableClock (clock_ip_name_t *name*) [inline], [static]

Parameters

<i>name</i>	Which clock to disable, see clock_ip_name_t .
-------------	---

30.7.3 static void CLOCK_SetEr32kClock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set ERCLK32K clock source.
------------	---

30.7.4 static void CLOCK_SetTpmClock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set TPM clock source.
------------	------------------------------------

30.7.5 static void CLOCK_SetLpsci0Clock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to setSet LPSCI0 (UART0) clock source.
------------	--

30.7.6 static void CLOCK_SetPllFllSelClock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set PLLFLLSEL clock source.
------------	--

30.7.7 static void CLOCK_SetClkOutClock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set CLKOUT source.
------------	---------------------------------

30.7.8 static void CLOCK_SetRtcClkOutClock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set RTC_CLKOUT source.
------------	-------------------------------------

30.7.9 bool CLOCK_EnableUsbfs0Clock (clock_usb_src_t *src*, uint32_t *freq*)

Function Documentation

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.

30.7.10 static void CLOCK_DisableUsbfs0Clock (void) [inline], [static]

Disable USB FS clock.

30.7.11 static void CLOCK_SetOutDiv (uint32_t outdiv1, uint32_t outdiv4) [inline], [static]

Set the SIM_CLKDIV1[OUTDIV1], SIM_CLKDIV1[OUTDIV4].

Parameters

<i>outdiv1</i>	Clock 1 output divider value.
<i>outdiv4</i>	Clock 4 output divider value.

30.7.12 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 clock_name_t
------------------	-------------------------------------

Returns

Clock frequency value in Hertz

30.7.13 uint32_t CLOCK_GetCoreSysClkFreq (void)

Returns

Clock frequency in Hz.

30.7.14 uint32_t CLOCK_GetPlatClkFreq (void)

Returns

Clock frequency in Hz.

30.7.15 uint32_t CLOCK_GetBusClkFreq (void)

Returns

Clock frequency in Hz.

30.7.16 uint32_t CLOCK_GetFlashClkFreq (void)

Returns

Clock frequency in Hz.

30.7.17 uint32_t CLOCK_GetPIIFlSelClkFreq (void)

Returns

Clock frequency in Hz.

30.7.18 uint32_t CLOCK_GetEr32kClkFreq (void)

Returns

Clock frequency in Hz.

30.7.19 uint32_t CLOCK_GetOsc0ErClkFreq (void)

Returns

Clock frequency in Hz.

30.7.20 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.
---------------	-------------------------------------

30.7.21 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.
---------------	-------------------------------------

30.7.22 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.

30.7.23 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.

30.7.24 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.

30.7.25 `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.

30.7.26 `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.

30.7.27 `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.
---------------	---

30.7.28 `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 _mcg_ircclk_enable_mode .
<i>ircs</i>	MCGIRCLK clock source, choose fast or slow.
<i>fcrdiv</i>	Fast IRC divider setting (FCRDIV).

Return values

<i>kStatus_MCG_Source-Used</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.

30.7.29 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_Source-Used</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.

30.7.30 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

30.7.31 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.
---------------	---

30.7.32 static void CLOCK_DisablePll0 (void) [inline], [static]

This function disables the PLL0 in FLL mode. It should be used together with the [CLOCK_EnablePll0](#).

30.7.33 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 generate.

30.7.34 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.
-------------	----------------------

30.7.35 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.
-------------	----------------------

30.7.36 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](#).

30.7.37 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 lost lock status flags of OSC0 and PLL0.
CLOCK_ClearStatusFlags(kMCG_Osc0LostFlag | kMCG_Pll0LostFlag);
```

Function Documentation

Parameters

<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration _mcg_status_flags_t .
-------------	---

30.7.38 static void OSC_SetExtRefClkConfig (OSC_Type * *base*, oscr_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.

30.7.39 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 _osc_cap_load .

Example:

