

Kinetis SDK v.2.1 API Reference Manual

NXP Semiconductors

Document Number: KSDK21KL43ZAPIRM
Rev. 0
Oct 2016



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	27
6.3	Data Structure Documentation	30
6.3.1	struct cmp_config_t	30
6.3.2	struct cmp_filter_config_t	30
6.3.3	struct cmp_dac_config_t	31
6.4	Macro Definition Documentation	31
6.4.1	FSL_CMP_DRIVER_VERSION	31
6.5	Enumeration Type Documentation	31
6.5.1	_cmp_interrupt_enable	31
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	32
6.6.1	CMP_Init	32
6.6.2	CMP_Deinit	33
6.6.3	CMP_Enable	33
6.6.4	CMP_GetDefaultConfig	33
6.6.5	CMP_SetInputChannels	34
6.6.6	CMP_EnableDMA	34
6.6.7	CMP_SetFilterConfig	34

Contents

Section Number	Title	Page Number
6.6.8	CMP_SetDACCConfig	35
6.6.9	CMP_EnableInterrupts	35
6.6.10	CMP_DisableInterrupts	35
6.6.11	CMP_GetStatusFlags	35
6.6.12	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	41
7.5.3	cop_timeout_mode_t	41
7.6	Function Documentation	41
7.6.1	COP_GetDefaultConfig	41
7.6.2	COP_Init	41
7.6.3	COP_Disable	42
7.6.4	COP_Refresh	42

Chapter DAC: Digital-to-Analog Converter Driver

8.1	Overview	43
8.2	Typical use case	43
8.2.1	Working as a basic DAC without the hardware buffer feature	43
8.2.2	Working with the hardware buffer	43
8.3	Data Structure Documentation	46
8.3.1	struct dac_config_t	46
8.3.2	struct dac_buffer_config_t	46
8.4	Macro Definition Documentation	47
8.4.1	FSL_DAC_DRIVER_VERSION	47
8.5	Enumeration Type Documentation	47
8.5.1	_dac_buffer_status_flags	47

Contents

Section Number	Title	Page Number
8.5.2	_dac_buffer_interrupt_enable	47
8.5.3	dac_reference_voltage_source_t	47
8.5.4	dac_buffer_trigger_mode_t	47
8.5.5	dac_buffer_work_mode_t	48
8.6	Function Documentation	48
8.6.1	DAC_Init	48
8.6.2	DAC_Deinit	48
8.6.3	DAC_GetDefaultConfig	48
8.6.4	DAC_Enable	49
8.6.5	DAC_EnableBuffer	49
8.6.6	DAC_SetBufferConfig	49
8.6.7	DAC_GetDefaultBufferConfig	49
8.6.8	DAC_EnableBufferDMA	50
8.6.9	DAC_SetBufferValue	50
8.6.10	DAC_DoSoftwareTriggerBuffer	50
8.6.11	DAC_GetBufferReadPointer	50
8.6.12	DAC_SetBufferReadPointer	51
8.6.13	DAC_EnableBufferInterrupts	51
8.6.14	DAC_DisableBufferInterrupts	51
8.6.15	DAC_GetBufferStatusFlags	51
8.6.16	DAC_ClearBufferStatusFlags	52

Chapter DMA: Direct Memory Access Controller Driver

9.1	Overview	53
9.2	Typical use case	53
9.2.1	DMA Operation	53
9.3	Data Structure Documentation	56
9.3.1	struct dma_transfer_config_t	56
9.3.2	struct dma_channel_link_config_t	57
9.3.3	struct dma_handle_t	57
9.4	Macro Definition Documentation	58
9.4.1	FSL_DMA_DRIVER_VERSION	58
9.5	Typedef Documentation	58
9.5.1	dma_callback	58
9.6	Enumeration Type Documentation	58
9.6.1	_dma_channel_status_flags	58
9.6.2	dma_transfer_size_t	58
9.6.3	dma_modulo_t	58
9.6.4	dma_channel_link_type_t	59

Contents

Section Number	Title	Page Number
9.6.5	dma_transfer_type_t	59
9.6.6	dma_transfer_options_t	59
9.7	Function Documentation	60
9.7.1	DMA_Init	60
9.7.2	DMA_Deinit	61
9.7.3	DMA_ResetChannel	61
9.7.4	DMA_SetTransferConfig	61
9.7.5	DMA_SetChannelLinkConfig	62
9.7.6	DMA_SetSourceAddress	62
9.7.7	DMA_SetDestinationAddress	62
9.7.8	DMA_SetTransferSize	63
9.7.9	DMA_SetModulo	63
9.7.10	DMA_EnableCycleSteal	63
9.7.11	DMA_EnableAutoAlign	64
9.7.12	DMA_EnableAsyncRequest	64
9.7.13	DMA_EnableInterrupts	64
9.7.14	DMA_DisableInterrupts	65
9.7.15	DMA_EnableChannelRequest	65
9.7.16	DMA_DisableChannelRequest	65
9.7.17	DMA_TriggerChannelStart	65
9.7.18	DMA_GetRemainingBytes	66
9.7.19	DMA_GetChannelStatusFlags	66
9.7.20	DMA_ClearChannelStatusFlags	66
9.7.21	DMA_CreateHandle	67
9.7.22	DMA_SetCallback	67
9.7.23	DMA_PreparesTransfer	67
9.7.24	DMA_SubmitTransfer	68
9.7.25	DMA_StartTransfer	68
9.7.26	DMA_StopTransfer	69
9.7.27	DMA_AbortTransfer	69
9.7.28	DMA_HandleIRQ	69

Chapter **DMAMUX: Direct Memory Access Multiplexer Driver**

10.1	Overview	71
10.2	Typical use case	71
10.2.1	DMAMUX Operation	71
10.3	Macro Definition Documentation	71
10.3.1	FSL_DMAMUX_DRIVER_VERSION	71
10.4	Function Documentation	72
10.4.1	DMAMUX_Init	72

Contents

Section Number	Title	Page Number
10.4.2	DMAMUX_Deinit	73
10.4.3	DMAMUX_EnableChannel	73
10.4.4	DMAMUX_DisableChannel	73
10.4.5	DMAMUX_SetSource	74
10.4.6	DMAMUX_EnablePeriodTrigger	74
10.4.7	DMAMUX_DisablePeriodTrigger	74

Chapter C90TFS Flash Driver

11.1	Overview	75
11.2	Data Structure Documentation	84
11.2.1	struct flash_execute_in_ram_function_config_t	84
11.2.2	struct flash_swap_state_config_t	84
11.2.3	struct flash_swap_ifr_field_config_t	84
11.2.4	union flash_swap_ifr_field_data_t	85
11.2.5	union pflash_protection_status_low_t	85
11.2.6	struct pflash_protection_status_t	86
11.2.7	struct flash_prefetch_speculation_status_t	86
11.2.8	struct flash_protection_config_t	86
11.2.9	struct flash_access_config_t	87
11.2.10	struct flash_operation_config_t	87
11.2.11	struct flash_config_t	88
11.3	Macro Definition Documentation	89
11.3.1	MAKE_VERSION	89
11.3.2	FSL_FLASH_DRIVER_VERSION	89
11.3.3	FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT	89
11.3.4	FLASH_DRIVER_IS_FLASH_RESIDENT	89
11.3.5	FLASH_DRIVER_IS_EXPORTED	90
11.3.6	kStatusGroupGeneric	90
11.3.7	MAKE_STATUS	90
11.3.8	FOUR_CHAR_CODE	90
11.4	Enumeration Type Documentation	90
11.4.1	_flash_driver_version_constants	90
11.4.2	_flash_status	90
11.4.3	_flash_driver_api_keys	91
11.4.4	flash_margin_value_t	91
11.4.5	flash_security_state_t	91
11.4.6	flash_protection_state_t	92
11.4.7	flash_execute_only_access_state_t	92
11.4.8	flash_property_tag_t	92
11.4.9	_flash_execute_in_ram_function_constants	93
11.4.10	flash_read_resource_option_t	93

Contents

Section Number	Title	Page Number
11.4.11	_flash_read_resource_range	93
11.4.12	flash_flexram_function_option_t	93
11.4.13	flash_swap_function_option_t	94
11.4.14	flash_swap_control_option_t	94
11.4.15	flash_swap_state_t	94
11.4.16	flash_swap_block_status_t	94
11.4.17	flash_partition_flexram_load_option_t	95
11.4.18	flash_memory_index_t	95
11.5	Function Documentation	95
11.5.1	FLASH_Init	95
11.5.2	FLASH_SetCallback	95
11.5.3	FLASH_PrepareExecuteInRamFunctions	96
11.5.4	FLASH_EraseAll	96
11.5.5	FLASH_Erase	97
11.5.6	FLASH_EraseAllUnsecure	98
11.5.7	FLASH_EraseAllExecuteOnlySegments	99
11.5.8	FLASH_Program	100
11.5.9	FLASH_ProgramOnce	101
11.5.10	FLASH_ReadResource	101
11.5.11	FLASH_ReadOnce	102
11.5.12	FLASH_GetSecurityState	103
11.5.13	FLASH_SecurityBypass	103
11.5.14	FLASH_VerifyEraseAll	104
11.5.15	FLASH_VerifyErase	105
11.5.16	FLASH_VerifyProgram	106
11.5.17	FLASH_VerifyEraseAllExecuteOnlySegments	107
11.5.18	FLASH_IsProtected	108
11.5.19	FLASH_IsExecuteOnly	109
11.5.20	FLASHGetProperty	109
11.5.21	FLASH_PflashSetProtection	110
11.5.22	FLASH_PflashGetProtection	110
Chapter FlexIO: FlexIO Driver		
12.1	Overview	113
12.2	FlexIO Driver	114
12.2.1	Overview	114
12.2.2	Data Structure Documentation	118
12.2.3	Macro Definition Documentation	121
12.2.4	Typedef Documentation	121
12.2.5	Enumeration Type Documentation	121
12.2.6	Function Documentation	125

Contents

Section Number	Title	Page Number
12.3	FlexIO Camera Driver	136
12.3.1	Overview	136
12.3.2	Typical use case	136
12.3.3	Data Structure Documentation	139
12.3.4	Macro Definition Documentation	140
12.3.5	Enumeration Type Documentation	140
12.3.6	Function Documentation	141
12.3.7	FlexIO eDMA Camera Driver	145
12.4	FlexIO I2C Master Driver	149
12.4.1	Overview	149
12.4.2	Typical use case	149
12.4.3	Data Structure Documentation	153
12.4.4	Macro Definition Documentation	156
12.4.5	Typedef Documentation	156
12.4.6	Enumeration Type Documentation	156
12.4.7	Function Documentation	157
12.5	FlexIO I2S Driver	166
12.5.1	Overview	166
12.5.2	Typical use case	166
12.5.3	Data Structure Documentation	171
12.5.4	Macro Definition Documentation	173
12.5.5	Enumeration Type Documentation	173
12.5.6	Function Documentation	175
12.5.7	FlexIO eDMA I2S Driver	186
12.5.8	FlexIO DMA I2S Driver	192
12.6	FlexIO SPI Driver	198
12.6.1	Overview	198
12.6.2	Typical use case	198
12.6.3	Data Structure Documentation	204
12.6.4	Macro Definition Documentation	209
12.6.5	Typedef Documentation	209
12.6.6	Enumeration Type Documentation	209
12.6.7	Function Documentation	211
12.6.8	FlexIO eDMA SPI Driver	225
12.6.9	FlexIO DMA SPI Driver	231
12.7	FlexIO UART Driver	236
12.7.1	Overview	236
12.7.2	Typical use case	236
12.7.3	Data Structure Documentation	244
12.7.4	Macro Definition Documentation	247
12.7.5	Typedef Documentation	247

Contents

Section Number	Title	Page Number
12.7.6	Enumeration Type Documentation	247
12.7.7	Function Documentation	248
12.7.8	FlexIO eDMA UART Driver	261
12.7.9	FlexIO DMA UART Driver	267

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

13.1	Overview	273
13.2	Data Structure Documentation	273
13.2.1	struct gpio_pin_config_t	273
13.3	Macro Definition Documentation	274
13.3.1	FSL_GPIO_DRIVER_VERSION	274
13.4	Enumeration Type Documentation	274
13.4.1	gpio_pin_direction_t	274
13.5	GPIO Driver	275
13.5.1	Overview	275
13.5.2	Typical use case	275
13.5.3	Function Documentation	276
13.6	FGPIO Driver	279
13.6.1	Overview	279
13.6.2	Typical use case	279
13.6.3	Function Documentation	280

Chapter **I2C: Inter-Integrated Circuit Driver**

14.1	Overview	285
14.2	I2C Driver	286
14.2.1	Overview	286
14.2.2	Typical use case	286
14.2.3	Data Structure Documentation	293
14.2.4	Macro Definition Documentation	298
14.2.5	Typedef Documentation	298
14.2.6	Enumeration Type Documentation	298
14.2.7	Function Documentation	300
14.3	I2C eDMA Driver	315
14.3.1	Overview	315
14.3.2	Data Structure Documentation	315
14.3.3	Typedef Documentation	316
14.3.4	Function Documentation	316

Contents

Section Number	Title	Page Number
14.4	I2C DMA Driver	319
14.4.1	Overview	319
14.4.2	Data Structure Documentation	319
14.4.3	Typedef Documentation	320
14.4.4	Function Documentation	320
14.5	I2C FreeRTOS Driver	322
14.5.1	Overview	322
14.5.2	Function Documentation	322
14.6	I2C µCOS/II Driver	324
14.6.1	Overview	324
14.6.2	Function Documentation	324
14.7	I2C µCOS/III Driver	326
14.7.1	Overview	326
14.7.2	Function Documentation	326
 Chapter LLWU: Low-Leakage Wakeup Unit Driver		
15.1	Overview	329
15.2	External wakeup pins configurations	329
15.3	Internal wakeup modules configurations	329
15.4	Digital pin filter for external wakeup pin configurations	329
15.5	Data Structure Documentation	330
15.5.1	struct llwu_external_pin_filter_mode_t	330
15.6	Macro Definition Documentation	330
15.6.1	FSL_LLWU_DRIVER_VERSION	330
15.7	Enumeration Type Documentation	330
15.7.1	llwu_external_pin_mode_t	330
15.7.2	llwu_pin_filter_mode_t	331
15.8	Function Documentation	331
15.8.1	LLWU_SetExternalWakePinMode	331
15.8.2	LLWU_GetExternalWakePinFlag	331
15.8.3	LLWU_ClearExternalWakePinFlag	332
15.8.4	LLWU_EnableInternalModuleInterruptWakup	333
15.8.5	LLWU_GetInternalWakeModuleFlag	333
15.8.6	LLWU_SetPinFilterMode	333
15.8.7	LLWU_GetPinFilterFlag	334

Contents

Section Number	Title	Page Number
15.8.8	LLWU_ClearPinFilterFlag	334

Chapter LPTMR: Low-Power Timer

16.1	Overview	335
16.2	Function groups	335
16.2.1	Initialization and deinitialization	335
16.2.2	Timer period Operations	335
16.2.3	Start and Stop timer operations	335
16.2.4	Status	336
16.2.5	Interrupt	336
16.3	Typical use case	336
16.3.1	LPTMR tick example	336
16.4	Data Structure Documentation	338
16.4.1	struct lptmr_config_t	338
16.5	Enumeration Type Documentation	339
16.5.1	lptmr_pin_select_t	339
16.5.2	lptmr_pin_polarity_t	339
16.5.3	lptmr_timer_mode_t	339
16.5.4	lptmr_prescaler_glitch_value_t	340
16.5.5	lptmr_prescaler_clock_select_t	340
16.5.6	lptmr_interrupt_enable_t	340
16.5.7	lptmr_status_flags_t	341
16.6	Function Documentation	341
16.6.1	LPTMR_Init	341
16.6.2	LPTMR_Deinit	341
16.6.3	LPTMR_GetDefaultConfig	341
16.6.4	LPTMR_EnableInterrupts	342
16.6.5	LPTMR_DisableInterrupts	342
16.6.6	LPTMR_GetEnabledInterrupts	342
16.6.7	LPTMR_GetStatusFlags	342
16.6.8	LPTMR_ClearStatusFlags	343
16.6.9	LPTMR_SetTimerPeriod	343
16.6.10	LPTMR_GetCurrentTimerCount	343
16.6.11	LPTMR_StartTimer	344
16.6.12	LPTMR_StopTimer	344

Chapter LPUART: Low Power UART Driver

17.1	Overview	345
-------------	---------------------------	------------

Contents

Section Number	Title	Page Number
17.2	LPUART Driver	346
17.2.1	Overview	346
17.2.2	Typical use case	346
17.2.3	Data Structure Documentation	350
17.2.4	Macro Definition Documentation	352
17.2.5	Typedef Documentation	352
17.2.6	Enumeration Type Documentation	352
17.2.7	Function Documentation	355
17.3	LPUART DMA Driver	370
17.3.1	Overview	370
17.3.2	Data Structure Documentation	370
17.3.3	Typedef Documentation	371
17.3.4	Function Documentation	371
17.4	LPUART eDMA Driver	375
17.4.1	Overview	375
17.4.2	Data Structure Documentation	376
17.4.3	Typedef Documentation	377
17.4.4	Function Documentation	377
17.5	LPUART µCOS/II Driver	381
17.5.1	Overview	381
17.5.2	Data Structure Documentation	381
17.5.3	Function Documentation	382
17.6	LPUART µCOS/III Driver	385
17.6.1	Overview	385
17.6.2	Data Structure Documentation	385
17.6.3	Function Documentation	386
17.7	LPUART FreeRTOS Driver	389
17.7.1	Overview	389
17.7.2	Data Structure Documentation	389
17.7.3	Function Documentation	390
Chapter 18: PIT: Periodic Interrupt Timer		
18.1	Overview	393
18.2	Function groups	393
18.2.1	Initialization and deinitialization	393
18.2.2	Timer period Operations	393
18.2.3	Start and Stop timer operations	393
18.2.4	Status	394
18.2.5	Interrupt	394

Contents

Section Number	Title	Page Number
18.3	Typical use case	394
18.3.1	PIT tick example	394
18.4	Data Structure Documentation	396
18.4.1	struct pit_config_t	396
18.5	Enumeration Type Documentation	396
18.5.1	pit_chnl_t	396
18.5.2	pit_interrupt_enable_t	397
18.5.3	pit_status_flags_t	397
18.6	Function Documentation	397
18.6.1	PIT_Init	397
18.6.2	PIT_Deinit	397
18.6.3	PIT_GetDefaultConfig	397
18.6.4	PIT_SetTimerChainMode	398
18.6.5	PIT_EnableInterrupts	398
18.6.6	PIT_DisableInterrupts	398
18.6.7	PIT_GetEnabledInterrupts	399
18.6.8	PIT_GetStatusFlags	399
18.6.9	PIT_ClearStatusFlags	399
18.6.10	PIT_SetTimerPeriod	400
18.6.11	PIT_GetCurrentTimerCount	400
18.6.12	PIT_StartTimer	401
18.6.13	PIT_StopTimer	401
18.6.14	PIT_GetLifetimeTimerCount	401