```
// To enable only 2 pF and 8 pF capacitor load, please use like this.
OSC_SetCapLoad(OSC, kOSC_Cap2P | kOSC_Cap8P);
```

30.7.40 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.
---------------	--

30.7.41 void CLOCK_DeinitOsc0 (void)

This function deinitializes the OSC0.

30.7.42 static void CLOCK_SetXtal0Freq (uint32_t freq) [inline], [static]

Parameters

<i>freq</i>	The XTAL0/EXTAL0 input clock frequency in Hz.
-------------	---

30.7.43 static void CLOCK_SetXtal32Freq (uint32_t freq) [inline], [static]

Parameters

<i>freq</i>	The XTAL32/EXTAL32/RTC_CLKIN input clock frequency in Hz.
-------------	---

30.7.44 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 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.

Function Documentation

<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.

30.7.45 **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](#).

30.7.46 **status_t** CLOCK_SetFeiMode (mcg_dm32_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

<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.

30.7.47 **status_t** CLOCK_SetFeeMode (**uint8_t** *frdiv*, **mcg_dmx32_t** *dmx32*, **mcg_drs_t** *drs*, **void(*)**(**void**) *flStableDelay*)

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>flStableDelay</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.

30.7.48 **status_t** CLOCK_SetFbiMode (**mcg_dmx32_t** *dmx32*, **mcg_drs_t** *drs*, **void(*)**(**void**) *flStableDelay*)

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>flStableDelay</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.

Function Documentation

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.

30.7.49 **status_t** CLOCK_SetFbeMode (**uint8_t** *frdiv*, **mcg_dmx32_t** *dmx32*, **mcg_drs_t** *drs*, **void(*)**(**void**) *flStableDelay*)

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>flStableDelay</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.
<i>kStatus_Success</i>	Switched to the target mode successfully.

30.7.50 **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.

30.7.51 **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.

30.7.52 **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 *pllcs* 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_Pll-ClkSelExtPll, NULL);`

Function Documentation

30.7.53 `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.

30.7.54 `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(...);  
*
```

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.

30.7.55 `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.

30.7.56 **status_t** CLOCK_BootToFeiMode (mcg_dm32_t *dm32*, 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>dm32</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 dm32 is set to kMCG_Dm32Fine, the slow IRC must not be trimmed to frequency above 32768 Hz.

30.7.57 **status_t** CLOCK_BootToFeeMode (mcg_oscsel_t *oscsel*, uint8_t *frdiv*, mcg_dm32_t *dm32*, 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

Function Documentation

<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.

30.7.58 **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 sytem boot up.

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 _mcg_ircclk_enable_mode .

Return values

<i>kStatus_MCG_Source-Used</i>	Could not change MCGIRCLK setting.
<i>kStatus_Success</i>	Switched to the target mode successfully.

30.7.59 **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 sytem boot up.

Parameters

<i>oscsel</i>	OSC clock select, MCG_C7[OSCSEL].
---------------	-----------------------------------

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.

30.7.60 **status_t** CLOCK_BootToPeeMode (**mcg_oscsel_t** *oscsel*, **mcg_pll_clk_select_t** *pllcs*, **mcg_pll_config_t** *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.

30.7.61 **status_t** CLOCK_SetMcgConfig (**mcg_config_t** *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](#).

Variable Documentation

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.

30.8 Variable Documentation

30.8.1 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(8000000); // 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.

30.8.2 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.

30.9 Multipurpose Clock Generator (MCG)

The MCUXpresso SDK provides a peripheral driver for the module of MCUXpresso SDK devices.

30.9.1 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.

30.9.1.1 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_GetPlloFreq\(\)](#), [CLOCK_GetPll1Freq\(\)](#), and [CLOCK_GetExtPllFreq\(\)](#). These functions get the clock frequency based on the current MCG registers.

30.9.1.2 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 divider. 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_EnablePllo\(\)](#) 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)

30.9.1.3 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.

30.9.1.4 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.

30.9.1.5 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.

30.9.1.6 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.

30.9.2 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.

30.9.2.1 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>flStableDelay=NULL</code>
	FBI -> BLPI	<code>CLOCK_SetLowPowerEnable(true)</code>

30.9.2.2 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>flStableDelay=NULL</code>
	FBI -> BLPI	<code>CLOCK_SetLowPowerEnable(true)</code>

Multipurpose Clock Generator (MCG)

30.9.2.3 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(...) // fl-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 flStableDelay=NULL
	FBI -> BLPI	CLOCK_SetLowPower-Enable(true)

30.9.2.4 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 -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)

Multipurpose Clock Generator (MCG)

	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)

30.9.2.5 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_SetExternalRefClk-Config(...)
	FBI -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	FEE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClk-Config(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with fllStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPower-Enable(true)

30.9.2.6 Switch between BLPI and PEI

Use case	Steps	Functions
BLPI -> PEI	BLPI -> PBI	CLOCK_SetPbiMode(...)
	PBI -> PEI	CLOCK_SetPeiMode(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config(...)
PEI -> BLPI	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config
	PEI -> FBI	CLOCK_InternalModeToFbi-ModeQuick(...)
	FBI -> BLPI	CLOCK_SetLowPower-Enable(true)

30.9.3 Code Configuration Option

30.9.3.1 MCG_USER_CONFIG_FLL_STABLE_DELAY_EN

When switching to use FLL with function [CLOCK_SetFeiMode\(\)](#) 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 = 300000;
    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 31 DMA Manager

31.1 Overview

DMA Manager provides a series of functions to manage the DMAMUX instances and channels.

31.2 Function groups

31.2.1 DMAMGR Initialization and De-initialization

This function group initializes and deinitializes the DMA Manager.

31.2.2 DMAMGR Operation

This function group requests/releases the DMAMUX channel and configures the channel request source.

31.3 Typical use case

31.3.1 DMAMGR static 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 static allocate mechanism */
channel = kDMAMGR_STATIC_ALLOCATE;
DMAMGR_RequestChannel(&dmamanager_handle, kDmaRequestMux0AlwaysOn63, channel, &handle)
    ;
```

31.3.2 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.

31.4 Data Structure Documentation

31.4.1 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](#)

- `uint32_t startChannel`
Channel numbers for the DMA instance which need to be managed by dma manager.
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.

31.4.1.0.0.56 Field Documentation

31.4.1.0.0.56.1 `void* dmamanager_handle_t::dma_base`

31.4.1.0.0.56.2 `uint32_t dmamanager_handle_t::channelNum`

31.4.1.0.0.56.3 `uint32_t dmamanager_handle_t::startChannel`

31.4.1.0.0.56.4 `bool dmamanager_handle_t::s_DMAMGR_Channels[64]`

31.4.1.0.0.56.5 `uint32_t dmamanager_handle_t::DmamuxInstanceStart`

31.4.1.0.0.56.6 `uint32_t dmamanager_handle_t::multiple`

31.5 Macro Definition Documentation

31.5.1 `#define DMAMGR_DYNAMIC_ALLOCATE 0xFFU`

31.6 Enumeration Type Documentation

31.6.1 `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.

31.7 Function Documentation

31.7.1 `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.

31.7.2 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.
--------------------------	--

31.7.3 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 dma_request_source_t
<i>channel</i>	The channel number users want to occupy. If using the dynamic channel allocate mechanism, set the channel equal to DMAMGR_DYNAMIC_ALLOCATE.
<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_NoFreeChannel</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.

31.7.4 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.

31.7.5 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 32

Secure Digital Card/Embedded MultiMedia Card (CARD)

32.1 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

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),
 - `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` {

Overview

```
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

- 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 block-Count)

- 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)
- IO extended write transfer function.*
- 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 read 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)

Data Structure Documentation

- *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.

32.2 Data Structure Documentation

32.2.1 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 united 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*

32.2.2 struct sdio_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- HOST_CONFIG [host](#)
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 [cccrVersion](#)
CCCR version.

Data Structure Documentation

- uint8_t [ioTotalNumber](#)
total number of IO function
- uint32_t [cccrflags](#)
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

32.2.3 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

32.2.4 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.

32.3 Macro Definition Documentation

32.3.1 #define FSL_SDMMC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 2U))
*/*2.1.2*/*

32.4 Enumeration Type Documentation

32.4.1 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.

Enumeration Type Documentation

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 invaild voltage
kStatus_SDMMC_SDIO_SwitchHighSpeedFail switch to high speed fail
kStatus_SDMMC_SDIO_ReadCISFail read CIS fail
kStatus_SDMMC_SDIO_InvalidCard invaild 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

32.4.2 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

32.4.3 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

32.4.4 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 3.1.8v

32.4.5 enum _host_endian_mode

Enumerator

kHOST_EndianModeBig Big endian mode.
kHOST_EndianModeHalfWordBig Half word big endian mode.
kHOST_EndianModeLittle Little endian mode.