Chapter **PMC: Power Management Controller**

19.1	Overview	403
19.2	Data Structure Documentation	404
19.2.1	struct pmc_low_volt_detect_config_t	404
19.2.2	struct pmc_low_volt_warning_config_t	404
19.2.3	struct pmc_bandgap_buffer_config_t	404
19.3	Macro Definition Documentation	405
19.3.1	FSL_PMC_DRIVER_VERSION	405
19.4	Enumeration Type Documentation	405
19.4.1	pmc_low_volt_detect_volt_select_t	405
19.4.2	pmc_low_volt_warning_volt_select_t	405
19.5	Function Documentation	405
19.5.1	PMC_ConfigureLowVoltDetect	405
19.5.2	PMC_GetLowVoltDetectFlag	406

Contents

Section Number	Title	Page Number
19.5.3	PMC_ClearLowVoltDetectFlag	406
19.5.4	PMC_ConfigureLowVoltWarning	406
19.5.5	PMC_GetLowVoltWarningFlag	407
19.5.6	PMC_ClearLowVoltWarningFlag	407
19.5.7	PMC_ConfigureBandgapBuffer	407
19.5.8	PMC_GetPeriphIOIsolationFlag	408
19.5.9	PMC_ClearPeriphIOIsolationFlag	408
19.5.10	PMC_IsRegulatorInRunRegulation	408

Chapter PORT: Port Control and Interrupts

20.1	Overview	411
20.2	Typical configuration use case	411
20.2.1	Input PORT configuration	411
20.2.2	I2C PORT Configuration	411
20.3	Data Structure Documentation	413
20.3.1	struct port_pin_config_t	413
20.4	Macro Definition Documentation	413
20.4.1	FSL_PORT_DRIVER_VERSION	413
20.5	Enumeration Type Documentation	413
20.5.1	_port_pull	413
20.5.2	_port_slew_rate	414
20.5.3	_port_passive_filter_enable	414
20.5.4	_port_drive_strength	414
20.5.5	port_mux_t	414
20.5.6	port_interrupt_t	415
20.6	Function Documentation	415
20.6.1	PORT_SetPinConfig	415
20.6.2	PORT_SetMultiplePinsConfig	415
20.6.3	PORT_SetPinMux	416
20.6.4	PORT_SetPinInterruptConfig	416
20.6.5	PORT_GetPinsInterruptFlags	417
20.6.6	PORT_ClearPinsInterruptFlags	418

Chapter RCM: Reset Control Module Driver

21.1	Overview	421
21.2	Data Structure Documentation	422
21.2.1	struct rcm_reset_pin_filter_config_t	422

Contents

Section Number	Title	Page Number
21.3	Macro Definition Documentation	422
21.3.1	FSL_RCM_DRIVER_VERSION	422
21.4	Enumeration Type Documentation	422
21.4.1	rcm_reset_source_t	422
21.4.2	rcm_run_wait_filter_mode_t	423
21.4.3	rcm_boot_rom_config_t	423
21.5	Function Documentation	423
21.5.1	RCM_GetPreviousResetSources	423
21.5.2	RCM_GetStickyResetSources	424
21.5.3	RCM_ClearStickyResetSources	424
21.5.4	RCM_ConfigureResetPinFilter	425
21.5.5	RCM_GetBootRomSource	425
21.5.6	RCM_ClearBootRomSource	425
21.5.7	RCM_SetForceBootRomSource	426
 Chapter RTC: Real Time Clock		
22.1	Overview	427
22.2	Function groups	427
22.2.1	Initialization and deinitialization	427
22.2.2	Set & Get Datetime	427
22.2.3	Set & Get Alarm	427
22.2.4	Start & Stop timer	428
22.2.5	Status	428
22.2.6	Interrupt	428
22.2.7	RTC Oscillator	428
22.2.8	Monotonic Counter	428
22.3	Typical use case	428
22.3.1	RTC tick example	428
22.4	Data Structure Documentation	431
22.4.1	struct rtc_datetime_t	431
22.4.2	struct rtc_config_t	432
22.5	Enumeration Type Documentation	433
22.5.1	rtc_interrupt_enable_t	433
22.5.2	rtc_status_flags_t	433
22.5.3	rtc_osc_cap_load_t	433
22.6	Function Documentation	433
22.6.1	RTC_Init	433
22.6.2	RTC_Deinit	434

Contents

Section Number	Title	Page Number
22.6.3	RTC_GetDefaultConfig	434
22.6.4	RTC_SetDatetime	434
22.6.5	RTC_GetDatetime	435
22.6.6	RTC_SetAlarm	435
22.6.7	RTC_GetAlarm	435
22.6.8	RTC_EnableInterrupts	436
22.6.9	RTC_DisableInterrupts	436
22.6.10	RTC_GetEnabledInterrupts	436
22.6.11	RTC_GetStatusFlags	436
22.6.12	RTC_ClearStatusFlags	437
22.6.13	RTC_StartTimer	437
22.6.14	RTC_StopTimer	437
22.6.15	RTC_SetOscCapLoad	437
22.6.16	RTC_Reset	438

Chapter SAI: Serial Audio Interface

23.1	Overview	439
23.2	Typical use case	439
23.2.1	SAI Send/receive using an interrupt method	439
23.2.2	SAI Send/receive using a DMA method	440
23.3	Data Structure Documentation	445
23.3.1	struct sai_config_t	445
23.3.2	struct sai_transfer_format_t	446
23.3.3	struct sai_transfer_t	446
23.3.4	struct _sai_handle	447
23.4	Macro Definition Documentation	447
23.4.1	SAI_XFER_QUEUE_SIZE	447
23.5	Enumeration Type Documentation	447
23.5.1	_sai_status_t	447
23.5.2	sai_protocol_t	448
23.5.3	sai_master_slave_t	448
23.5.4	sai_mono_stereo_t	448
23.5.5	sai_sync_mode_t	448
23.5.6	sai_mclk_source_t	448
23.5.7	sai_bclk_source_t	449
23.5.8	_sai_interrupt_enable_t	449
23.5.9	_sai_dma_enable_t	449
23.5.10	_sai_flags	449
23.5.11	sai_reset_type_t	449
23.5.12	sai_fifo_packing_t	450

Contents

Section Number	Title	Page Number
23.5.13	sai_sample_rate_t	450
23.5.14	sai_word_width_t	450
23.6	Function Documentation	450
23.6.1	SAI_TxInit	450
23.6.2	SAI_RxInit	451
23.6.3	SAI_TxGetDefaultConfig	451
23.6.4	SAI_RxGetDefaultConfig	451
23.6.5	SAI_Deinit	452
23.6.6	SAI_TxReset	452
23.6.7	SAI_RxReset	452
23.6.8	SAI_TxEnable	452
23.6.9	SAI_RxEnable	453
23.6.10	SAI_TxGetStatusFlag	453
23.6.11	SAI_TxClearStatusFlags	453
23.6.12	SAI_RxGetStatusFlag	453
23.6.13	SAI_RxClearStatusFlags	454
23.6.14	SAI_TxEnableInterrupts	454
23.6.15	SAI_RxEnableInterrupts	454
23.6.16	SAI_TxDisableInterrupts	455
23.6.17	SAI_RxDisableInterrupts	456
23.6.18	SAI_TxEnableDMA	456
23.6.19	SAI_RxEnableDMA	456
23.6.20	SAI_TxGetDataRegisterAddress	457
23.6.21	SAI_RxGetDataRegisterAddress	458
23.6.22	SAI_TxSetFormat	458
23.6.23	SAI_RxSetFormat	459
23.6.24	SAI_WriteBlocking	459
23.6.25	SAI_WriteData	459
23.6.26	SAI_ReadBlocking	460
23.6.27	SAI_ReadData	460
23.6.28	SAI_TransferTxCreateHandle	460
23.6.29	SAI_TransferRxCreateHandle	461
23.6.30	SAI_TransferTxSetFormat	461
23.6.31	SAI_TransferRxSetFormat	462
23.6.32	SAI_TransferSendNonBlocking	462
23.6.33	SAI_TransferReceiveNonBlocking	463
23.6.34	SAI_TransferGetSendCount	463
23.6.35	SAI_TransferGetReceiveCount	464
23.6.36	SAI_TransferAbortSend	464
23.6.37	SAI_TransferAbortReceive	465
23.6.38	SAI_TransferTxHandleIRQ	465
23.6.39	SAI_TransferRxHandleIRQ	465
23.7	SAI DMA Driver	466

Contents

Section Number	Title	Page Number
23.7.1	Overview	466
23.7.2	Data Structure Documentation	467
23.7.3	Function Documentation	467
23.8	SAI eDMA Driver	473
23.8.1	Overview	473
23.8.2	Data Structure Documentation	474
23.8.3	Function Documentation	475
Chapter	SIM: System Integration Module Driver	
24.1	Overview	481
24.2	Data Structure Documentation	481
24.2.1	struct sim_uid_t	481
24.3	Enumeration Type Documentation	482
24.3.1	_sim_usb_volt_reg_enable_mode	482
24.3.2	_sim_flash_mode	482
24.4	Function Documentation	482
24.4.1	SIM_SetUsbVoltRegulatorEnableMode	482
24.4.2	SIM_GetUniqueId	483
24.4.3	SIM_SetFlashMode	483
Chapter	SLCD: Segment LCD Driver	
25.1	Overview	485
25.2	Typical use case	485
25.2.1	SLCD Initialization operation	485
25.3	Data Structure Documentation	490
25.3.1	struct slcd_fault_detect_config_t	490
25.3.2	struct slcd_clock_config_t	490
25.3.3	struct slcd_config_t	491
25.4	Macro Definition Documentation	492
25.4.1	FSL_SLCD_DRIVER_VERSION	492
25.5	Enumeration Type Documentation	492
25.5.1	slcd_power_supply_option_t	492
25.5.2	slcd_regulated_voltage_trim_t	493
25.5.3	slcd_load_adjust_t	493
25.5.4	slcd_clock_src_t	494
25.5.5	slcd_alt_clock_div_t	494

Contents

Section Number	Title	Page Number
25.5.6	slcd_clock_prescaler_t	494
25.5.7	slcd_duty_cycle_t	495
25.5.8	slcd_phase_type_t	495
25.5.9	slcd_phase_index_t	495
25.5.10	slcd_display_mode_t	496
25.5.11	slcd_blink_mode_t	496
25.5.12	slcd_blink_rate_t	496
25.5.13	slcd_fault_detect_clock_prescaler_t	496
25.5.14	slcd_fault_detect_sample_window_width_t	497
25.5.15	slcd_interrupt_enable_t	497
25.5.16	slcd_lowpower_behavior	497
25.6	Function Documentation	497
25.6.1	SLCD_Init	497
25.6.2	SLCD_Deinit	498
25.6.3	SLCD_GetDefaultConfig	498
25.6.4	SLCD_StartDisplay	498
25.6.5	SLCD_StopDisplay	499
25.6.6	SLCD_StartBlinkMode	499
25.6.7	SLCD_StopBlinkMode	499
25.6.8	SLCD_SetBackPlanePhase	499
25.6.9	SLCD_SetFrontPlaneSegments	500
25.6.10	SLCD_SetFrontPlaneOnePhase	500
25.6.11	SLCD_EnablePadSafeState	501
25.6.12	SLCD_GetFaultDetectCounter	501
25.6.13	SLCD_EnableInterrupts	501
25.6.14	SLCD_DisableInterrupts	502
25.6.15	SLCD_GetInterruptStatus	502
25.6.16	SLCD_ClearInterruptStatus	502
Chapter	SMC: System Mode Controller Driver	
26.1	Overview	505
26.2	Typical use case	505
26.2.1	Enter wait or stop modes	505
26.3	Data Structure Documentation	507
26.3.1	struct smc_power_mode_vlls_config_t	507
26.4	Macro Definition Documentation	507
26.4.1	FSL_SMC_DRIVER_VERSION	507
26.5	Enumeration Type Documentation	508
26.5.1	smc_power_mode_protection_t	508

Contents

Section Number	Title	Page Number
26.5.2	<code>smc_power_state_t</code>	508
26.5.3	<code>smc_run_mode_t</code>	508
26.5.4	<code>smc_stop_mode_t</code>	508
26.5.5	<code>smc_stop_submode_t</code>	509
26.5.6	<code>smc_partial_stop_option_t</code>	509
26.5.7	<code>_smc_status</code>	509
26.6	Function Documentation	509
26.6.1	<code>SMC_SetPowerModeProtection</code>	509
26.6.2	<code>SMC_GetPowerModeState</code>	510
26.6.3	<code>SMC_PreEnterStopModes</code>	510
26.6.4	<code>SMC_PostExitStopModes</code>	510
26.6.5	<code>SMC_PreEnterWaitModes</code>	510
26.6.6	<code>SMC_PostExitWaitModes</code>	510
26.6.7	<code>SMC_SetPowerModeRun</code>	511
26.6.8	<code>SMC_SetPowerModeWait</code>	512
26.6.9	<code>SMC_SetPowerModeStop</code>	512
26.6.10	<code>SMC_SetPowerModeVlpr</code>	512
26.6.11	<code>SMC_SetPowerModeVlpw</code>	513
26.6.12	<code>SMC_SetPowerModeVlps</code>	513
26.6.13	<code>SMC_SetPowerModeLls</code>	513
26.6.14	<code>SMC_SetPowerModeVlls</code>	513
Chapter	SPI: Serial Peripheral Interface Driver	
27.1	Overview	515
27.2	SPI Driver	516
27.2.1	<code>Overview</code>	516
27.2.2	<code>Typical use case</code>	516
27.2.3	<code>Data Structure Documentation</code>	522
27.2.4	<code>Macro Definition Documentation</code>	524
27.2.5	<code>Enumeration Type Documentation</code>	524
27.2.6	<code>Function Documentation</code>	527
27.3	SPI DMA Driver	537
27.3.1	<code>Overview</code>	537
27.3.2	<code>Data Structure Documentation</code>	538
27.3.3	<code>Typedef Documentation</code>	538
27.3.4	<code>Function Documentation</code>	538
27.4	SPI FreeRTOS driver	543
27.4.1	<code>Overview</code>	543
27.4.2	<code>Function Documentation</code>	543

Contents

Section Number	Title	Page Number
27.5	SPI µCOS/II driver	545
27.5.1	Overview	545
27.5.2	Function Documentation	545
27.6	SPI µCOS/III driver	547
27.6.1	Overview	547
27.6.2	Function Documentation	547
 Chapter TPM: Timer PWM Module		
28.1	Overview	549
28.2	Typical use case	550
28.2.1	PWM output	550
28.3	Data Structure Documentation	554
28.3.1	struct tpm_chnl_pwm_signal_param_t	554
28.3.2	struct tpm_config_t	554
28.4	Enumeration Type Documentation	555
28.4.1	tpm_chnl_t	555
28.4.2	tpm_pwm_mode_t	555
28.4.3	tpm_pwm_level_select_t	556
28.4.4	tpm_trigger_select_t	556
28.4.5	tpm_trigger_source_t	556
28.4.6	tpm_output_compare_mode_t	556
28.4.7	tpm_input_capture_edge_t	557
28.4.8	tpm_clock_source_t	557
28.4.9	tpm_clock_prescale_t	557
28.4.10	tpm_interrupt_enable_t	557
28.4.11	tpm_status_flags_t	558
28.5	Function Documentation	558
28.5.1	TPM_Init	558
28.5.2	TPM_Deinit	558
28.5.3	TPM_GetDefaultConfig	558
28.5.4	TPM_SetupPwm	559
28.5.5	TPM_UpdatePwmDutyCycle	559
28.5.6	TPM_UpdateChnlEdgeLevelSelect	560
28.5.7	TPM_SetupInputCapture	560
28.5.8	TPM_SetupOutputCompare	560
28.5.9	TPM_EnableInterrupts	561
28.5.10	TPM_DisableInterrupts	561
28.5.11	TPM_GetEnabledInterrupts	561
28.5.12	TPM_GetStatusFlags	561

Contents

Section Number	Title	Page Number
28.5.13	TPM_ClearStatusFlags	562
28.5.14	TPM_StartTimer	562
28.5.15	TPM_StopTimer	562
Chapter UART: Universal Asynchronous Receiver/Transmitter Driver		
29.1	Overview	563
29.2	UART Driver	564
29.2.1	Overview	564
29.2.2	Typical use case	564
29.2.3	Data Structure Documentation	572
29.2.4	Macro Definition Documentation	574
29.2.5	Typedef Documentation	574
29.2.6	Enumeration Type Documentation	574
29.2.7	Function Documentation	576
29.3	UART DMA Driver	589
29.3.1	Overview	589
29.3.2	Data Structure Documentation	589
29.3.3	Typedef Documentation	590
29.3.4	Function Documentation	590
29.4	UART eDMA Driver	594
29.4.1	Overview	594
29.4.2	Data Structure Documentation	594
29.4.3	Typedef Documentation	595
29.4.4	Function Documentation	595
29.5	UART FreeRTOS Driver	599
29.5.1	Overview	599
29.5.2	Data Structure Documentation	599
29.5.3	Function Documentation	600
29.6	UART μCOS/II Driver	602
29.6.1	Overview	602
29.6.2	Data Structure Documentation	602
29.6.3	Function Documentation	603
29.7	UART μCOS/III Driver	605
29.7.1	Overview	605
29.7.2	Data Structure Documentation	605
29.7.3	Function Documentation	606

Contents

Section Number	Title	Page Number
Chapter	VREF: Voltage Reference Driver	
30.1	Overview	609
30.2	Typical use case and example	609
30.3	Data Structure Documentation	610
30.3.1	struct vref_config_t	610
30.4	Macro Definition Documentation	610
30.4.1	FSL_VREF_DRIVER_VERSION	610
30.5	Enumeration Type Documentation	610
30.5.1	vref_buffer_mode_t	610
30.6	Function Documentation	610
30.6.1	VREF_Init	610
30.6.2	VREF_Deinit	611
30.6.3	VREF_GetDefaultConfig	611
30.6.4	VREF_SetTrimVal	611
30.6.5	VREF_GetTrimVal	612
Chapter	Clock Driver	
31.1	Overview	613
31.2	Get frequency	613
31.3	External clock frequency	613
31.4	Data Structure Documentation	618
31.4.1	struct sim_clock_config_t	618
31.4.2	struct oscer_config_t	618
31.4.3	struct osc_config_t	619
31.4.4	struct mcglite_config_t	619
31.5	Macro Definition Documentation	620
31.5.1	FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL	620
31.5.2	FSL_CLOCK_DRIVER_VERSION	620
31.5.3	DMAMUX_CLOCKS	620
31.5.4	RTC_CLOCKS	620
31.5.5	SAI_CLOCKS	621
31.5.6	SPI_CLOCKS	621
31.5.7	SLCD_CLOCKS	621
31.5.8	PIT_CLOCKS	621
31.5.9	PORT_CLOCKS	621