32.5 Function Documentation

32.5.1 status_t SD_Init (sd_card_t * *card*)

This function initializes the card on a specific host controller.

Function Documentation

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_SetBus-WidthFailed</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.

32.5.2 void SD_Deinit (sd_card_t * *card*)

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

32.5.3 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.

32.5.4 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.

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.

Function Documentation

<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.

32.5.5 **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_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<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.

32.5.6 **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_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_Success</i>	Operate successfully.

32.5.7 **status_t MMC_Init (mmc_card_t * *card*)**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

Function Documentation

<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_SDMMC_All-SendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_Set-RelativeAddressFailed</i>	Set relative address failed.
<i>kStatus_SDMMC_Send-CsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Select-CardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_Send-ExtendedCsdFailed</i>	Send EXT_CSD failed.
<i>kStatus_SDMMC_SetBus-WidthFailed</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.

32.5.8 void MMC_Deinit (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

32.5.9 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.

32.5.10 **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.
<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.

32.5.11 **status_t MMC_WriteBlocks (mmc_card_t * *card*, const uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)**

Function Documentation

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_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.

32.5.12 **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_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_Success</i>	Operate successfully.

32.5.13 **status_t MMC_SelectPartition (mmc_card_t * *card*, mmc_access_partition_t *partitionNumber*)**

Parameters

<i>card</i>	Card descriptor.
<i>partition-Number</i>	The partition number.

Return values

<i>kStatus_SDMMC_-ConfigureExtendedCsd-Failed</i>	Configure EXT_CSD failed.
<i>kStatus_Success</i>	Operate successfully.

32.5.14 **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

Function Documentation

<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_-ConfigureExtendedCsd-Failed</i>	Configure EXT_CSD failed.
<i>kStatus_SDMMC_-ConfigureBootFailed</i>	Configure boot failed.
<i>kStatus_Success</i>	Operate successfully.

32.5.15 **status_t** SDIO_CardIsActive (**sdio_card_t** * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

32.5.16 **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 numner
<i>register</i>	address
<i>the</i>	data pinter to write
<i>raw</i>	flag, indicate read after write or write only

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

32.5.17 **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
<i>register</i>	address
<i>data</i>	pointer to read

Return values

<i>kStatus_SDMMC_TransferFailed</i>	
<i>kStatus_Success</i>	

32.5.18 **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

Function Documentation

Return values

<i>kStatus_SDMMC_- TransferFailed</i>	
<i>kStatus_SDMMC_SDIO- _InvalidArgument</i>	
<i>kStatus_Success</i>	

32.5.19 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>	

32.5.20 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>	

32.5.21 **status_t** SDIO_SetBlockSize (**sdio_card_t** * *card*, **sdio_func_num_t** *func*, **uint32_t** *blockSize*)

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>	

32.5.22 **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>	

32.5.23 **status_t** SDIO_SetDataBusWidth (**sdio_card_t** * *card*, **sdio_bus_width_t** *busWidth*)

Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>data</i>	bus width

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

32.5.24 **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>	

32.5.25 **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>	
<i>kStatus_SDMMC_- TransferFailed</i>	
<i>kStatus_Success</i>	

32.5.26 status_t SDIO_Init (sdio_card_t * card)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Go- IdleFailed</i>	
<i>kStatus_SDMMC_Hand- ShakeOperation- ConditionFailed</i>	
<i>kStatus_SDMMC_SDIO- _InvalidCard</i>	
<i>kStatus_SDMMC_SDIO- _InvalidVoltage</i>	
<i>kStatus_SDMMC_Send- RelativeAddressFailed</i>	
<i>kStatus_SDMMC_Select- CardFailed</i>	
<i>kStatus_SDMMC_SDIO- _SwitchHighSpeedFail</i>	
<i>kStatus_SDMMC_SDIO- _ReadCISFail</i>	
<i>kStatus_SDMMC_- TransferFailed</i>	
<i>kStatus_Success</i>	

32.5.27 `status_t SDIO_EnableInterrupt (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>	

32.5.28 **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>	

32.5.29 **status_t** SDIO_SelectIO (**sdio_card_t** * *card*, **sdio_func_num_t** *func*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Function Documentation

Return values

<i>kStatus_SDMMC_- TransferFailed</i>	
<i>kStatus_Success</i>	

32.5.30 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>	

32.5.31 void SDIO_Delnit (sdio_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

32.5.32 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
--------------------------	----------------

32.5.33 status_t CardInsertDetect (HOST_TYPE * *hostBase*)

Function Documentation

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

32.5.34 **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

32.5.35 **void** HOST_Deinit (**void** * *host*)

Parameters

<i>host</i>	the pointer to host structure in card structure.
-------------	--

Chapter 33

SPI based Secure Digital Card (SDSPI)

33.1 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...](#)

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.

33.2 Data Structure Documentation

33.2.1 struct sdsapi_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.

33.2.2 struct sdsapi_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.

33.2.3 struct sdsapi_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- [sdsapi_host_t](#) * [host](#)
Host state information.
- uint32_t [relativeAddress](#)
Relative address of the card.
- uint32_t [flags](#)
Flags defined in _sdsapi_card_flag.
- uint8_t [rawCid](#) [16U]
Raw CID content.
- uint8_t [rawCsd](#) [16U]

Enumeration Type Documentation

- *Raw CSD content.*
uint8_t [rawScr](#) [8U]
- *Raw SCR 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.*

33.2.3.0.0.57 Field Documentation

33.2.3.0.0.57.1 uint32_t sdspi_card_t::flags

33.3 Enumeration Type Documentation

33.3.1 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.

33.3.2 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.

33.3.3 enum sdspi_response_type_t

Enumerator

kSDSPI_ResponseTypeR1 Response 1.

kSDSPI_ResponseTypeR1b Response 1 with busy.

kSDSPI_ResponseTypeR2 Response 2.

kSDSPI_ResponseTypeR3 Response 3.

kSDSPI_ResponseTypeR7 Response 7.

33.4 Function Documentation

33.4.1 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_Set-FrequencyFailed</i>	Set frequency failed.
<i>kStatus_SDSPI_GoIdle-Failed</i>	Go idle failed.
<i>kStatus_SDSPI_Send-InterfaceConditionFailed</i>	Send interface condition failed.

Function Documentation

<i>kStatus_SDSPI_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_Timeout</i>	Send command timeout.
<i>kStatus_SDSPI_Not-SupportYet</i>	Not support yet.
<i>kStatus_SDSPI_ReadOcr-Failed</i>	Read OCR failed.
<i>kStatus_SDSPI_SetBlock-SizeFailed</i>	Set block size failed.
<i>kStatus_SDSPI_SendCsd-Failed</i>	Send CSD failed.
<i>kStatus_SDSPI_SendCid-Failed</i>	Send CID failed.
<i>kStatus_Success</i>	Operate successfully.

33.4.2 void SDSPI_Deinit (sdspi_card_t * *card*)

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor
-------------	-----------------

33.4.3 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.

33.4.4 **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.

33.4.5 **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_Write-Protected</i>	Card is write protected.
<i>kStatus_SDSPI_Send-CommandFailed</i>	Send command failed.
<i>kStatus_SDSPI-ResponseError</i>	Response is error.
<i>kStatus_SDSPI_Write-Failed</i>	Write data failed.
<i>kStatus_SDSPI-ExchangeFailed</i>	Exchange data over SPI failed.
<i>kStatus_SDSPI_Wait-ReadyFailed</i>	Wait card to be ready status failed.
<i>kStatus_Success</i>	Operate successfully.

Chapter 34 Debug Console

34.1 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.

34.2 Function groups

34.2.1 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);
```

34.2.2 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 */
```

34.3 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 equivalent 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\n\rTime: %u ticks %2.5f milliseconds\n\rDONE\n\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](#)

34.4 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.

34.4.1 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.

34.4.2 Guide Semihosting for Keil μ Vision

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 */
```

Semihosting

```
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.

34.4.3 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 flages" 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.

34.4.4 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 recoding. 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.

34.4.5 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" : Telet.
 - 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 35

Notification Framework

35.1 Overview

This section describes the programming interface of the Notifier driver.

35.2 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.