Contents

Section Number	Title	Page Number
31.5.10	LPUART_CLOCKS	622
31.5.11	DAC_CLOCKS	622
31.5.12	LPTMR_CLOCKS	622
31.5.13	ADC16_CLOCKS	622
31.5.14	FLEXIO_CLOCKS	622
31.5.15	VREF_CLOCKS	623
31.5.16	DMA_CLOCKS	623
31.5.17	UART_CLOCKS	623
31.5.18	TPM_CLOCKS	623
31.5.19	I2C_CLOCKS	623
31.5.20	FTF_CLOCKS	624
31.5.21	CMP_CLOCKS	624
31.5.22	SYS_CLK	624
31.6	Enumeration Type Documentation	624
31.6.1	clock_name_t	624
31.6.2	clock_usb_src_t	625
31.6.3	clock_ip_name_t	625
31.6.4	_osc_cap_load	625
31.6.5	_oscer_enable_mode	625
31.6.6	osc_mode_t	625
31.6.7	mcglite_clkout_src_t	625
31.6.8	mcglite_lirc_mode_t	626
31.6.9	mcglite_lirc_div_t	626
31.6.10	mcglite_mode_t	626
31.6.11	_mcglite_irclk_enable_mode	626
31.7	Function Documentation	626
31.7.1	CLOCK_EnableClock	626
31.7.2	CLOCK_DisableClock	627
31.7.3	CLOCK_SetEr32kClock	627
31.7.4	CLOCK_SetLpuart0Clock	627
31.7.5	CLOCK_SetLpuart1Clock	627
31.7.6	CLOCK_SetTpmClock	627
31.7.7	CLOCK_SetFlexio0Clock	628
31.7.8	CLOCK_EnableUsbfs0Clock	628
31.7.9	CLOCK_DisableUsbfs0Clock	628
31.7.10	CLOCK_SetClkOutClock	628
31.7.11	CLOCK_SetRtcClkOutClock	628
31.7.12	CLOCK_SetOutDiv	629
31.7.13	CLOCK_GetFreq	629
31.7.14	CLOCK_GetCoreSysClkFreq	629
31.7.15	CLOCK_GetPlatClkFreq	629
31.7.16	CLOCK_GetBusClkFreq	630
31.7.17	CLOCK_GetFlashClkFreq	630

Contents

Section Number	Title	Page Number
31.7.18	CLOCK_GetEr32kClkFreq	630
31.7.19	CLOCK_GetOsc0ErClkFreq	630
31.7.20	CLOCK_SetSimConfig	630
31.7.21	CLOCK_SetSimSafeDivs	630
31.7.22	CLOCK_GetOutClkFreq	631
31.7.23	CLOCK_GetInternalRefClkFreq	631
31.7.24	CLOCK_GetPeriphClkFreq	631
31.7.25	CLOCK_GetMode	631
31.7.26	CLOCK_SetMcgliteConfig	632
31.7.27	OSC_SetExtRefClkConfig	633
31.7.28	OSC_SetCapLoad	633
31.7.29	CLOCK_InitOsc0	634
31.7.30	CLOCK_DeinitOsc0	635
31.7.31	CLOCK_SetXtal0Freq	635
31.7.32	CLOCK_SetXtal32Freq	635
31.8	Variable Documentation	635
31.8.1	g_xtal0Freq	635
31.8.2	g_xtal32Freq	635
31.9	Multipurpose Clock Generator Lite (MCGLITE)	637
31.9.1	Function description	637
Chapter	Debug Console	
32.1	Overview	639
32.2	Function groups	639
32.2.1	Initialization	639
32.2.2	Advanced Feature	640
32.3	Typical use case	643
32.4	Semihosting	645
32.4.1	Guide Semihosting for IAR	645
32.4.2	Guide Semihosting for Keil µVision	645
32.4.3	Guide Semihosting for KDS	647
32.4.4	Guide Semihosting for ATL	647
32.4.5	Guide Semihosting for ARMGCC	648
Chapter	Notification Framework	
33.1	Overview	651
33.2	Notifier Overview	651

Contents

Section Number	Title	Page Number
33.3	Data Structure Documentation	653
33.3.1	struct notifier_notification_block_t	653
33.3.2	struct notifier_callback_config_t	654
33.3.3	struct notifier_handle_t	654
33.4	Typedef Documentation	655
33.4.1	notifier_user_config_t	655
33.4.2	notifier_user_function_t	655
33.4.3	notifier_callback_t	656
33.5	Enumeration Type Documentation	656
33.5.1	_notifier_status	656
33.5.2	notifier_policy_t	657
33.5.3	notifier_notification_type_t	657
33.5.4	notifier_callback_type_t	657
33.6	Function Documentation	658
33.6.1	NOTIFIER_CreateHandle	658
33.6.2	NOTIFIER_SwitchConfig	659
33.6.3	NOTIFIER_GetErrorCallbackIndex	660

Chapter **Shell**

34.1	Overview	661
34.2	Function groups	661
34.2.1	Initialization	661
34.2.2	Advanced Feature	661
34.2.3	Shell Operation	662
34.3	Data Structure Documentation	663
34.3.1	struct shell_context_struct	663
34.3.2	struct shell_command_context_t	664
34.3.3	struct shell_command_context_list_t	664
34.4	Macro Definition Documentation	665
34.4.1	SHELL_USE_HISTORY	665
34.4.2	SHELL_SEARCH_IN_HIST	665
34.4.3	SHELL_USE_FILE_STREAM	665
34.4.4	SHELL_AUTO_COMPLETE	665
34.4.5	SHELL_BUFFER_SIZE	665
34.4.6	SHELL_MAX_ARGS	665
34.4.7	SHELL_HIST_MAX	665
34.4.8	SHELL_MAX_CMD	665
34.5	Typedef Documentation	665

Contents

Section Number	Title	Page Number
34.5.1	send_data_cb_t	665
34.5.2	recv_data_cb_t	665
34.5.3	printf_data_t	665
34.5.4	cmd_function_t	665
34.6	Enumeration Type Documentation	665
34.6.1	fun_key_status_t	665
34.7	Function Documentation	666
34.7.1	SHELL_Init	666
34.7.2	SHELL_RegisterCommand	666
34.7.3	SHELL_Main	666

Chapter 1

Introduction

The Software Development Kit (KSDK) v2.1 is a collection of software enablement, for NXP Kinetis Microcontrollers, that includes peripheral drivers, multicore support, USB stack, and integrated RTOS support for FreeRTOS, μC/OS-II, and μC/OS-III. In addition to the base enablement, the KSDK is augmented with demo applications, driver example projects, and API documentation to help users quickly leverage the support provided by KSDK v2.1. The KEx Web UI is available to provide access to all SDK v2.1 packages. See the *SDK v.2.1.0 Release Notes* (document KSDK210RN) and the supported Devices section at www.nxp.com/k sdk for details.

The SDK v2.1 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 KSDK peripheral drivers and leverage native RTOS services to better comply to the RTOS cases.
- Real time operation systems (RTOS) including FreeRTOS OS, μC/OS-II, and μC/OS-III.
- 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 SDK v2.1 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
- LPCXpresso IDE

The peripheral drivers and RTOS driver wrappers can be used across multiple devices within the Kinetis product family without modification. The configuration items for each driver are encapsulated into C language data structures. Kinetis device-specific configuration information is provided as part of the KSDK 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 Kinetis SDK 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 Kinetis SDK documents, see the kex.nxp.com/apidoc.

Deliverable	Location
Examples	<install_dir>/examples/
Demo Applications	<install_dir>/examples/<board_name>/demo_apps/
Driver Examples	<install_dir>/examples/<board_name>/driver_examples/
Documentation	<install_dir>/doc/
USB Documentation	<install_dir>/doc/usb/
Middleware	<install_dir>/middleware/
USB Stack	<install_dir>/middleware/usb_<version>
Drivers	<install_dir>/<device_name>/drivers/
CMSIS Standard ARM Cortex-M Headers, math and DSP Libraries	<install_dir>/<device_name>/CMSIS/
Device Startup and Linker	<install_dir>/<device_name>/<toolchain>/
SDK Utilities	<install_dir>/<device_name>/utilities/
RTOS Kernels	<install_dir>/rtos/

Table 2: KSDK Folder Structure

Chapter 2

Driver errors status

- #kStatus_DMA_Busy = 5000
- kStatus_FLEXIO_I2S_Idle = 2300
- kStatus_FLEXIO_I2S_TxBusy = 2301
- kStatus_FLEXIO_I2S_RxBusy = 2302
- kStatus_FLEXIO_I2S_Error = 2303
- kStatus_FLEXIO_I2S_QueueFull = 2304
- 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_NOTIFIER_ErrorNotificationBefore = 9800
- kStatus_NOTIFIER_ErrorNotificationAfter = 9801

Chapter 3

Architectural Overview

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

Overview

The Kinetis 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 Kinetis SDK
5. Demo Applications based on the Kinetis SDK

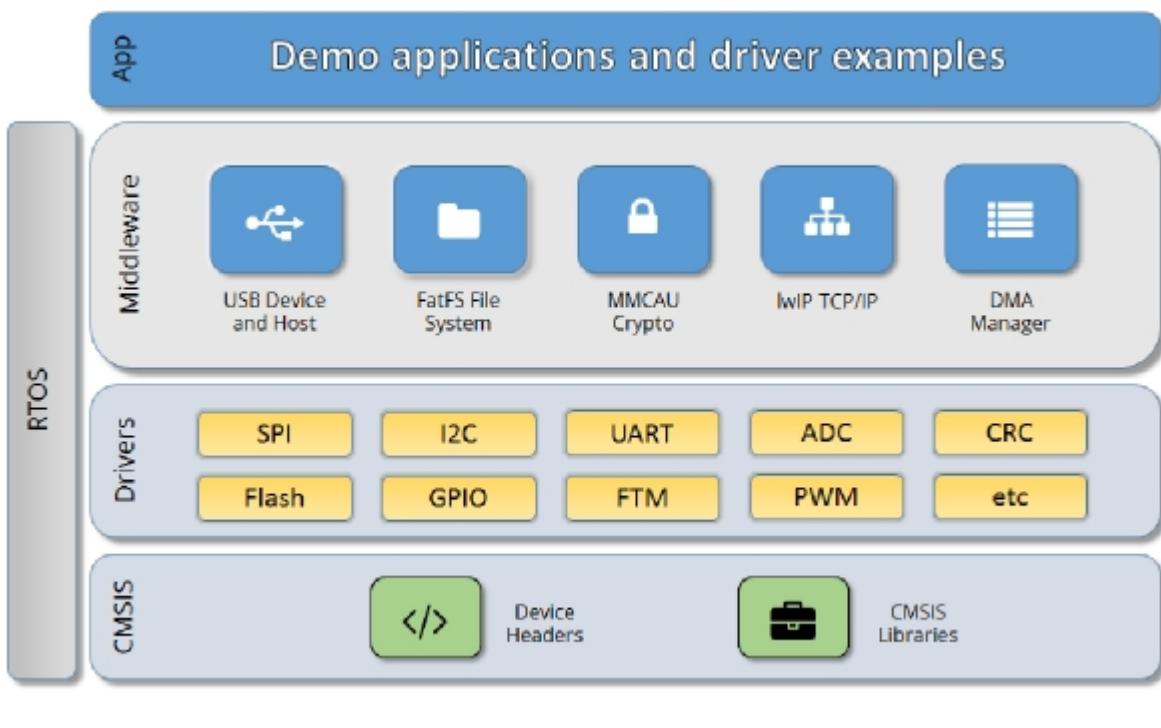


Figure 1: KSDK Block Diagram

Kinetis MCU header files

Each supported Kinetis MCU device in the KSDK has an overall System-on Chip (SoC) memory-mapped

header file. This header file contains the memory map and register base address for each peripheral and the IRQ vector table with associated vector numbers. The overall SoC header file provides access to the peripheral registers through pointers and predefined bit masks. In addition to the overall SoC memory-mapped header file, the KSDK 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 KSDK 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.

KSDK Peripheral Drivers

The KSDK peripheral drivers mainly consist of low-level functional APIs for the Kinetis MCU product family on-chip peripherals and also of high-level transactional APIs for some bus drivers/DMA driver/e-DMA driver to quickly enable the peripherals and perform transfers.

All KSDK 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 KSDK 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 Kinetis devices. It's up to the user to ensure that NVIC interrupts are properly disabled after usage is complete.

Interrupt handling for transactional APIs

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

```
PUBWEAK SPI0_IRQHandler  
PUBWEAK SPI0_DriverIRQHandler  
SPI0_IRQHandler
```

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

The first level of the weak implementation are the functions defined in the vector table. In the devices/<DEVICE_NAME>/<TOOLCHAIN>/startup_<DEVICE_NAME>.S file, the implementation of the first layer weak function calls the second layer of weak function. The implementation of the second layer weak function (ex. SPI0_DriverIRQHandler) jumps to itself (BX). The KSDK drivers with transactional APIs provide the reimplementation of the second layer function inside of the peripheral driver. If the KSDK drivers with transactional APIs are linked into the image, the SPI0_DriverIRQHandler is replaced with the function implemented in the KSDK 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 KSDK 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 Kinetis MCU device to another. An overall Peripheral Feature Header File is provided for the KSDK-supported MCU device to define the features or configuration differences for each Kinetis sub-family device.

Application

See the *Getting Started with Kinetis SDK (KSDK) v2.1* document (KSDK21GSUG).



Chapter 4

Trademarks

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions

Freescale, the Freescale logo, Kinetis, and Processor Expert are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Tower is a trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM, ARM powered logo, Keil, and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2016 Freescale Semiconductors, Inc.

Chapter 5

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

5.1 Overview

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

5.2 Typical use case

5.2.1 Polling Configuration

```
adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

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

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

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

5.2.2 Interrupt Configuration

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

Typical use case

```
// ...

adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

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

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

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

// ...

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

Data Structures

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

Enumerations

- enum `_adc16_channel_status_flags` { `kADC16_ChannelConversionDoneFlag` = ADC_SC1_COC_O_MASK }

- *Channel status flags.*
- enum `_adc16_status_flags` {

 `kADC16_ActiveFlag` = ADC_SC2_ADACT_MASK,

 `kADC16_CalibrationFailedFlag` = ADC_SC3_CALF_MASK }
- Converter status flags.*
- enum `adc16_channel_mux_mode_t` {

 `kADC16_ChannelMuxA` = 0U,

 `kADC16_ChannelMuxB` = 1U }
- Channel multiplexer mode for each channel.*
- enum `adc16_clock_divider_t` {

 `kADC16_ClockDivider1` = 0U,

 `kADC16_ClockDivider2` = 1U,

 `kADC16_ClockDivider4` = 2U,

 `kADC16_ClockDivider8` = 3U }
- Clock divider for the converter.*
- enum `adc16_resolution_t` {

 `kADC16_Resolution8or9Bit` = 0U,

 `kADC16_Resolution12or13Bit` = 1U,

 `kADC16_Resolution10or11Bit` = 2U,

 `kADC16_ResolutionSE8Bit` = `kADC16_Resolution8or9Bit`,

 `kADC16_ResolutionSE12Bit` = `kADC16_Resolution12or13Bit`,

 `kADC16_ResolutionSE10Bit` = `kADC16_Resolution10or11Bit`,

 `kADC16_ResolutionDF9Bit` = `kADC16_Resolution8or9Bit`,

 `kADC16_ResolutionDF13Bit` = `kADC16_Resolution12or13Bit`,

 `kADC16_ResolutionDF11Bit` = `kADC16_Resolution10or11Bit`,

 `kADC16_Resolution16Bit` = 3U,

 `kADC16_ResolutionSE16Bit` = `kADC16_Resolution16Bit`,

 `kADC16_ResolutionDF16Bit` = `kADC16_Resolution16Bit` }
- Converter's resolution.*
- enum `adc16_clock_source_t` {

 `kADC16_ClockSourceAlt0` = 0U,

 `kADC16_ClockSourceAlt1` = 1U,

 `kADC16_ClockSourceAlt2` = 2U,

 `kADC16_ClockSourceAlt3` = 3U,

 `kADC16_ClockSourceAsynchronousClock` = `kADC16_ClockSourceAlt3` }
- Clock source.*
- enum `adc16_long_sample_mode_t` {

 `kADC16_LongSampleCycle24` = 0U,

 `kADC16_LongSampleCycle16` = 1U,

 `kADC16_LongSampleCycle10` = 2U,

 `kADC16_LongSampleCycle6` = 3U,

 `kADC16_LongSampleDisabled` = 4U }
- Long sample mode.*
- enum `adc16_reference_voltage_source_t` {

 `kADC16_ReferenceVoltageSourceVref` = 0U,

 `kADC16_ReferenceVoltageSourceValt` = 1U }

Typical use case

- Reference voltage source.
 - enum `adc16_hardware_average_mode_t` {
 kADC16_HardwareAverageCount4 = 0U,
 kADC16_HardwareAverageCount8 = 1U,
 kADC16_HardwareAverageCount16 = 2U,
 kADC16_HardwareAverageCount32 = 3U,
 kADC16_HardwareAverageDisabled = 4U }
- Hardware average mode.
 - enum `adc16_hardware_compare_mode_t` {
 kADC16_HardwareCompareMode0 = 0U,
 kADC16_HardwareCompareMode1 = 1U,
 kADC16_HardwareCompareMode2 = 2U,
 kADC16_HardwareCompareMode3 = 3U }
- Hardware compare mode.

Driver version

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

Initialization

- void `ADC16_Init` (ADC_Type *base, const `adc16_config_t` *config)
Initializes the ADC16 module.
- void `ADC16_Deinit` (ADC_Type *base)
De-initializes the ADC16 module.
- void `ADC16_GetDefaultConfig` (`adc16_config_t` *config)
Gets an available pre-defined settings for the converter's configuration.
- status_t `ADC16_DoAutoCalibration` (ADC_Type *base)
Automates the hardware calibration.
- static void `ADC16_SetOffsetValue` (ADC_Type *base, int16_t value)
Sets the offset value for the conversion result.

Advanced Features

- static void `ADC16_EnableDMA` (ADC_Type *base, bool enable)
Enables generating the DMA trigger when the conversion is complete.
- static void `ADC16_EnableHardwareTrigger` (ADC_Type *base, bool enable)
Enables the hardware trigger mode.
- void `ADC16_SetChannelMuxMode` (ADC_Type *base, `adc16_channel_mux_mode_t` mode)
Sets the channel mux mode.
- void `ADC16_SetHardwareCompareConfig` (ADC_Type *base, const `adc16_hardware_compare_config_t` *config)
Configures the hardware compare mode.
- void `ADC16_SetHardwareAverage` (ADC_Type *base, `adc16_hardware_average_mode_t` mode)
Sets the hardware average mode.
- uint32_t `ADC16_GetStatusFlags` (ADC_Type *base)
Gets the status flags of the converter.
- void `ADC16_ClearStatusFlags` (ADC_Type *base, uint32_t mask)
Clears the status flags of the converter.

Conversion Channel

- void [ADC16_SetChannelConfig](#) (ADC_Type *base, uint32_t channelGroup, const [adc16_channel_config_t](#) *config)
Configures the conversion channel.
- static uint32_t [ADC16_GetChannelConversionValue](#) (ADC_Type *base, uint32_t channelGroup)
Gets the conversion value.
- uint32_t [ADC16_GetChannelStatusFlags](#) (ADC_Type *base, uint32_t channelGroup)
Gets the status flags of channel.