35.3 Data Structure Documentation

35.3.1 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

35.3.1.0.0.58 Field Documentation

35.3.1.0.0.58.1 `notifier_user_config_t* notifier_notification_block_t::targetConfig`

35.3.1.0.0.58.2 `notifier_policy_t notifier_notification_block_t::policy`

35.3.1.0.0.58.3 `notifier_notification_type_t notifier_notification_block_t::notifyType`

35.3.2 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.

35.3.2.0.0.59 Field Documentation

35.3.2.0.0.59.1 `notifier_callback_t notifier_callback_config_t::callback`

35.3.2.0.0.59.2 `notifier_callback_type_t notifier_callback_config_t::callbackType`

35.3.2.0.0.59.3 `void* notifier_callback_config_t::callbackData`

35.3.3 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.

35.3.3.0.0.60 Field Documentation

35.3.3.0.0.60.1 `notifier_user_config_t** notifier_handle_t::configsTable`

35.3.3.0.0.60.2 `uint8_t notifier_handle_t::configsNumber`

35.3.3.0.0.60.3 `notifier_callback_config_t* notifier_handle_t::callbacksTable`

35.3.3.0.0.60.4 `uint8_t notifier_handle_t::callbacksNumber`

35.3.3.0.0.60.5 `uint8_t notifier_handle_t::errorCallbackIndex`

35.3.3.0.0.60.6 `uint8_t notifier_handle_t::currentConfigIndex`

35.3.3.0.0.60.7 `notifier_user_function_t notifier_handle_t::userFunction`

35.3.3.0.0.60.8 `void* notifier_handle_t::userData`

35.4 Typedef Documentation

35.4.1 `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.

35.4.2 `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`.

35.4.3 `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`.

35.5 Enumeration Type Documentation

35.5.1 `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.

35.5.2 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.

35.5.3 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.

35.5.4 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.

35.6 Function Documentation

35.6.1 `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.

35.6.2 **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.

35.6.3 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 36 Shell

36.1 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.

36.2 Function groups

36.2.1 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>> ");
```

36.2.2 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.

36.2.3 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.

36.3 Data Structure Documentation

36.3.1 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

- *Total history command in hist buff.*
char [hist_buf](#) [SHELL_HIST_MAX][SHELL_BUFFER_SIZE]
- *History buffer.*
bool [exit](#)
Exit Flag.

36.3.2 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.

36.3.2.0.0.61 Field Documentation

36.3.2.0.0.61.1 const char* shell_command_context_t::pcCommand

For example "help". It must be all lower case.

36.3.2.0.0.61.2 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".

36.3.2.0.0.61.3 const cmd_function_t shell_command_context_t::pFuncCallBack

36.3.2.0.0.61.4 uint8_t shell_command_context_t::cExpectedNumberOfParameters

36.3.3 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.

36.4 Macro Definition Documentation

36.4.1 `#define SHELL_USE_HISTORY (0U)`

36.4.2 `#define SHELL_SEARCH_IN_HIST (1U)`

36.4.3 `#define SHELL_USE_FILE_STREAM (0U)`

36.4.4 `#define SHELL_AUTO_COMPLETE (1U)`

36.4.5 `#define SHELL_BUFFER_SIZE (64U)`

36.4.6 `#define SHELL_MAX_ARGS (8U)`

36.4.7 `#define SHELL_HIST_MAX (3U)`

36.4.8 `#define SHELL_MAX_CMD (20U)`

36.5 Typedef Documentation

36.5.1 `typedef void(* send_data_cb_t)(uint8_t *buf, uint32_t len)`

36.5.2 `typedef void(* recv_data_cb_t)(uint8_t *buf, uint32_t len)`

36.5.3 `typedef int(* printf_data_t)(const char *format,...)`

36.5.4 `typedef int32_t(* cmd_function_t)(p_shell_context_t context, int32_t argc, char **argv)`

36.6 Enumeration Type Documentation

36.6.1 `enum fun_key_status_t`

Enumerator

kSHELL_Normal Normal key.

kSHELL_Special Special key.

kSHELL_Function Function key.

36.7 Function Documentation

36.7.1 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

36.7.2 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

36.7.3 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.

How to Reach Us:**Home Page:**nxp.com**Web Support:**nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

NXP reserves the right to make changes without further notice to any products herein. NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

NXP, the NXP logo, Freescale, the Freescale logo, 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: MCUXSDKKL46APIRM

Rev. 0

Mar 2017