5.3 Data Structure Documentation

5.3.1 struct adc16_config_t

Data Fields

- [adc16_reference_voltage_source_t](#) referenceVoltageSource
Select the reference voltage source.
- [adc16_clock_source_t](#) clockSource
Select the input clock source to converter.
- bool enableAsynchronousClock
Enable the asynchronous clock output.
- [adc16_clock_divider_t](#) clockDivider
Select the divider of input clock source.
- [adc16_resolution_t](#) resolution
Select the sample resolution mode.
- [adc16_long_sample_mode_t](#) longSampleMode
Select the long sample mode.
- bool enableHighSpeed
Enable the high-speed mode.
- bool enableLowPower
Enable low power.
- bool enableContinuousConversion
Enable continuous conversion mode.

Data Structure Documentation

5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 `adc16_reference_voltage_source_t adc16_config_t::referenceVoltageSource`

5.3.1.0.0.1.2 `adc16_clock_source_t adc16_config_t::clockSource`

5.3.1.0.0.1.3 `bool adc16_config_t::enableAsynchronousClock`

5.3.1.0.0.1.4 `adc16_clock_divider_t adc16_config_t::clockDivider`

5.3.1.0.0.1.5 `adc16_resolution_t adc16_config_t::resolution`

5.3.1.0.0.1.6 `adc16_long_sample_mode_t adc16_config_t::longSampleMode`

5.3.1.0.0.1.7 `bool adc16_config_t::enableHighSpeed`

5.3.1.0.0.1.8 `bool adc16_config_t::enableLowPower`

5.3.1.0.0.1.9 `bool adc16_config_t::enableContinuousConversion`

5.3.2 struct `adc16_hardware_compare_config_t`

Data Fields

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

5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 `adc16_hardware_compare_mode_t adc16_hardware_compare_config_t::hardwareCompareMode`

See "adc16_hardware_compare_mode_t".

5.3.2.0.0.2.2 `int16_t adc16_hardware_compare_config_t::value1`

5.3.2.0.0.2.3 `int16_t adc16_hardware_compare_config_t::value2`

5.3.3 struct `adc16_channel_config_t`

Data Fields

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

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

5.3.3.0.0.3 Field Documentation

5.3.3.0.0.3.1 `uint32_t adc16_channel_config_t::channelNumber`

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

5.3.3.0.0.3.2 `bool adc16_channel_config_t::enableInterruptOnConversionCompleted`

5.3.3.0.0.3.3 `bool adc16_channel_config_t::enableDifferentialConversion`

5.4 Macro Definition Documentation

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

5.5 Enumeration Type Documentation

5.5.1 `enum _adc16_channel_status_flags`

Enumerator

kADC16_ChannelConversionDoneFlag Conversion done.

5.5.2 `enum _adc16_status_flags`

Enumerator

kADC16_ActiveFlag Converter is active.

kADC16_CalibrationFailedFlag Calibration is failed.

5.5.3 `enum adc16_channel_mux_mode_t`

For some ADC16 channels, there are two pin selections in channel multiplexer. For example, ADC0_SE4a and ADC0_SE4b are the different channels that share the same channel number.

Enumerator

kADC16_ChannelMuxA For channel with channel mux a.

kADC16_ChannelMuxB For channel with channel mux b.

Enumeration Type Documentation

5.5.4 enum adc16_clock_divider_t

Enumerator

- kADC16_ClockDivider1* For divider 1 from the input clock to the module.
- kADC16_ClockDivider2* For divider 2 from the input clock to the module.
- kADC16_ClockDivider4* For divider 4 from the input clock to the module.
- kADC16_ClockDivider8* For divider 8 from the input clock to the module.

5.5.5 enum adc16_resolution_t

Enumerator

- kADC16_Resolution8or9Bit* Single End 8-bit or Differential Sample 9-bit.
- kADC16_Resolution12or13Bit* Single End 12-bit or Differential Sample 13-bit.
- kADC16_Resolution10or11Bit* Single End 10-bit or Differential Sample 11-bit.
- kADC16_ResolutionSE8Bit* Single End 8-bit.
- kADC16_ResolutionSE12Bit* Single End 12-bit.
- kADC16_ResolutionSE10Bit* Single End 10-bit.
- kADC16_ResolutionDF9Bit* Differential Sample 9-bit.
- kADC16_ResolutionDF13Bit* Differential Sample 13-bit.
- kADC16_ResolutionDF11Bit* Differential Sample 11-bit.
- kADC16_Resolution16Bit* Single End 16-bit or Differential Sample 16-bit.
- kADC16_ResolutionSE16Bit* Single End 16-bit.
- kADC16_ResolutionDF16Bit* Differential Sample 16-bit.

5.5.6 enum adc16_clock_source_t

Enumerator

- kADC16_ClockSourceAlt0* Selection 0 of the clock source.
- kADC16_ClockSourceAlt1* Selection 1 of the clock source.
- kADC16_ClockSourceAlt2* Selection 2 of the clock source.
- kADC16_ClockSourceAlt3* Selection 3 of the clock source.
- kADC16_ClockSourceAsynchronousClock* Using internal asynchronous clock.

5.5.7 enum adc16_long_sample_mode_t

Enumerator

- kADC16_LongSampleCycle24* 20 extra ADCK cycles, 24 ADCK cycles total.
- kADC16_LongSampleCycle16* 12 extra ADCK cycles, 16 ADCK cycles total.

kADC16_LongSampleCycle10 6 extra ADCK cycles, 10 ADCK cycles total.

kADC16_LongSampleCycle6 2 extra ADCK cycles, 6 ADCK cycles total.

kADC16_LongSampleDisabled Disable the long sample feature.

5.5.8 enum adc16_reference_voltage_source_t

Enumerator

kADC16_ReferenceVoltageSourceVref For external pins pair of VrefH and VrefL.

kADC16_ReferenceVoltageSourceValt For alternate reference pair of ValtH and ValtL.

5.5.9 enum adc16_hardware_average_mode_t

Enumerator

kADC16_HardwareAverageCount4 For hardware average with 4 samples.

kADC16_HardwareAverageCount8 For hardware average with 8 samples.

kADC16_HardwareAverageCount16 For hardware average with 16 samples.

kADC16_HardwareAverageCount32 For hardware average with 32 samples.

kADC16_HardwareAverageDisabled Disable the hardware average feature.

5.5.10 enum adc16_hardware_compare_mode_t

Enumerator

kADC16_HardwareCompareMode0 $x < \text{value1}$.

kADC16_HardwareCompareMode1 $x > \text{value1}$.

kADC16_HardwareCompareMode2 if $\text{value1} \leq \text{value2}$, then $x < \text{value1} \parallel x > \text{value2}$; else,
 $\text{value1} > x > \text{value2}$.

kADC16_HardwareCompareMode3 if $\text{value1} \leq \text{value2}$, then $\text{value1} \leq x \leq \text{value2}$; else $x \geq \text{value1} \parallel x \leq \text{value2}$.

5.6 Function Documentation

5.6.1 void ADC16_Init(ADC_Type * base, const adc16_config_t * config)

Function Documentation

Parameters

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

5.6.2 void ADC16_Deinit (ADC_Type * *base*)

Parameters

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

5.6.3 void ADC16_GetDefaultConfig (adc16_config_t * *config*)

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

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

Parameters

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

5.6.4 status_t ADC16_DoAutoCalibration (ADC_Type * *base*)

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

Parameters

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

Returns

Execution status.

Return values

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

5.6.5 static void ADC16_SetOffsetValue (ADC_Type * *base*, int16_t *value*) [inline], [static]

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

Parameters

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

5.6.6 static void ADC16_EnableDMA (ADC_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

5.6.7 static void ADC16_EnableHardwareTrigger (ADC_Type * *base*, bool *enable*) [inline], [static]

Function Documentation

Parameters

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

5.6.8 void ADC16_SetChannelMuxMode (ADC_Type * *base*, adc16_channel_mux_mode_t *mode*)

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

Parameters

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

5.6.9 void ADC16_SetHardwareCompareConfig (ADC_Type * *base*, const adc16_hardware_compare_config_t * *config*)

The hardware compare mode provides a way to process the conversion result automatically by using hardware. Only the result in the compare range is available. To compare the range, see "adc16_hardware_compare_mode_t" or the appropriate reference manual for more information.

Parameters

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

5.6.10 void ADC16_SetHardwareAverage (ADC_Type * *base*, adc16_hardware_average_mode_t *mode*)

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

Parameters

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

5.6.11 **uint32_t ADC16_GetStatusFlags (ADC_Type * *base*)**

Parameters

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

Returns

Flags' mask if indicated flags are asserted. See "_adc16_status_flags".

5.6.12 **void ADC16_ClearStatusFlags (ADC_Type * *base*, uint32_t *mask*)**

Parameters

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

5.6.13 **void ADC16_SetChannelConfig (ADC_Type * *base*, uint32_t *channelGroup*, const adc16_channel_config_t * *config*)**

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

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

Function Documentation

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

Parameters

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

5.6.14 static uint32_t ADC16_GetChannelConversionValue (ADC_Type * *base*, uint32_t *channelGroup*) [inline], [static]

Parameters

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

Returns

Conversion value.

5.6.15 uint32_t ADC16_GetChannelStatusFlags (ADC_Type * *base*, uint32_t *channelGroup*)

Parameters

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

Returns

Flags' mask if indicated flags are asserted. See "_adc16_channel_status_flags".

Function Documentation

Chapter 6

CMP: Analog Comparator Driver

6.1 Overview

The KSDK provides a peripheral driver for the Analog Comparator (CMP) module of Kinetis 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.
        }
    }
}
```

6.2.2 Interrupt Configuration

```
volatile uint32_t g_CmpFlags = 0U;
```

Typical use case

```
// ...

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.
- void `CMP_SetFilterConfig` (`CMP_Type` *base, const `cmp_filter_config_t` *config)

 Configures the filter.
- 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.

Data Structure Documentation

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

- `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 `uint8_t cmp_filter_config_t::filterCount`

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

6.3.2.0.0.5.2 `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.

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.

Function Documentation

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.
<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;
*
```

Function Documentation

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 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.

6.6.8 void CMP_SetDACConfig (**CMP_Type** * *base*, **const cmp_dac_config_t** * *config*)

Parameters

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

6.6.9 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".

6.6.10 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".

6.6.11 **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".

Function Documentation

6.6.12 void CMP_ClearStatusFlags (*CMP_Type* * *base*, *uint32_t* *mask*)

Parameters

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

Function Documentation

Chapter 7

COP: Watchdog Driver

7.1 Overview

The KSDK provides a peripheral driver for the Computer Operating Properly module (COP) of Kinetis 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_McgIrClock = 1U,
 kCOP_OscErClock = 2U,
 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.
- enum `cop_timeout_mode_t` {
 kCOP_ShortTimeoutMode = 0U,
 kCOP_LongTimeoutMode = 1U }
Define the COP timeout mode.

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)

Enumeration Type Documentation

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

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_timeout_mode_t timeoutMode`
COP timeout mode: long timeout or short timeout.
- `bool enableStop`
Enable or disable COP in STOP mode.
- `bool enableDebug`
Enable or disable COP in DEBUG 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_McgIrClock` COP clock sourced from MCGIRCLK.

`kCOP_OscErClock` COP clock sourced from OSCERCLK.

`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

7.5.3 enum cop_timeout_mode_t

Enumerator

kCOP_ShortTimeoutMode COP selects long timeout.
kCOP_LongTimeoutMode COP selects short timeout.

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:

Function Documentation

```
* 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.

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 KSDK provides a peripheral driver for the Digital-to-Analog Converter (DAC) module of Kinetis 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_SetDefaultBufferConfig(&dacBufferConfigStruct);
DAC_SetBufferConfig(DEMO_DAC_INSTANCE, &dacBufferConfigStruct);
DAC_SetBufferReadPointer(DEMO_DAC_INSTANCE, 0U); // Make sure the read pointer
      to the start.
for (index = 0U, dacValue = 0; index < DEMO_DAC_USED_BUFFER_SIZE; index++, dacValue += (0xFFFFU /
    DEMO_DAC_USED_BUFFER_SIZE))
{
    DAC_SetBufferValue(DEMO_DAC_INSTANCE, index, dacValue);
}
// Clears flags.
#if defined(FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION) && FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferWatermarkInterruptFlag = false;
#endif // FSL_FEATURE_DAC_HAS_WATERMARK_DETECTION
    g_DacBufferReadPointerTopPositionInterruptFlag = false;
    g_DacBufferReadPointerBottomPositionInterruptFlag = false;

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

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

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

Data Structures

- struct `dac_config_t`
DAC module configuration. [More...](#)
- struct `dac_buffer_config_t`
DAC buffer configuration. [More...](#)

Enumerations

- enum `_dac_buffer_status_flags` {
 kDAC_BufferReadPointerTopPositionFlag = DAC_SR_DACBFRPTF_MASK,
 kDAC_BufferReadPointerBottomPositionFlag = DAC_SR_DACBFRPBF_MASK }

- 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,

 kDAC_BufferWorkAsFIFOMode }
- DAC buffer work mode.*

Driver version

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

Initialization

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

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.

Data Structure Documentation

- static uint8_t **DAC_GetBufferReadPointer** (DAC_Type *base)
Gets the current read pointer of the DAC buffer.
- void **DAC_SetBufferReadPointer** (DAC_Type *base, uint8_t index)
Sets the current read pointer of the DAC buffer.
- void **DAC_EnableBufferInterrupts** (DAC_Type *base, uint32_t mask)
Enables interrupts for the DAC buffer.
- void **DAC_DisableBufferInterrupts** (DAC_Type *base, uint32_t mask)
Disables interrupts for the DAC buffer.
- uint32_t **DAC_GetBufferStatusFlags** (DAC_Type *base)
Gets the flags of events for the DAC buffer.
- void **DAC_ClearBufferStatusFlags** (DAC_Type *base, uint32_t mask)
Clears the flags of events for the DAC buffer.

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.

Function Documentation

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.

kDAC_BufferWorkAsFIFOMode FIFO mode.

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;
*
```

Function Documentation

Parameters

<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 KSDK provides a peripheral driver for the Direct Memory Access (DMA) of Kinetis 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_PrepTransfer(&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,
 kDMA_ConfigurationErrorFlag = DMA_DSR_BCR_CE_MASK }

Typical use case

- *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_PrepTransfer](#) (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.

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

9.6.4 enum dma_channel_link_type_t

Enumerator

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

9.6.5 enum dma_transfer_type_t

Enumerator

<i>kDMA_MemoryToMemory</i>	Memory to Memory transfer.
<i>kDMA_PeripheralToMemory</i>	Peripheral to Memory transfer.
<i>kDMA_MemoryToPeripheral</i>	Memory to Peripheral transfer.

9.6.6 enum dma_transfer_options_t

Enumerator

<i>kDMA_NoOptions</i>	Transfer without options.
<i>kDMA_EnableInterrupt</i>	Enable interrupt while transfer complete.

Function Documentation

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 + DSIZE) 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 _dma_channel_status-_flags 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_PreparesTransfer (*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.
---------------	---------------------

Function Documentation

Chapter 10

DMAMUX: Direct Memory Access Multiplexer Driver

10.1 Overview

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

10.2 Typical use case

10.2.1 DMAMUX Operation

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

Driver version

- #define **FSL_DMAMUX_DRIVER_VERSION** (MAKE_VERSION(2, 0, 2))
DMAMUX driver version 2.0.2.

DMAMUX Initialization and de-initialization

- void **DMAMUX_Init** (DMAMUX_Type *base)
Initializes the DMAMUX peripheral.
- void **DMAMUX_Deinit** (DMAMUX_Type *base)
Deinitializes the DMAMUX peripheral.

DMAMUX Channel Operation

- static void **DMAMUX_EnableChannel** (DMAMUX_Type *base, uint32_t channel)
Enables the DMAMUX channel.
- static void **DMAMUX_DisableChannel** (DMAMUX_Type *base, uint32_t channel)
Disables the DMAMUX channel.
- static void **DMAMUX_SetSource** (DMAMUX_Type *base, uint32_t channel, uint32_t source)
Configures the DMAMUX channel source.
- static void **DMAMUX_EnablePeriodTrigger** (DMAMUX_Type *base, uint32_t channel)
Enables the DMAMUX period trigger.
- static void **DMAMUX_DisablePeriodTrigger** (DMAMUX_Type *base, uint32_t channel)
Disables the DMAMUX period trigger.

10.3 Macro Definition Documentation

10.3.1 #define **FSL_DMAMUX_DRIVER_VERSION** (MAKE_VERSION(2, 0, 2))

Function Documentation

10.4 Function Documentation

10.4.1 void DMAMUX_Init (**DMAMUX_Type** * *base*)

This function ungates the DMAMUX clock.

Parameters

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

10.4.2 void DMAMUX_Deinit (DMAMUX_Type * *base*)

This function gates the DMAMUX clock.

Parameters

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

10.4.3 static void DMAMUX_EnableChannel (DMAMUX_Type * *base*, uint32_t *channel*) [inline], [static]

This function enables the DMAMUX channel.

Parameters

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

10.4.4 static void DMAMUX_DisableChannel (DMAMUX_Type * *base*, uint32_t *channel*) [inline], [static]

This function disables the DMAMUX channel.

Note

The user must disable the DMAMUX channel before configuring it.

Parameters

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

Function Documentation

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

10.4.5 static void DMAMUX_SetSource (DMAMUX_Type * *base*, uint32_t *channel*, uint32_t *source*) [inline], [static]

Parameters

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

10.4.6 static void DMAMUX_EnablePeriodTrigger (DMAMUX_Type * *base*, uint32_t *channel*) [inline], [static]

This function enables the DMAMUX period trigger feature.

Parameters

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

10.4.7 static void DMAMUX_DisablePeriodTrigger (DMAMUX_Type * *base*, uint32_t *channel*) [inline], [static]

This function disables the DMAMUX period trigger.

Parameters

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

Chapter 11

C90TFS Flash Driver

11.1 Overview

The flash provides the C90TFS Flash driver of Kinetis 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 }`

Enumeration for various flash properties.

- enum `_flash_execute_in_ram_function_constants` {
 `kFLASH_ExecuteInRamFunctionMaxSizeInWords = 16U,`
 `kFLASH_ExecuteInRamFunctionTotalNum = 2U }`

Constants for execute-in-RAM flash function.

- enum `flash_read_resource_option_t` {
 `kFLASH_ResourceOptionFlashIfr,`
 `kFLASH_ResourceOptionVersionId = 0x01U }`

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

- enum `_flash_read_resource_range` {

```

kFLASH_ResourceRangePflashIfrSizeInBytes = 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 `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.

- enum `flash_memory_index_t` {
 kFLASH_MemoryIndexPrimaryFlash = 0x00U,
 kFLASH_MemoryIndexSecondaryFlash = 0x01U }

Enumeration for the flash memory index.

- enum `flash_prefetch_speculation_option_t`

Overview

Enumeration for the two possible options of flash prefetch speculation.

Flash version

- enum `_flash_driver_version_constants` {
 `kFLASH_DriverVersionName` = 'F',
 `kFLASH_DriverVersionMajor` = 2,
 `kFLASH_DriverVersionMinor` = 2,
 `kFLASH_DriverVersionBugfix` = 0 }
 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, 2, 0))
 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_IS_SECONDARY_FLASH_SUPPORTED` (0)
 Indicates whether the secondary flash is supported in the Flash driver.
- #define `FLASH_SSD_SECONDARY_FLASH_HAS_ITS_OWN_PROTECTION_REGISTER` (0)
 Indicates whether the secondary flash has its own protection register in flash module.
- #define `FLASH_SSD_SECONDARY_FLASH_HAS_ITS_OWN_ACCESS_REGISTER` (0)
 Indicates whether the secondary flash has its own Execute-Only access register in flash module.
- #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_EraseAllUnsecure** (**flash_config_t** *config, uint32_t key)
Erases the entire flash, including protected sectors.
- status_t **FLASH_EraseAllExecuteOnlySegments** (**flash_config_t** *config, uint32_t key)
Erases all program flash execute-only segments defined by the FXACC registers.

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

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.

<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_SetFlexramAsRamError</i>	Failed to set flexram as RAM.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_RecoverFlexramAsEepromError</i>	Failed to recover FlexRAM as EEPROM.

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_InvalidArgument</i>	An invalid argument is provided.

Overview

<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_SetFlexramAsEepromError</i>	Failed to set flexram as eeprom.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_RecoverFlexramAsRamError</i>	Failed to recover the FlexRAM as RAM.

- 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.

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>eepromData-SizeCode</i>	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
<i>flexnvm-PartitionCode</i>	Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	Invalid argument is provided.
<i>kStatus_FLASH_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.

Data Structure Documentation

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 prot32b`
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::data-Option`

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 sectorCmdAddressAlignment**
An erase sector command address alignment.
- **uint32_t partCmdAddressAlignment**
A program/verify part command address alignment.
- **32_t resourceCmdAddressAlignment**
A read resource command address alignment.
- **uint32_t checkCmdAddressAlignment**
A program check command address alignment.

Data Structure Documentation

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::sectorCmdAddressAlignment`

11.2.10.0.0.21.6 `uint32_t flash_operation_config_t::partCmdAddressAlignment`

11.2.10.0.0.21.7 `uint32_t flash_operation_config_t::resourceCmdAddressAlignment`

11.2.10.0.0.21.8 `uint32_t flash_operation_config_t::checkCmdAddressAlignment`

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.
- `uint32_t PFlashBlockCount`
A number of PFlash blocks.
- `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 For the non-FlexNVM device, this is the base address of the acceleration RAM memory.
- `uint32_t FlexRAMTotalSize`
For the FlexNVM device, this is the size of the FlexRAM For the non-FlexNVM device, this is the size of the acceleration RAM memory.
- `uint32_t DFlashBlockBase`
For the FlexNVM device, this is the base address of the D-Flash memory (FlexNVM memory) For the

- *non-FlexNVM device, this field is unused.*

- **uint32_t DFlashTotalSize**
For the FlexNVM device, this is the total size of the FlexNVM memory; For the non-FlexNVM device, this field is unused.

- **uint32_t EEpromTotalSize**

- *For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM For the non-FlexNVM device, this field is unused.*

- **uint32_t FlashMemoryIndex**

- *0 - primary flash; 1 - secondary flash*

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 uint32_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

11.3 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, 2, 0))

Version 2.2.0.

11.3.3 #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Enables the FlexNVM support by default.

11.3.4 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for the flash resident application.

Enumeration Type Documentation

11.3.5 #define FLASH_DRIVER_IS_EXPORTED 0

Used for the KSDK application.

11.3.6 #define kStatusGroupGeneric 0

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

11.3.8 #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.

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.

Enumeration Type Documentation

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.

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.
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 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.

Enumeration Type Documentation

11.4.13 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.

11.4.14 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.

11.4.15 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.

11.4.16 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.

11.4.17 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.

11.4.18 enum flash_memory_index_t

Enumerator

kFLASH_MemoryIndexPrimaryFlash Current flash memory is primary flash.

kFLASH_MemoryIndexSecondaryFlash Current flash memory is secondary flash.

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_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

11.5.2 status_t FLASH_SetCallback (flash_config_t * config, flash_callback_t callback)

Function Documentation

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_PreparesExecuteInRamFunctions (*flash_config_t * config*)

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
---------------	--

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.

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.
<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.

Function Documentation

<i>kStatus_FLASH_AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FLASH_AddressError</i>	The address is out of range.
<i>kStatus_FLASH_EraseKeyError</i>	The API erase key is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

11.5.6 **status_t FLASH_EraseAllUnsecure (flash_config_t * config, uint32_t key)**

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
<i>key</i>	A value used to validate all flash erase APIs.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_EraseKeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.

<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

11.5.7 **status_t FLASH_EraseAllExecuteOnlySegments (flash_config_t * config, uint32_t key)**

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
<i>key</i>	A value used to validate all flash erase APIs.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_EraseKeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

11.5.8 **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.

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 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.

11.5.9 **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.
<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.

11.5.10 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.

Function Documentation

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.

11.5.11 **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.
<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.

11.5.12 **status_t FLASH_GetSecurityState (flash_config_t * config, flash_security_state_t * state)**

This function retrieves the current flash security status, including the security enabling state and the backdoor key enabling state.

Parameters

<i>config</i>	A pointer to storage for the driver runtime state.
<i>state</i>	A pointer to the value returned for the current security status code:

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.

11.5.13 **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.

Function Documentation

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.

11.5.14 **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.
<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.

11.5.15 **status_t FLASH_VerifyErase (flash_config_t * config, uint32_t start, uint32_t lengthInBytes, flash_margin_value_t margin)**

This function checks the appropriate number of flash sectors based on the desired start address and length to check whether the flash is erased to the specified read margin level.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be verified. The start address does not need to be sector-aligned but must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<i>margin</i>	Read margin choice.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.

Function Documentation

<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.

11.5.16 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_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_AddressError</i>	Address is out of range.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during the command execution.

11.5.17 **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_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.

Function Documentation

<i>kStatus_FLASH_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.

11.5.18 **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.

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.

11.5.19 **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.

11.5.20 `status_t FLASHGetProperty (flash_config_t * config, flash_property_tag_t whichProperty, uint32_t * value)`

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>whichProperty</i>	The desired property from the list of properties in enum flash_property_tag_t
<i>value</i>	A pointer to the value returned for the desired flash property.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_UnknownProperty</i>	An unknown property tag.

Function Documentation

**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_Invalid-Argument</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.

Function Documentation

Chapter 12

FlexIO: FlexIO Driver

12.1 Overview

The KSDK provides a generic driver and multiple protocol-specific FlexIO drivers for the FlexIO module of Kinetis devices.

Modules

- [FlexIO Camera Driver](#)
- [FlexIO Driver](#)
- [FlexIO I2C Master Driver](#)
- [FlexIO I2S Driver](#)
- [FlexIO SPI Driver](#)
- [FlexIO UART Driver](#)

FlexIO Driver

12.2 FlexIO Driver

12.2.1 Overview

Data Structures

- struct `flexio_config_t`
Define FlexIO user configuration structure. [More...](#)
- struct `flexio_timer_config_t`
Define FlexIO timer configuration structure. [More...](#)
- struct `flexio_shifter_config_t`
Define FlexIO shifter configuration structure. [More...](#)

Macros

- #define `FLEXIO_TIMER_TRIGGER_SEL_PININPUT`(x) ((uint32_t)(x) << 1U)
Calculate FlexIO timer trigger.

Typedefs

- typedef void(* `flexio_isr_t`)(void *base, void *handle)
typedef for FlexIO simulated driver interrupt handler.

Enumerations

- enum `flexio_timer_trigger_polarity_t` {
 `kFLEXIO_TimerTriggerPolarityActiveHigh` = 0x0U,
 `kFLEXIO_TimerTriggerPolarityActiveLow` = 0x1U }
Define time of timer trigger polarity.
- enum `flexio_timer_trigger_source_t` {
 `kFLEXIO_TimerTriggerSourceExternal` = 0x0U,
 `kFLEXIO_TimerTriggerSourceInternal` = 0x1U }
Define type of timer trigger source.
- enum `flexio_pin_config_t` {
 `kFLEXIO_PinConfigOutputDisabled` = 0x0U,
 `kFLEXIO_PinConfigOpenDrainOrBidirection` = 0x1U,
 `kFLEXIO_PinConfigBidirectionOutputData` = 0x2U,
 `kFLEXIO_PinConfigOutput` = 0x3U }
Define type of timer/shifter pin configuration.
- enum `flexio_pin_polarity_t` {
 `kFLEXIO_PinActiveHigh` = 0x0U,
 `kFLEXIO_PinActiveLow` = 0x1U }
Definition of pin polarity.

- enum `flexio_timer_mode_t` {

 `kFLEXIO_TimerModeDisabled` = 0x0U,

 `kFLEXIO_TimerModeDual8BitBaudBit` = 0x1U,

 `kFLEXIO_TimerModeDual8BitPWM` = 0x2U,

 `kFLEXIO_TimerModeSingle16Bit` = 0x3U }

 Define type of timer work mode.
- enum `flexio_timer_output_t` {

 `kFLEXIO_TimerOutputOneNotAffectedByReset` = 0x0U,

 `kFLEXIO_TimerOutputZeroNotAffectedByReset` = 0x1U,

 `kFLEXIO_TimerOutputOneAffectedByReset` = 0x2U,

 `kFLEXIO_TimerOutputZeroAffectedByReset` = 0x3U }

 Define type of timer initial output or timer reset condition.
- enum `flexio_timer_decrement_source_t` {

 `kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput` = 0x0U,

 `kFLEXIO_TimerDecSrcOnTriggerInputShiftTimerOutput` = 0x1U,

 `kFLEXIO_TimerDecSrcOnPinInputShiftPinInput` = 0x2U,

 `kFLEXIO_TimerDecSrcOnTriggerInputShiftTriggerInput` = 0x3U }

 Define type of timer decrement.
- enum `flexio_timer_reset_condition_t` {

 `kFLEXIO_TimerResetNever` = 0x0U,

 `kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput` = 0x2U,

 `kFLEXIO_TimerResetOnTimerTriggerEqualToTimerOutput` = 0x3U,

 `kFLEXIO_TimerResetOnTimerPinRisingEdge` = 0x4U,

 `kFLEXIO_TimerResetOnTimerTriggerRisingEdge` = 0x6U,

 `kFLEXIO_TimerResetOnTimerTriggerBothEdge` = 0x7U }

 Define type of timer reset condition.
- enum `flexio_timer_disable_condition_t` {

 `kFLEXIO_TimerDisableNever` = 0x0U,

 `kFLEXIO_TimerDisableOnPreTimerDisable` = 0x1U,

 `kFLEXIO_TimerDisableOnTimerCompare` = 0x2U,

 `kFLEXIO_TimerDisableOnTimerCompareTriggerLow` = 0x3U,

 `kFLEXIO_TimerDisableOnPinBothEdge` = 0x4U,

 `kFLEXIO_TimerDisableOnPinBothEdgeTriggerHigh` = 0x5U,

 `kFLEXIO_TimerDisableOnTriggerFallingEdge` = 0x6U }

 Define type of timer disable condition.
- enum `flexio_timer_enable_condition_t` {

 `kFLEXIO_TimerEnabledAlways` = 0x0U,

 `kFLEXIO_TimerEnableOnPrevTimerEnable` = 0x1U,

 `kFLEXIO_TimerEnableOnTriggerHigh` = 0x2U,

 `kFLEXIO_TimerEnableOnTriggerHighPinHigh` = 0x3U,

 `kFLEXIO_TimerEnableOnPinRisingEdge` = 0x4U,

 `kFLEXIO_TimerEnableOnPinRisingEdgeTriggerHigh` = 0x5U,

 `kFLEXIO_TimerEnableOnTriggerRisingEdge` = 0x6U,

 `kFLEXIO_TimerEnableOnTriggerBothEdge` = 0x7U }

 Define type of timer enable condition.
- enum `flexio_timer_stop_bit_condition_t` {

FlexIO Driver

```
kFLEXIO_TimerStopBitDisabled = 0x0U,  
kFLEXIO_TimerStopBitEnableOnTimerCompare = 0x1U,  
kFLEXIO_TimerStopBitEnableOnTimerDisable = 0x2U,  
kFLEXIO_TimerStopBitEnableOnTimerCompareDisable = 0x3U }
```

Define type of timer stop bit generate condition.

- enum `flexio_timer_start_bit_condition_t`{
 kFLEXIO_TimerStartBitDisabled = 0x0U,
 kFLEXIO_TimerStartBitEnabled = 0x1U }

Define type of timer start bit generate condition.

- enum `flexio_shifter_timer_polarity_t`
Define type of timer polarity for shifter control.
- enum `flexio_shifter_mode_t`{

```
    kFLEXIO_ShifterDisabled = 0x0U,  
    kFLEXIO_ShifterModeReceive = 0x1U,  
    kFLEXIO_ShifterModeTransmit = 0x2U,  
    kFLEXIO_ShifterModeMatchStore = 0x4U,  
    kFLEXIO_ShifterModeMatchContinuous = 0x5U }
```

Define type of shifter working mode.

- enum `flexio_shifter_input_source_t`{
 kFLEXIO_ShifterInputFromPin = 0x0U,
 kFLEXIO_ShifterInputFromNextShifterOutput = 0x1U }

Define type of shifter input source.

- enum `flexio_shifter_stop_bit_t`{
 kFLEXIO_ShifterStopBitDisable = 0x0U,
 kFLEXIO_ShifterStopBitLow = 0x2U,
 kFLEXIO_ShifterStopBitHigh = 0x3U }

Define of STOP bit configuration.

- enum `flexio_shifter_start_bit_t`{
 kFLEXIO_ShifterStartBitDisabledLoadDataOnEnable = 0x0U,
 kFLEXIO_ShifterStartBitDisabledLoadDataOnShift = 0x1U,
 kFLEXIO_ShifterStartBitLow = 0x2U,
 kFLEXIO_ShifterStartBitHigh = 0x3U }

Define type of START bit configuration.

- enum `flexio_shifter_buffer_type_t`{
 kFLEXIO_ShifterBuffer = 0x0U,
 kFLEXIO_ShifterBufferBitSwapped = 0x1U,
 kFLEXIO_ShifterBufferByteSwapped = 0x2U,
 kFLEXIO_ShifterBufferBitByteSwapped = 0x3U }

Define FlexIO shifter buffer type.

Driver version

- #define `FSL_FLEXIO_DRIVER_VERSION`(`MAKE_VERSION(2, 0, 1)`)
FlexIO driver version 2.0.1.

FlexIO Initialization and De-initialization

- void **FLEXIO_GetDefaultConfig** (**flexio_config_t** *userConfig)
Gets the default configuration to configure the FlexIO module.
- void **FLEXIO_Init** (**FLEXIO_Type** *base, const **flexio_config_t** *userConfig)
Configures the FlexIO with a FlexIO configuration.
- void **FLEXIO_Deinit** (**FLEXIO_Type** *base)
Gates the FlexIO clock.

FlexIO Basic Operation

- void **FLEXIO_Reset** (**FLEXIO_Type** *base)
Resets the FlexIO module.
- static void **FLEXIO_Enable** (**FLEXIO_Type** *base, bool enable)
Enables the FlexIO module operation.
- void **FLEXIO_SetShifterConfig** (**FLEXIO_Type** *base, uint8_t index, const **flexio_shifter_config_t** *shifterConfig)
Configures the shifter with the shifter configuration.
- void **FLEXIO_SetTimerConfig** (**FLEXIO_Type** *base, uint8_t index, const **flexio_timer_config_t** *timerConfig)
Configures the timer with the timer configuration.

FlexIO Interrupt Operation

- static void **FLEXIO_EnableShifterStatusInterrupts** (**FLEXIO_Type** *base, uint32_t mask)
Enables the shifter status interrupt.
- static void **FLEXIO_DisableShifterStatusInterrupts** (**FLEXIO_Type** *base, uint32_t mask)
Disables the shifter status interrupt.
- static void **FLEXIO_EnableShifterErrorInterrupts** (**FLEXIO_Type** *base, uint32_t mask)
Enables the shifter error interrupt.
- static void **FLEXIO_DisableShifterErrorInterrupts** (**FLEXIO_Type** *base, uint32_t mask)
Disables the shifter error interrupt.
- static void **FLEXIO_EnableTimerStatusInterrupts** (**FLEXIO_Type** *base, uint32_t mask)
Enables the timer status interrupt.
- static void **FLEXIO_DisableTimerStatusInterrupts** (**FLEXIO_Type** *base, uint32_t mask)
Disables the timer status interrupt.

FlexIO Status Operation

- static uint32_t **FLEXIO_GetShifterStatusFlags** (**FLEXIO_Type** *base)
Gets the shifter status flags.
- static void **FLEXIO_ClearShifterStatusFlags** (**FLEXIO_Type** *base, uint32_t mask)
Clears the shifter status flags.
- static uint32_t **FLEXIO_GetShifterErrorFlags** (**FLEXIO_Type** *base)
Gets the shifter error flags.
- static void **FLEXIO_ClearShifterErrorFlags** (**FLEXIO_Type** *base, uint32_t mask)
Clears the shifter error flags.

FlexIO Driver

- static uint32_t **FLEXIO_GetTimerStatusFlags** (FLEXIO_Type *base)
Gets the timer status flags.
- static void **FLEXIO_ClearTimerStatusFlags** (FLEXIO_Type *base, uint32_t mask)
Clears the timer status flags.

FlexIO DMA Operation

- static void **FLEXIO_EnableShifterStatusDMA** (FLEXIO_Type *base, uint32_t mask, bool enable)
Enables/disables the shifter status DMA.
- uint32_t **FLEXIO_GetShifterBufferAddress** (FLEXIO_Type *base, flexio_shifter_buffer_type_t type, uint8_t index)
Gets the shifter buffer address for the DMA transfer usage.
- status_t **FLEXIO_RegisterHandleIRQ** (void *base, void *handle, flexio_isr_t isr)
Registers the handle and the interrupt handler for the FlexIO-simulated peripheral.
- status_t **FLEXIO_UnregisterHandleIRQ** (void *base)
Unregisters the handle and the interrupt handler for the FlexIO-simulated peripheral.

12.2.2 Data Structure Documentation

12.2.2.1 struct flexio_config_t

Data Fields

- bool **enableFlexio**
Enable/disable FlexIO module.
- bool **enableInDoze**
Enable/disable FlexIO operation in doze mode.
- bool **enableInDebug**
Enable/disable FlexIO operation in debug mode.
- bool **enableFastAccess**
Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.

12.2.2.1.0.23 Field Documentation

12.2.2.1.0.23.1 bool flexio_config_t::enableFastAccess

12.2.2.2 struct flexio_timer_config_t

Data Fields

- uint32_t **triggerSelect**
The internal trigger selection number using MACROs.
- flexio_timer_trigger_polarity_t **triggerPolarity**
Trigger Polarity.
- flexio_timer_trigger_source_t **triggerSource**
Trigger Source, internal (see 'trgsel') or external.
- flexio_pin_config_t **pinConfig**

Timer Pin Configuration.

- **uint32_t pinSelect**
Timer Pin number Select.
- **flexio_pin_polarity_t pinPolarity**
Timer Pin Polarity.
- **flexio_timer_mode_t timerMode**
Timer work Mode.
- **flexio_timer_output_t timerOutput**
Configures the initial state of the Timer Output and
whether it is affected by the Timer reset.
- **flexio_timer_decrement_source_t timerDecrement**
Configures the source of the Timer decrement and the
source of the Shift clock.
- **flexio_timer_reset_condition_t timerReset**
Configures the condition that causes the timer counter
(and optionally the timer output) to be reset.
- **flexio_timer_disable_condition_t timerDisable**
Configures the condition that causes the Timer to be
disabled and stop decrementing.
- **flexio_timer_enable_condition_t timerEnable**
Configures the condition that causes the Timer to be
enabled and start decrementing.
- **flexio_timer_stop_bit_condition_t timerStop**
Timer STOP Bit generation.
- **flexio_timer_start_bit_condition_t timerStart**
Timer STRAT Bit generation.
- **uint32_t timerCompare**
Value for Timer Compare N Register.

FlexIO Driver

12.2.2.2.0.24 Field Documentation

12.2.2.2.0.24.1 `uint32_t flexio_timer_config_t::triggerSelect`

12.2.2.2.0.24.2 `flexio_timer_polarity_t flexio_timer_config_t::triggerPolarity`

12.2.2.2.0.24.3 `flexio_timer_trigger_source_t flexio_timer_config_t::triggerSource`

12.2.2.2.0.24.4 `flexio_pin_config_t flexio_timer_config_t::pinConfig`

12.2.2.2.0.24.5 `uint32_t flexio_timer_config_t::pinSelect`

12.2.2.2.0.24.6 `flexio_pin_polarity_t flexio_timer_config_t::pinPolarity`

12.2.2.2.0.24.7 `flexio_timer_mode_t flexio_timer_config_t::timerMode`

12.2.2.2.0.24.8 `flexio_timer_output_t flexio_timer_config_t::timerOutput`

12.2.2.2.0.24.9 `flexio_timer_decrement_source_t flexio_timer_config_t::timerDecrement`

12.2.2.2.0.24.10 `flexio_timer_reset_condition_t flexio_timer_config_t::timerReset`

12.2.2.2.0.24.11 `flexio_timer_disable_condition_t flexio_timer_config_t::timerDisable`

12.2.2.2.0.24.12 `flexio_timer_enable_condition_t flexio_timer_config_t::timerEnable`

12.2.2.2.0.24.13 `flexio_timer_stop_bit_condition_t flexio_timer_config_t::timerStop`

12.2.2.2.0.24.14 `flexio_timer_start_bit_condition_t flexio_timer_config_t::timerStart`

12.2.2.2.0.24.15 `uint32_t flexio_timer_config_t::timerCompare`

12.2.2.3 `struct flexio_shifter_config_t`

Data Fields

- `uint32_t timerSelect`
Selects which Timer is used for controlling the logic/shift register and generating the Shift clock.
- `flexio_shifter_timer_polarity_t timerPolarity`
Timer Polarity.
- `flexio_pin_config_t pinConfig`
Shifter Pin Configuration.
- `uint32_t pinSelect`
Shifter Pin number Select.
- `flexio_pin_polarity_t pinPolarity`
Shifter Pin Polarity.
- `flexio_shifter_mode_t shifterMode`
Configures the mode of the Shifter.
- `flexio_shifter_input_source_t inputSource`
Selects the input source for the shifter.

- **flexio_shifter_stop_bit_t shifterStop**
Shifter STOP bit.
- **flexio_shifter_start_bit_t shifterStart**
Shifter START bit.

12.2.2.3.0.25 Field Documentation

12.2.2.3.0.25.1 `uint32_t flexio_shifter_config_t::timerSelect`

12.2.2.3.0.25.2 `flexio_shifter_timer_polarity_t flexio_shifter_config_t::timerPolarity`

12.2.2.3.0.25.3 `flexio_pin_config_t flexio_shifter_config_t::pinConfig`

12.2.2.3.0.25.4 `uint32_t flexio_shifter_config_t::pinSelect`

12.2.2.3.0.25.5 `flexio_pin_polarity_t flexio_shifter_config_t::pinPolarity`

12.2.2.3.0.25.6 `flexio_shifter_mode_t flexio_shifter_config_t::shifterMode`

12.2.2.3.0.25.7 `flexio_shifter_input_source_t flexio_shifter_config_t::inputSource`

12.2.2.3.0.25.8 `flexio_shifter_stop_bit_t flexio_shifter_config_t::shifterStop`

12.2.2.3.0.25.9 `flexio_shifter_start_bit_t flexio_shifter_config_t::shifterStart`

12.2.3 Macro Definition Documentation

12.2.3.1 `#define FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))`

12.2.3.2 `#define FLEXIO_TIMER_TRIGGER_SEL_PININPUT(x) ((uint32_t)(x) << 1U)`

12.2.4 Typedef Documentation

12.2.4.1 `typedef void(* flexio_isr_t)(void *base, void *handle)`

12.2.5 Enumeration Type Documentation

12.2.5.1 `enum flexio_timer_trigger_polarity_t`

Enumerator

kFLEXIO_TimerTriggerPolarityActiveHigh Active high.

kFLEXIO_TimerTriggerPolarityActiveLow Active low.

FlexIO Driver

12.2.5.2 enum flexio_timer_trigger_source_t

Enumerator

kFLEXIO_TimerTriggerSourceExternal External trigger selected.

kFLEXIO_TimerTriggerSourceInternal Internal trigger selected.

12.2.5.3 enum flexio_pin_config_t

Enumerator

kFLEXIO_PinConfigOutputDisabled Pin output disabled.

kFLEXIO_PinConfigOpenDrainOrBidirection Pin open drain or bidirectional output enable.

kFLEXIO_PinConfigBidirectionOutputData Pin bidirectional output data.

kFLEXIO_PinConfigOutput Pin output.

12.2.5.4 enum flexio_pin_polarity_t

Enumerator

kFLEXIO_PinActiveHigh Active high.

kFLEXIO_PinActiveLow Active low.

12.2.5.5 enum flexio_timer_mode_t

Enumerator

kFLEXIO_TimerModeDisabled Timer Disabled.

kFLEXIO_TimerModeDual8BitBaudBit Dual 8-bit counters baud/bit mode.

kFLEXIO_TimerModeDual8BitPWM Dual 8-bit counters PWM mode.

kFLEXIO_TimerModeSingle16Bit Single 16-bit counter mode.

12.2.5.6 enum flexio_timer_output_t

Enumerator

kFLEXIO_TimerOutputOneNotAffectedByReset Logic one when enabled and is not affected by timer reset.

kFLEXIO_TimerOutputZeroNotAffectedByReset Logic zero when enabled and is not affected by timer reset.

kFLEXIO_TimerOutputOneAffectedByReset Logic one when enabled and on timer reset.

kFLEXIO_TimerOutputZeroAffectedByReset Logic zero when enabled and on timer reset.

12.2.5.7 enum flexio_timer_decrement_source_t

Enumerator

- kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput*** Decrement counter on FlexIO clock, Shift clock equals Timer output.
- kFLEXIO_TimerDecSrcOnTriggerInputShiftTimerOutput*** Decrement counter on Trigger input (both edges), Shift clock equals Timer output.
- kFLEXIO_TimerDecSrcOnPinInputShiftPinInput*** Decrement counter on Pin input (both edges), Shift clock equals Pin input.
- kFLEXIO_TimerDecSrcOnTriggerInputShiftTriggerInput*** Decrement counter on Trigger input (both edges), Shift clock equals Trigger input.

12.2.5.8 enum flexio_timer_reset_condition_t

Enumerator

- kFLEXIO_TimerResetNever*** Timer never reset.
- kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput*** Timer reset on Timer Pin equal to Timer Output.
- kFLEXIO_TimerResetOnTimerTriggerEqualToTimerOutput*** Timer reset on Timer Trigger equal to Timer Output.
- kFLEXIO_TimerResetOnTimerPinRisingEdge*** Timer reset on Timer Pin rising edge.
- kFLEXIO_TimerResetOnTimerTriggerRisingEdge*** Timer reset on Trigger rising edge.
- kFLEXIO_TimerResetOnTimerTriggerBothEdge*** Timer reset on Trigger rising or falling edge.

12.2.5.9 enum flexio_timer_disable_condition_t

Enumerator

- kFLEXIO_TimerDisableNever*** Timer never disabled.
- kFLEXIO_TimerDisableOnPreTimerDisable*** Timer disabled on Timer N-1 disable.
- kFLEXIO_TimerDisableOnTimerCompare*** Timer disabled on Timer compare.
- kFLEXIO_TimerDisableOnTimerCompareTriggerLow*** Timer disabled on Timer compare and Trigger Low.
- kFLEXIO_TimerDisableOnPinBothEdge*** Timer disabled on Pin rising or falling edge.
- kFLEXIO_TimerDisableOnPinBothEdgeTriggerHigh*** Timer disabled on Pin rising or falling edge provided Trigger is high.
- kFLEXIO_TimerDisableOnTriggerFallingEdge*** Timer disabled on Trigger falling edge.

12.2.5.10 enum flexio_timer_enable_condition_t

Enumerator

- kFLEXIO_TimerEnabledAlways*** Timer always enabled.

FlexIO Driver

kFLEXIO_TimerEnableOnPrevTimerEnable Timer enabled on Timer N-1 enable.
kFLEXIO_TimerEnableOnTriggerHigh Timer enabled on Trigger high.
kFLEXIO_TimerEnableOnTriggerHighPinHigh Timer enabled on Trigger high and Pin high.
kFLEXIO_TimerEnableOnPinRisingEdge Timer enabled on Pin rising edge.
kFLEXIO_TimerEnableOnPinRisingEdgeTriggerHigh Timer enabled on Pin rising edge and Trigger high.
kFLEXIO_TimerEnableOnTriggerRisingEdge Timer enabled on Trigger rising edge.
kFLEXIO_TimerEnableOnTriggerBothEdge Timer enabled on Trigger rising or falling edge.

12.2.5.11 enum flexio_timer_stop_bit_condition_t

Enumerator

kFLEXIO_TimerStopBitDisabled Stop bit disabled.
kFLEXIO_TimerStopBitEnableOnTimerCompare Stop bit is enabled on timer compare.
kFLEXIO_TimerStopBitEnableOnTimerDisable Stop bit is enabled on timer disable.
kFLEXIO_TimerStopBitEnableOnTimerCompareDisable Stop bit is enabled on timer compare and timer disable.

12.2.5.12 enum flexio_timer_start_bit_condition_t

Enumerator

kFLEXIO_TimerStartBitDisabled Start bit disabled.
kFLEXIO_TimerStartBitEnabled Start bit enabled.

12.2.5.13 enum flexio_shifter_timer_polarity_t

12.2.5.14 enum flexio_shifter_mode_t

Enumerator

kFLEXIO_ShifterDisabled Shifter is disabled.
kFLEXIO_ShifterModeReceive Receive mode.
kFLEXIO_ShifterModeTransmit Transmit mode.
kFLEXIO_ShifterModeMatchStore Match store mode.
kFLEXIO_ShifterModeMatchContinuous Match continuous mode.

12.2.5.15 enum flexio_shifter_input_source_t

Enumerator

kFLEXIO_ShifterInputFromPin Shifter input from pin.
kFLEXIO_ShifterInputFromNextShifterOutput Shifter input from Shifter N+1.

12.2.5.16 enum flexio_shifter_stop_bit_t

Enumerator

kFLEXIO_ShifterStopBitDisable Disable shifter stop bit.

kFLEXIO_ShifterStopBitLow Set shifter stop bit to logic low level.

kFLEXIO_ShifterStopBitHigh Set shifter stop bit to logic high level.

12.2.5.17 enum flexio_shifter_start_bit_t

Enumerator

kFLEXIO_ShifterStartBitDisabledLoadDataOnEnable Disable shifter start bit, transmitter loads data on enable.

kFLEXIO_ShifterStartBitDisabledLoadDataOnShift Disable shifter start bit, transmitter loads data on first shift.

kFLEXIO_ShifterStartBitLow Set shifter start bit to logic low level.

kFLEXIO_ShifterStartBitHigh Set shifter start bit to logic high level.

12.2.5.18 enum flexio_shifter_buffer_type_t

Enumerator

kFLEXIO_ShifterBuffer Shifter Buffer N Register.

kFLEXIO_ShifterBufferBitSwapped Shifter Buffer N Bit Byte Swapped Register.

kFLEXIO_ShifterBufferByteSwapped Shifter Buffer N Byte Swapped Register.

kFLEXIO_ShifterBufferBitByteSwapped Shifter Buffer N Bit Swapped Register.

12.2.6 Function Documentation

12.2.6.1 void FLEXIO_GetDefaultConfig (***flexio_config_t * userConfig***)

The configuration can used directly to call the FLEXIO_Configure().

Example:

```
flexio_config_t config;
FLEXIO_GetDefaultConfig(&config);
```

FlexIO Driver

Parameters

<i>userConfig</i>	pointer to flexio_config_t structure
-------------------	--

12.2.6.2 void FLEXIO_Init ([FLEXIO_Type](#) * *base*, const [flexio_config_t](#) * *userConfig*)

The configuration structure can be filled by the user or be set with default values by [FLEXIO_GetDefaultConfig\(\)](#).

Example

```
flexio_config_t config = {  
    .enableFlexio = true,  
    .enableInDoze = false,  
    .enableInDebug = true,  
    .enableFastAccess = false  
};  
FLEXIO_Configure(base, &config);
```

Parameters

<i>base</i>	FlexIO peripheral base address
<i>userConfig</i>	pointer to flexio_config_t structure

12.2.6.3 void FLEXIO_Deinit ([FLEXIO_Type](#) * *base*)

Call this API to stop the FlexIO clock.

Note

After calling this API, call the FLEXIO_Init to use the FlexIO module.

Parameters

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

12.2.6.4 void FLEXIO_Reset ([FLEXIO_Type](#) * *base*)

Parameters

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

12.2.6.5 static void FLEXIO_Enable (FLEXIO_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	FlexIO peripheral base address
<i>enable</i>	true to enable, false to disable.

12.2.6.6 void FLEXIO_SetShifterConfig (FLEXIO_Type * *base*, uint8_t *index*, const flexio_shifter_config_t * *shifterConfig*)

The configuration structure covers both the SHIFTCTL and SHIFTCFG registers. To configure the shifter to the proper mode, select which timer controls the shifter to shift, whether to generate start bit/stop bit, and the polarity of start bit and stop bit.

Example

```
flexio_shifter_config_t config = {
.timerSelect = 0,
.timerPolarity = kFLEXIO_ShifterTimerPolarityOnPositive,
.pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,
.pinPolarity = kFLEXIO_PinActiveLow,
.shifterMode = kFLEXIO_ShifterModeTransmit,
.inputSource = kFLEXIO_ShifterInputFromPin,
.shifterStop = kFLEXIO_ShifterStopBitHigh,
.shifterStart = kFLEXIO_ShifterStartBitLow
};
FLEXIO_SetShifterConfig(base, &config);
```

Parameters

<i>base</i>	FlexIO peripheral base address
<i>index</i>	Shifter index
<i>shifterConfig</i>	Pointer to flexio_shifter_config_t structure

12.2.6.7 void FLEXIO_SetTimerConfig (FLEXIO_Type * *base*, uint8_t *index*, const flexio_timer_config_t * *timerConfig*)

The configuration structure covers both the TIMCTL and TIMCFG registers. To configure the timer to the proper mode, select trigger source for timer and the timer pin output and the timing for timer.

FlexIO Driver

Example

```
flexio_timer_config_t config = {  
    .triggerSelect = FLEXIO_TIMER_TRIGGER_SEL_SHIFTnSTAT(0),  
    .triggerPolarity = kFLEXIO_TimerTriggerPolarityActiveLow,  
    .triggerSource = kFLEXIO_TimerTriggerSourceInternal,  
    .pinConfig = kFLEXIO_PinConfigOpenDrainOrBidirection,  
    .pinSelect = 0,  
    .pinPolarity = kFLEXIO_PinActiveHigh,  
    .timerMode = kFLEXIO_TimerModeDual8BitBaudBit,  
    .timerOutput = kFLEXIO_TimerOutputZeroNotAffectedByReset,  
    .timerDecrement = kFLEXIO_TimerDecSrcOnFlexIOClockShiftTimerOutput  
    ,  
    .timerReset = kFLEXIO_TimerResetOnTimerPinEqualToTimerOutput,  
    .timerDisable = kFLEXIO_TimerDisableOnTimerCompare,  
    .timerEnable = kFLEXIO_TimerEnableOnTriggerHigh,  
    .timerStop = kFLEXIO_TimerStopBitEnableOnTimerDisable,  
    .timerStart = kFLEXIO_TimerStartBitEnabled  
};  
FLEXIO_SetTimerConfig(base, &config);
```

Parameters

<i>base</i>	FlexIO peripheral base address
<i>index</i>	Timer index
<i>timerConfig</i>	Pointer to the flexio_timer_config_t structure

12.2.6.8 static void FLEXIO_EnableShifterStatusInterrupts (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

The interrupt generates when the corresponding SSF is set.

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter status mask which can be calculated by $(1 \ll \text{shifter index})$

Note

For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using $((1 \ll \text{shifter index}0) | (1 \ll \text{shifter index}1))$

12.2.6.9 static void FLEXIO_DisableShifterStatusInterrupts (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

The interrupt won't generate when the corresponding SSF is set.

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter status mask which can be calculated by ($1 << \text{shifter index}$)

Note

For multiple shifter status interrupt enable, for example, two shifter status enable, can calculate the mask by using $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

12.2.6.10 static void FLEXIO_EnableShifterErrorInterrupts (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

The interrupt generates when the corresponding SEF is set.

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter error mask which can be calculated by ($1 << \text{shifter index}$)

Note

For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

12.2.6.11 static void FLEXIO_DisableShifterErrorInterrupts (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

The interrupt won't generate when the corresponding SEF is set.

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter error mask which can be calculated by ($1 << \text{shifter index}$)

Note

For multiple shifter error interrupt enable, for example, two shifter error enable, can calculate the mask by using $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

**12.2.6.12 static void FLEXIO_EnableTimerStatusInterrupts (FLEXIO_Type * *base*,
 uint32_t *mask*) [inline], [static]**

The interrupt generates when the corresponding SSF is set.

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The timer status mask which can be calculated by ($1 << \text{timer index}$)

Note

For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 << \text{timer index}0) | (1 << \text{timer index}1))$

12.2.6.13 static void FLEXIO_DisableTimerStatusInterrupts (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

The interrupt won't generate when the corresponding SSF is set.

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The timer status mask which can be calculated by ($1 << \text{timer index}$)

Note

For multiple timer status interrupt enable, for example, two timer status enable, can calculate the mask by using $((1 << \text{timer index}0) | (1 << \text{timer index}1))$

12.2.6.14 static uint32_t FLEXIO_GetShifterStatusFlags (FLEXIO_Type * *base*) [inline], [static]

Parameters

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

Returns

Shifter status flags

12.2.6.15 static void FLEXIO_ClearShifterStatusFlags (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

FlexIO Driver

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter status mask which can be calculated by ($1 << \text{shifter index}$)

Note

For clearing multiple shifter status flags, for example, two shifter status flags, can calculate the mask by using $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

**12.2.6.16 static uint32_t FLEXIO_GetShifterErrorFlags (FLEXIO_Type * *base*)
[inline], [static]**

Parameters

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

Returns

Shifter error flags

**12.2.6.17 static void FLEXIO_ClearShifterErrorFlags (FLEXIO_Type * *base*, uint32_t
mask) [inline], [static]**

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter error mask which can be calculated by ($1 << \text{shifter index}$)

Note

For clearing multiple shifter error flags, for example, two shifter error flags, can calculate the mask by using $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

**12.2.6.18 static uint32_t FLEXIO_GetTimerStatusFlags (FLEXIO_Type * *base*)
[inline], [static]**

Parameters

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

Returns

Timer status flags

12.2.6.19 static void FLEXIO_ClearTimerStatusFlags (FLEXIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The timer status mask which can be calculated by (1 << timer index)

Note

For clearing multiple timer status flags, for example, two timer status flags, can calculate the mask by using ((1 << timer index0) | (1 << timer index1))

12.2.6.20 static void FLEXIO_EnableShifterStatusDMA (FLEXIO_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

The DMA request generates when the corresponding SSF is set.

Note

For multiple shifter status DMA enables, for example, calculate the mask by using ((1 << shifter index0) | (1 << shifter index1))

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter status mask which can be calculated by (1 << shifter index)

FlexIO Driver

<i>enable</i>	True to enable, false to disable.
---------------	-----------------------------------

12.2.6.21 `uint32_t FLEXIO_GetShifterBufferAddress (FLEXIO_Type * base, flexio_shifter_buffer_type_t type, uint8_t index)`

Parameters

<i>base</i>	FlexIO peripheral base address
<i>type</i>	Shifter type of flexio_shifter_buffer_type_t
<i>index</i>	Shifter index

Returns

Corresponding shifter buffer index

12.2.6.22 `status_t FLEXIO_RegisterHandleIRQ (void * base, void * handle, flexio_isr_t isr)`

Parameters

<i>base</i>	Pointer to the FlexIO simulated peripheral type.
<i>handle</i>	Pointer to the handler for FlexIO simulated peripheral.
<i>isr</i>	FlexIO simulated peripheral interrupt handler.

Return values

<i>kStatus_Success</i>	Successfully create the handle.
<i>kStatus_OutOfRange</i>	The FlexIO type/handle/ISR table out of range.

12.2.6.23 `status_t FLEXIO_UnregisterHandleIRQ (void * base)`

Parameters

<i>base</i>	Pointer to the FlexIO simulated peripheral type.
-------------	--

Return values

<i>kStatus_Success</i>	Successfully create the handle.
<i>kStatus_OutOfRange</i>	The FlexIO type/handle/ISR table out of range.

12.3 FlexIO Camera Driver

12.3.1 Overview

The KSDK provides driver for the camera function using Flexible I/O.

FlexIO Camera driver includes functional APIs and eDMA transactional APIs. Functional APIs target low level APIs. Users can use functional APIs for FlexIO Camera initialization/configuration/operation purpose. Using the functional API requires knowledge of the FlexIO Camera peripheral and how to organize functional APIs to meet the requirements of the application. All functional API use the [FLEXIO_CAMERA_Type](#) * as the first parameter. FlexIO Camera functional operation groups provide the functional APIs set.

eDMA transactional APIs target high-level APIs. Users can use the transactional API to enable the peripheral quickly and can also use in the application if the code size and performance of transactional APIs satisfy requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the `flexio_camera_edma_handle_t` as the second parameter. Users need to initialize the handle by calling the [FLEXIO_CAMERA_TransferCreateHandleEDMA\(\)](#) API.

eDMA transactional APIs support asynchronous receive. This means that the functions [FLEXIO_CAMERA_TransferReceiveEDMA\(\)](#) set up an interrupt for data receive. When the receive is complete, the upper layer is notified through a callback function with the status `kStatus_FLEXIO_CAMERA_RxIdle`.

12.3.2 Typical use case

12.3.2.1 FlexIO Camera Receive using eDMA method

```
volatile uint32_t isEDMAGetOnePictureFinish = false;
edma_handle_t g_edmaHandle;
flexio_camera_edma_handle_t g_cameraEdmaHandle;
edma_config_t edmaConfig;
FLEXIO_CAMERA_Type g_FlexioCameraDevice = {.flexioBase = FLEXIO0,
                                             .datPinStartIdx = 24U, /* fxio_pin 24 -31 are used. */
                                             .pclkPinIdx = 1U,    /* fxio_pin 1 is used as pclk pin. */
                                             .hrefPinIdx = 18U,   /* flexio_pin 18 is used as href pin. */
                                             .shifterStartIdx = 0U, /* Shifter 0 = 7 are used. */
                                             .shifterCount = 8U,
                                             .timerIdx = 0U};

flexio_camera_config_t cameraConfig;

/* Configure DMAMUX */
DMAMUX_Init(DMAMUX0);
/* Configure DMA */
EDMA_GetDefaultConfig(&edmaConfig);
EDMA_Init(DMA0, &edmaConfig);

DMAMUX_SetSource(DMAMUX0, DMA_CHN_FLEXIO_TO_FRAMEBUFF, (g_FlexioCameraDevice.
                                                       shifterStartIdx + 1U));
DMAMUX_EnableChannel(DMAMUX0, DMA_CHN_FLEXIO_TO_FRAMEBUFF);
EDMA_CreateHandle(&g_edmaHandle, DMA0, DMA_CHN_FLEXIO_TO_FRAMEBUFF);

FLEXIO_CAMERA_GetDefaultConfig(&cameraConfig);
FLEXIO_CAMERA_Init(&g_FlexioCameraDevice, &cameraConfig);
/* Clear all the flag. */
```

```

FLEXIO_CAMERA_ClearStatusFlags(&g_FlexioCameraDevice,
                               kFLEXIO_CAMERA_RxDataRegFullFlag |
                               kFLEXIO_CAMERA_RxErrorFlag);
FLEXIO_ClearTimerStatusFlags(FLEXIO0, 0xFF);
FLEXIO_CAMERA_TransferCreateHandleEDMA(&g_FlexioCameraDevice, &
                                       g_cameraEdmaHandle, FLEXIO_CAMERA_UserCallback, NULL,
                                       &g_edmaHandle);
cameraTransfer.dataAddress = (uint32_t)u16CameraFrameBuffer;
cameraTransfer.dataNum = sizeof(u16CameraFrameBuffer);
FLEXIO_CAMERA_TransferReceiveEDMA(&g_FlexioCameraDevice, &
                                   g_cameraEdmaHandle, &cameraTransfer);
while (!(isEDMAGetOnePictureFinish))
{
    ;
}
/* A callback function is also needed */
void FLEXIO_CAMERA_UserCallback(FLEXIO_CAMERA_Type *base,
                                flexio_camera_edma_handle_t *handle,
                                status_t status,
                                void *userData)
{
    userData = userData;
    /* eDMA Transfer finished */
    if (kStatus_FLEXIO_CAMERA_RxIdle == status)
    {
        isEDMAGetOnePictureFinish = true;
    }
}

```

Modules

- FlexIO eDMA Camera Driver

Data Structures

- struct **FLEXIO_CAMERA_Type**
Define structure of configuring the FlexIO Camera device. [More...](#)
- struct **flexio_camera_config_t**
Define FlexIO Camera user configuration structure. [More...](#)
- struct **flexio_camera_transfer_t**
Define FlexIO Camera transfer structure. [More...](#)

Macros

- #define **FLEXIO_CAMERA_PARALLEL_DATA_WIDTH** (8U)
Define the Camera CPI interface is constantly 8-bit width.

Enumerations

- enum **_flexio_camera_status** {
 kStatus_FLEXIO_CAMERA_RxBusy = MAKE_STATUS(kStatusGroup_FLEXIO_CAMERA,

FlexIO Camera Driver

- ```
0),
kStatus_FLEXIO_CAMERA_RxIdle = MAKE_STATUS(kStatusGroup_FLEXIO_CAMERA, 1)
}

Error codes for the Camera driver.
• enum _flexio_camera_status_flags {
 kFLEXIO_CAMERA_RxDataRegFullFlag = 0x1U,
 kFLEXIO_CAMERA_RxErrorFlag = 0x2U }

```
- Define FlexIO Camera status mask.*

## Driver version

- #define FSL\_FLEXIO\_CAMERA\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 0))  
*FlexIO Camera driver version 2.1.0.*

## Initialization and configuration

- void FLEXIO\_CAMERA\_Init (FLEXIO\_CAMERA\_Type \*base, const flexio\_camera\_config\_t \*config)  
*Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO Camera.*
- void FLEXIO\_CAMERA\_Deinit (FLEXIO\_CAMERA\_Type \*base)  
*Disables the FlexIO Camera and gates the FlexIO clock.*
- void FLEXIO\_CAMERA\_GetDefaultConfig (flexio\_camera\_config\_t \*config)  
*Gets the default configuration to configure the FlexIO Camera.*
- static void FLEXIO\_CAMERA\_Enable (FLEXIO\_CAMERA\_Type \*base, bool enable)  
*Enables/disables the FlexIO Camera module operation.*

## Status

- uint32\_t FLEXIO\_CAMERA\_GetStatusFlags (FLEXIO\_CAMERA\_Type \*base)  
*Gets the FlexIO Camera status flags.*
- void FLEXIO\_CAMERA\_ClearStatusFlags (FLEXIO\_CAMERA\_Type \*base, uint32\_t mask)  
*Clears the receive buffer full flag manually.*

## Interrupts

- void FLEXIO\_CAMERA\_EnableInterrupt (FLEXIO\_CAMERA\_Type \*base)  
*Switches on the interrupt for receive buffer full event.*
- void FLEXIO\_CAMERA\_DisableInterrupt (FLEXIO\_CAMERA\_Type \*base)  
*Switches off the interrupt for receive buffer full event.*

## DMA support

- static void FLEXIO\_CAMERA\_EnableRxDMA (FLEXIO\_CAMERA\_Type \*base, bool enable)

*Enables/disables the FlexIO Camera receive DMA.*

- static uint32\_t **FLEXIO\_CAMERA\_GetRxBufferAddress** (**FLEXIO\_CAMERA\_Type** \*base)  
*Gets the data from the receive buffer.*

### 12.3.3 Data Structure Documentation

#### 12.3.3.1 struct **FLEXIO\_CAMERA\_Type**

##### Data Fields

- **FLEXIO\_Type \* flexioBase**  
*FlexIO module base address.*
- **uint32\_t datPinStartIdx**  
*First data pin (D0) index for flexio\_camera.*
- **uint32\_t pclkPinIdx**  
*Pixel clock pin (PCLK) index for flexio\_camera.*
- **uint32\_t hrefPinIdx**  
*Horizontal sync pin (HREF) index for flexio\_camera.*
- **uint32\_t shifterStartIdx**  
*First shifter index used for flexio\_camera data FIFO.*
- **uint32\_t shifterCount**  
*The count of shifters that are used as flexio\_camera data FIFO.*
- **uint32\_t timerIdx**  
*Timer index used for flexio\_camera in FlexIO.*

##### 12.3.3.1.0.26 Field Documentation

###### 12.3.3.1.0.26.1 **FLEXIO\_Type\* FLEXIO\_CAMERA\_Type::flexioBase**

###### 12.3.3.1.0.26.2 **uint32\_t FLEXIO\_CAMERA\_Type::datPinStartIdx**

Then the successive following **FLEXIO\_CAMERA\_DATA\_WIDTH-1** pins are used as D1-D7.

###### 12.3.3.1.0.26.3 **uint32\_t FLEXIO\_CAMERA\_Type::pclkPinIdx**

###### 12.3.3.1.0.26.4 **uint32\_t FLEXIO\_CAMERA\_Type::hrefPinIdx**

###### 12.3.3.1.0.26.5 **uint32\_t FLEXIO\_CAMERA\_Type::shifterStartIdx**

###### 12.3.3.1.0.26.6 **uint32\_t FLEXIO\_CAMERA\_Type::shifterCount**

###### 12.3.3.1.0.26.7 **uint32\_t FLEXIO\_CAMERA\_Type::timerIdx**

#### 12.3.3.2 struct **flexio\_camera\_config\_t**

##### Data Fields

- bool **enablecamera**  
*Enable/disable FlexIO Camera TX & RX.*

## FlexIO Camera Driver

- bool `enableInDoze`  
*Enable/disable FlexIO operation in doze mode.*
- bool `enableInDebug`  
*Enable/disable FlexIO operation in debug mode.*
- bool `enableFastAccess`  
*Enable/disable fast access to FlexIO registers,  
fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.*

### 12.3.3.2.0.27 Field Documentation

12.3.3.2.0.27.1 `bool flexio_camera_config_t::enablecamera`

12.3.3.2.0.27.2 `bool flexio_camera_config_t::enableFastAccess`

### 12.3.3.3 `struct flexio_camera_transfer_t`

#### Data Fields

- `uint32_t dataAddress`  
*Transfer buffer.*
- `uint32_t dataNum`  
*Transfer num.*

### 12.3.4 Macro Definition Documentation

12.3.4.1 `#define FSL_FLEXIO_CAMERA_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`

12.3.4.2 `#define FLEXIO_CAMERA_PARALLEL_DATA_WIDTH (8U)`

### 12.3.5 Enumeration Type Documentation

#### 12.3.5.1 `enum _flexio_camera_status`

Enumerator

`kStatus_FLEXIO_CAMERA_RxBusy` Receiver is busy.

`kStatus_FLEXIO_CAMERA_RxIdle` Camera receiver is idle.

#### 12.3.5.2 `enum _flexio_camera_status_flags`

Enumerator

`kFLEXIO_CAMERA_RxDataRegFullFlag` Receive buffer full flag.

`kFLEXIO_CAMERA_RxErrorFlag` Receive buffer error flag.

## 12.3.6 Function Documentation

12.3.6.1 `void FLEXIO_CAMERA_Init( FLEXIO_CAMERA_Type * base, const flexio_camera_config_t * config )`

## FlexIO Camera Driver

Parameters

|               |                                                             |
|---------------|-------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_CAMERA_Type</a> structure     |
| <i>config</i> | Pointer to <a href="#">flexio_camera_config_t</a> structure |

### 12.3.6.2 void FLEXIO\_CAMERA\_Deinit ( [FLEXIO\\_CAMERA\\_Type](#) \* *base* )

Note

After calling this API, call [FLEXIO\\_CAMERA\\_Init](#) to use the FlexIO Camera module.

Parameters

|             |                                                         |
|-------------|---------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_CAMERA_Type</a> structure |
|-------------|---------------------------------------------------------|

### 12.3.6.3 void FLEXIO\_CAMERA\_GetDefaultConfig ( [flexio\\_camera\\_config\\_t](#) \* *config* )

The configuration can be used directly for calling the [FLEXIO\\_CAMERA\\_Init\(\)](#). Example:

```
flexio_camera_config_t config;
FLEXIO_CAMERA_GetDefaultConfig(&userConfig);
```

Parameters

|               |                                                                 |
|---------------|-----------------------------------------------------------------|
| <i>config</i> | Pointer to the <a href="#">flexio_camera_config_t</a> structure |
|---------------|-----------------------------------------------------------------|

### 12.3.6.4 static void FLEXIO\_CAMERA\_Enable ( [FLEXIO\\_CAMERA\\_Type](#) \* *base*, [bool](#) *enable* ) [inline], [static]

Parameters

|               |                                                   |
|---------------|---------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> |
| <i>enable</i> | True to enable, false to disable.                 |

### 12.3.6.5 [uint32\\_t](#) FLEXIO\_CAMERA\_GetStatusFlags ( [FLEXIO\\_CAMERA\\_Type](#) \* *base* )

Parameters

|             |                                                         |
|-------------|---------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_CAMERA_Type</a> structure |
|-------------|---------------------------------------------------------|

Returns

FlexIO shifter status flags

- `FLEXIO_SHIFTSTAT_SSF_MASK`
- 0

#### **12.3.6.6 void FLEXIO\_CAMERA\_ClearStatusFlags ( `FLEXIO_CAMERA_Type * base,` `uint32_t mask` )**

Parameters

|             |                                                                                                                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to the device.                                                                                                                                                                                                         |
| <i>mask</i> | status flag The parameter can be any combination of the following values: <ul style="list-style-type: none"> <li>• <code>kFLEXIO_CAMERA_RxDataRegFullFlag</code></li> <li>• <code>kFLEXIO_CAMERA_RxErrorFlag</code></li> </ul> |

#### **12.3.6.7 void FLEXIO\_CAMERA\_EnableInterrupt ( `FLEXIO_CAMERA_Type * base` )**

Parameters

|             |                        |
|-------------|------------------------|
| <i>base</i> | Pointer to the device. |
|-------------|------------------------|

#### **12.3.6.8 void FLEXIO\_CAMERA\_DisableInterrupt ( `FLEXIO_CAMERA_Type * base` )**

Parameters

|             |                        |
|-------------|------------------------|
| <i>base</i> | Pointer to the device. |
|-------------|------------------------|

#### **12.3.6.9 static void FLEXIO\_CAMERA\_EnableRxDMA ( `FLEXIO_CAMERA_Type * base,` `bool enable` ) [inline], [static]**

## FlexIO Camera Driver

Parameters

|               |                                                         |
|---------------|---------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_CAMERA_Type</a> structure |
| <i>enable</i> | True to enable, false to disable.                       |

The FlexIO Camera mode can't work without the DMA or eDMA support. Usually, it needs at least two DMA or eDMA channels, one for transferring data from Camera, such as 0V7670 to FlexIO buffer, another is for transferring data from FlexIO buffer to LCD.

### 12.3.6.10 static uint32\_t FLEXIO\_CAMERA\_GetRxBufferAddress ( FLEXIO\_CAMERA\_Type \* *base* ) [inline], [static]

Parameters

|             |                        |
|-------------|------------------------|
| <i>base</i> | Pointer to the device. |
|-------------|------------------------|

Returns

data Pointer to the buffer that keeps the data with count of *base*->shifterCount .

## 12.3.7 FlexIO eDMA Camera Driver

### 12.3.7.1 Overview

#### Data Structures

- struct `flexio_camera_edma_handle_t`  
*Camera eDMA handle. [More...](#)*

#### TypeDefs

- typedef void(\* `flexio_camera_edma_transfer_callback_t`)(`FLEXIO_CAMERA_Type` \*base, `flexio_camera_edma_handle_t` \*handle, `status_t` status, `void` \*userData)  
*Camera transfer callback function.*

#### eDMA transactional

- `status_t FLEXIO_CAMERA_TransferCreateHandleEDMA` (`FLEXIO_CAMERA_Type` \*base, `flexio_camera_edma_handle_t` \*handle, `flexio_camera_edma_transfer_callback_t` callback, `void` \*userData, `edma_handle_t` \*rxEdmaHandle)  
*Initializes the Camera handle, which is used in transactional functions.*
- `status_t FLEXIO_CAMERA_TransferReceiveEDMA` (`FLEXIO_CAMERA_Type` \*base, `flexio_camera_edma_handle_t` \*handle, `flexio_camera_transfer_t` \*xfer)  
*Receives data using eDMA.*
- `void FLEXIO_CAMERA_TransferAbortReceiveEDMA` (`FLEXIO_CAMERA_Type` \*base, `flexio_camera_edma_handle_t` \*handle)  
*Aborts the receive data which used the eDMA.*
- `status_t FLEXIO_CAMERA_TransferGetReceiveCountEDMA` (`FLEXIO_CAMERA_Type` \*base, `flexio_camera_edma_handle_t` \*handle, `size_t` \*count)  
*Gets the remaining bytes to be received.*

### 12.3.7.2 Data Structure Documentation

#### 12.3.7.2.1 struct \_flexio\_camera\_edma\_handle

Forward declaration of the handle typedef.

#### Data Fields

- `flexio_camera_edma_transfer_callback_t` `callback`  
*Callback function.*
- `void *` `userData`  
*Camera callback function parameter.*
- `size_t` `rxSize`  
*Total bytes to be received.*
- `edma_handle_t` \* `rxEdmaHandle`

## FlexIO Camera Driver

*The eDMA RX channel used.*

- `uint8_t nbytes`  
*eDMA minor byte transfer count initially configured.*
- `volatile uint8_t rxState`  
*RX transfer state.*

### 12.3.7.2.1.1 Field Documentation

12.3.7.2.1.1.1 `flexio_camera_edma_transfer_callback_t flexio_camera_edma_handle_t::callback`

12.3.7.2.1.1.2 `void* flexio_camera_edma_handle_t::userData`

12.3.7.2.1.1.3 `size_t flexio_camera_edma_handle_t::rxSize`

12.3.7.2.1.1.4 `edma_handle_t* flexio_camera_edma_handle_t::rxEdmaHandle`

12.3.7.2.1.1.5 `uint8_t flexio_camera_edma_handle_t::nbytes`

### 12.3.7.3 Typedef Documentation

12.3.7.3.1 `typedef void(* flexio_camera_edma_transfer_callback_t)(FLEXIO_CAMERA_Type *base, flexio_camera_edma_handle_t *handle, status_t status, void *userData)`

### 12.3.7.4 Function Documentation

12.3.7.4.1 `status_t FLEXIO_CAMERA_TransferCreateHandleEDMA ( FLEXIO_CAMERA_Type * base, flexio_camera_edma_handle_t * handle, flexio_camera_edma_transfer_callback_t callback, void * userData, edma_handle_t * rxEdmaHandle )`

Parameters

|                           |                                                                |
|---------------------------|----------------------------------------------------------------|
| <code>base</code>         | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> .            |
| <code>handle</code>       | Pointer to <code>flexio_camera_edma_handle_t</code> structure. |
| <code>callback</code>     | The callback function.                                         |
| <code>userData</code>     | The parameter of the callback function.                        |
| <code>rxEdmaHandle</code> | User requested DMA handle for RX DMA transfer.                 |

Return values

|                              |                                 |
|------------------------------|---------------------------------|
| <code>kStatus_Success</code> | Successfully create the handle. |
|------------------------------|---------------------------------|

|                           |                                                        |
|---------------------------|--------------------------------------------------------|
| <i>kStatus_OutOfRange</i> | The FlexIO Camera eDMA type/handle table out of range. |
|---------------------------|--------------------------------------------------------|

#### 12.3.7.4.2 status\_t FLEXIO\_CAMERA\_TransferReceiveEDMA ( **FLEXIO\_CAMERA\_Type** \* *base*, **flexio\_camera\_edma\_handle\_t** \* *handle*, **flexio\_camera\_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>   | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> .                            |
| <i>handle</i> | Pointer to the <a href="#">flexio_camera_edma_handle_t</a> structure.          |
| <i>xfer</i>   | Camera eDMA transfer structure, see <a href="#">flexio_camera_transfer_t</a> . |

Return values

|                               |                              |
|-------------------------------|------------------------------|
| <i>kStatus_Success</i>        | if succeeded, others failed. |
| <i>kStatus_CAMERA_Rx-Busy</i> | Previous transfer on going.  |

#### 12.3.7.4.3 void FLEXIO\_CAMERA\_TransferAbortReceiveEDMA ( **FLEXIO\_CAMERA\_Type** \* *base*, **flexio\_camera\_edma\_handle\_t** \* *handle* )

This function aborts the receive data which used the eDMA.

Parameters

|               |                                                                       |
|---------------|-----------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> .                   |
| <i>handle</i> | Pointer to the <a href="#">flexio_camera_edma_handle_t</a> structure. |

#### 12.3.7.4.4 status\_t FLEXIO\_CAMERA\_TransferGetReceiveCountEDMA ( **FLEXIO\_CAMERA\_Type** \* *base*, **flexio\_camera\_edma\_handle\_t** \* *handle*, **size\_t** \* *count* )

This function gets the number of bytes still not received.

## FlexIO Camera Driver

Parameters

|               |                                                              |
|---------------|--------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> .          |
| <i>handle</i> | Pointer to the flexio_camera_edma_handle_t structure.        |
| <i>count</i>  | Number of bytes sent so far by the non-blocking transaction. |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | Succeed get the transfer count. |
| <i>kStatus_InvalidArgument</i> | The count parameter is invalid. |

## 12.4 FlexIO I2C Master Driver

### 12.4.1 Overview

The KSDK provides a peripheral driver for I2C master function using Flexible I/O module of Kinetis devices.

The FlexIO I2C master driver includes functional APIs and transactional APIs.

Functional APIs target low level APIs. Functional APIs can be used for the FlexIO I2C master initialization/configuration/operation for the optimization/customization purpose. Using the functional APIs requires the knowledge of the FlexIO I2C master peripheral and how to organize functional APIs to meet the application requirements. The FlexIO I2C master functional operation groups provide the functional APIs set.

Transactional APIs target 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 an asynchronous transfer. This means that the functions [FLEXIO\\_I2C\\_MasterTransferNonBlocking\(\)](#) set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus\_Success status.

### 12.4.2 Typical use case

#### 12.4.2.1 FlexIO I2C master transfer using an interrupt method

```
flexio_i2c_master_handle_t g_m_handle;
flexio_i2c_master_config_t masterConfig;
flexio_i2c_master_transfer_t masterXfer;
volatile bool completionFlag = false;
const uint8_t sendData[] = [.....];
FLEXIO_I2C_Type i2cDev;

void FLEXIO_I2C_MasterCallback(FLEXIO_I2C_Type *base, status_t status, void *userData)
{
 userData = userData;

 if (kStatus_Success == status)
 {
 completionFlag = true;
 }
}

void main(void)
{
 //...

 FLEXIO_I2C_MasterGetDefaultConfig(&masterConfig);

 FLEXIO_I2C_MasterInit(&i2cDev, &user_config);
 FLEXIO_I2C_MasterTransferCreateHandle(&i2cDev, &g_m_handle,
 FLEXIO_I2C_MasterCallback, NULL);

 // Prepares to send.
```

## FlexIO I2C Master Driver

```
masterXfer.slaveAddress = g_accel_address[0];
masterXfer.direction = kI2C_Read;
masterXfer.subaddress = &who_am_i_reg;
masterXfer.subaddressSize = 1;
masterXfer.data = &who_am_i_value;
masterXfer.dataSize = 1;
masterXfer.flags = kI2C_TransferDefaultFlag;

// Sends out.
FLEXIO_I2C_MasterTransferNonBlocking(&i2cDev, &g_m_handle, &
 masterXfer);

// Wait for sending is complete.
while (!completionFlag)
{
}

// ...
}
```

## Data Structures

- struct **FLEXIO\_I2C\_Type**  
*Define FlexIO I2C master access structure typedef.* [More...](#)
- struct **flexio\_i2c\_master\_config\_t**  
*Define FlexIO I2C master user configuration structure.* [More...](#)
- struct **flexio\_i2c\_master\_transfer\_t**  
*Define FlexIO I2C master transfer structure.* [More...](#)
- struct **flexio\_i2c\_master\_handle\_t**  
*Define FlexIO I2C master handle structure.* [More...](#)

## TypeDefs

- typedef void(\* **flexio\_i2c\_master\_transfer\_callback\_t** )(FLEXIO\_I2C\_Type \*base, flexio\_i2c\_master\_handle\_t \*handle, status\_t status, void \*userData)  
*FlexIO I2C master transfer callback typedef.*

## Enumerations

- enum **\_flexio\_i2c\_status** {  
 kStatus\_FLEXIO\_I2C\_Busy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2C, 0),  
 kStatus\_FLEXIO\_I2C\_Idle = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2C, 1),  
 kStatus\_FLEXIO\_I2C\_Nak = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2C, 2) }  
*FlexIO I2C transfer status.*
- enum **\_flexio\_i2c\_master\_interrupt** {  
 kFLEXIO\_I2C\_TxEmptyInterruptEnable = 0x1U,  
 kFLEXIO\_I2C\_RxFullInterruptEnable = 0x2U }  
*Define FlexIO I2C master interrupt mask.*
- enum **\_flexio\_i2c\_master\_status\_flags** {

```
kFLEXIO_I2C_TxEmptyFlag = 0x1U,
kFLEXIO_I2C_RxFullFlag = 0x2U,
kFLEXIO_I2C_ReceiveNakFlag = 0x4U }
```

*Define FlexIO I2C master status mask.*

- enum `flexio_i2c_direction_t` {
 `kFLEXIO_I2C_Write` = 0x0U,
 `kFLEXIO_I2C_Read` = 0x1U }

*Direction of master transfer.*

## Driver version

- #define `FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 2)`)
 *FlexIO I2C master driver version 2.1.2.*

## Initialization and deinitialization

- void `FLEXIO_I2C_MasterInit` (`FLEXIO_I2C_Type` \*base, `flexio_i2c_master_config_t` \*masterConfig, `uint32_t` srcClock\_Hz)
 *Ungates the FlexIO clock, resets the FlexIO module, and configures the FlexIO I2C hardware configuration.*
- void `FLEXIO_I2C_MasterDeinit` (`FLEXIO_I2C_Type` \*base)
 *De-initializes the FlexIO I2C master peripheral.*
- void `FLEXIO_I2C_MasterGetDefaultConfig` (`flexio_i2c_master_config_t` \*masterConfig)
 *Gets the default configuration to configure the FlexIO module.*
- static void `FLEXIO_I2C_MasterEnable` (`FLEXIO_I2C_Type` \*base, `bool` enable)
 *Enables/disables the FlexIO module operation.*

## Status

- `uint32_t FLEXIO_I2C_MasterGetStatusFlags` (`FLEXIO_I2C_Type` \*base)
 *Gets the FlexIO I2C master status flags.*
- void `FLEXIO_I2C_MasterClearStatusFlags` (`FLEXIO_I2C_Type` \*base, `uint32_t` mask)
 *Clears the FlexIO I2C master status flags.*

## Interrupts

- void `FLEXIO_I2C_MasterEnableInterrupts` (`FLEXIO_I2C_Type` \*base, `uint32_t` mask)
 *Enables the FlexIO i2c master interrupt requests.*
- void `FLEXIO_I2C_MasterDisableInterrupts` (`FLEXIO_I2C_Type` \*base, `uint32_t` mask)
 *Disables the FlexIO I2C master interrupt requests.*

## FlexIO I2C Master Driver

### Bus Operations

- void `FLEXIO_I2C_MasterSetBaudRate` (`FLEXIO_I2C_Type` \*base, `uint32_t` baudRate\_Bps, `uint32_t` srcClock\_Hz)  
*Sets the FlexIO I2C master transfer baudrate.*
- void `FLEXIO_I2C_MasterStart` (`FLEXIO_I2C_Type` \*base, `uint8_t` address, `flexio_i2c_direction_t` direction)  
*Sends START + 7-bit address to the bus.*
- void `FLEXIO_I2C_MasterStop` (`FLEXIO_I2C_Type` \*base)  
*Sends the stop signal on the bus.*
- void `FLEXIO_I2C_MasterRepeatedStart` (`FLEXIO_I2C_Type` \*base)  
*Sends the repeated start signal on the bus.*
- void `FLEXIO_I2C_MasterAbortStop` (`FLEXIO_I2C_Type` \*base)  
*Sends the stop signal when transfer is still on-going.*
- void `FLEXIO_I2C_MasterEnableAck` (`FLEXIO_I2C_Type` \*base, `bool` enable)  
*Configures the sent ACK/NAK for the following byte.*
- `status_t FLEXIO_I2C_MasterSetTransferCount` (`FLEXIO_I2C_Type` \*base, `uint8_t` count)  
*Sets the number of bytes to be transferred from a start signal to a stop signal.*
- static void `FLEXIO_I2C_MasterWriteByte` (`FLEXIO_I2C_Type` \*base, `uint32_t` data)  
*Writes one byte of data to the I2C bus.*
- static `uint8_t FLEXIO_I2C_MasterReadByte` (`FLEXIO_I2C_Type` \*base)  
*Reads one byte of data from the I2C bus.*
- `status_t FLEXIO_I2C_MasterWriteBlocking` (`FLEXIO_I2C_Type` \*base, `const uint8_t` \*txBuff, `uint8_t` txSize)  
*Sends a buffer of data in bytes.*
- void `FLEXIO_I2C_MasterReadBlocking` (`FLEXIO_I2C_Type` \*base, `uint8_t` \*rxBuff, `uint8_t` rxSize)  
*Receives a buffer of bytes.*
- `status_t FLEXIO_I2C_MasterTransferBlocking` (`FLEXIO_I2C_Type` \*base, `flexio_i2c_master_transfer_t` \*xfer)  
*Performs a master polling transfer on the I2C bus.*

### Transactional

- `status_t FLEXIO_I2C_MasterTransferCreateHandle` (`FLEXIO_I2C_Type` \*base, `flexio_i2c_master_handle_t` \*handle, `flexio_i2c_master_transfer_callback_t` callback, `void` \*userData)  
*Initializes the I2C handle which is used in transactional functions.*
- `status_t FLEXIO_I2C_MasterTransferNonBlocking` (`FLEXIO_I2C_Type` \*base, `flexio_i2c_master_handle_t` \*handle, `flexio_i2c_master_transfer_t` \*xfer)  
*Performs a master interrupt non-blocking transfer on the I2C bus.*
- `status_t FLEXIO_I2C_MasterTransferGetCount` (`FLEXIO_I2C_Type` \*base, `flexio_i2c_master_handle_t` \*handle, `size_t` \*count)  
*Gets the master transfer status during a interrupt non-blocking transfer.*
- void `FLEXIO_I2C_MasterTransferAbort` (`FLEXIO_I2C_Type` \*base, `flexio_i2c_master_handle_t` \*handle)  
*Aborts an interrupt non-blocking transfer early.*
- void `FLEXIO_I2C_MasterTransferHandleIRQ` (`void` \*i2cType, `void` \*i2cHandle)  
*Master interrupt handler.*

## 12.4.3 Data Structure Documentation

### 12.4.3.1 struct FLEXIO\_I2C\_Type

#### Data Fields

- `FLEXIO_Type * flexioBase`  
*FlexIO base pointer.*
- `uint8_t SDAPinIndex`  
*Pin select for I2C SDA.*
- `uint8_t SCLPinIndex`  
*Pin select for I2C SCL.*
- `uint8_t shifterIndex [2]`  
*Shifter index used in FlexIO I2C.*
- `uint8_t timerIndex [2]`  
*Timer index used in FlexIO I2C.*

#### 12.4.3.1.0.1 Field Documentation

##### 12.4.3.1.0.1.1 `FLEXIO_Type* FLEXIO_I2C_Type::flexioBase`

##### 12.4.3.1.0.1.2 `uint8_t FLEXIO_I2C_Type::SDAPinIndex`

##### 12.4.3.1.0.1.3 `uint8_t FLEXIO_I2C_Type::SCLPinIndex`

##### 12.4.3.1.0.1.4 `uint8_t FLEXIO_I2C_Type::shifterIndex[2]`

##### 12.4.3.1.0.1.5 `uint8_t FLEXIO_I2C_Type::timerIndex[2]`

### 12.4.3.2 struct flexio\_i2c\_master\_config\_t

#### Data Fields

- `bool enableMaster`  
*Enables the FlexIO I2C peripheral at initialization time.*
- `bool enableInDoze`  
*Enable/disable FlexIO operation in doze mode.*
- `bool enableInDebug`  
*Enable/disable FlexIO operation in debug mode.*
- `bool enableFastAccess`  
*Enable/disable fast access to FlexIO registers, fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.*
- `uint32_t baudRate_Bps`  
*Baud rate in Bps.*

## FlexIO I2C Master Driver

### 12.4.3.2.0.2 Field Documentation

- 12.4.3.2.0.2.1 `bool flexio_i2c_master_config_t::enableMaster`
- 12.4.3.2.0.2.2 `bool flexio_i2c_master_config_t::enableInDoze`
- 12.4.3.2.0.2.3 `bool flexio_i2c_master_config_t::enableInDebug`
- 12.4.3.2.0.2.4 `bool flexio_i2c_master_config_t::enableFastAccess`
- 12.4.3.2.0.2.5 `uint32_t flexio_i2c_master_config_t::baudRate_Bps`

### 12.4.3.3 struct flexio\_i2c\_master\_transfer\_t

#### Data Fields

- `uint32_t flags`  
*Transfer flag which controls the transfer, reserved for FlexIO I2C.*
- `uint8_t slaveAddress`  
*7-bit slave address.*
- `flexio_i2c_direction_t direction`  
*Transfer direction, read or write.*
- `uint32_t subaddress`  
*Sub address.*
- `uint8_t subaddressSize`  
*Size of command buffer.*
- `uint8_t volatile * data`  
*Transfer buffer.*
- `volatile size_t dataSize`  
*Transfer size.*

### 12.4.3.3.0.3 Field Documentation

- 12.4.3.3.0.3.1 `uint32_t flexio_i2c_master_transfer_t::flags`
- 12.4.3.3.0.3.2 `uint8_t flexio_i2c_master_transfer_t::slaveAddress`
- 12.4.3.3.0.3.3 `flexio_i2c_direction_t flexio_i2c_master_transfer_t::direction`
- 12.4.3.3.0.3.4 `uint32_t flexio_i2c_master_transfer_t::subaddress`

Transferred MSB first.

**12.4.3.3.0.3.5** `uint8_t flexio_i2c_master_transfer_t::subaddressSize`

**12.4.3.3.0.3.6** `uint8_t volatile* flexio_i2c_master_transfer_t::data`

**12.4.3.3.0.3.7** `volatile size_t flexio_i2c_master_transfer_t::dataSize`

#### 12.4.3.4 `struct _flexio_i2c_master_handle`

FlexIO I2C master handle typedef.

#### Data Fields

- `flexio_i2c_master_transfer_t transfer`  
*FlexIO I2C master transfer copy.*
- `size_t transferSize`  
*Total bytes to be transferred.*
- `uint8_t state`  
*Transfer state maintained during transfer.*
- `flexio_i2c_master_transfer_callback_t completionCallback`  
*Callback function called at transfer event.*
- `void *userData`  
*Callback parameter passed to callback function.*

#### 12.4.3.4.0.4 Field Documentation

**12.4.3.4.0.4.1** `flexio_i2c_master_transfer_t flexio_i2c_master_handle_t::transfer`

**12.4.3.4.0.4.2** `size_t flexio_i2c_master_handle_t::transferSize`

**12.4.3.4.0.4.3** `uint8_t flexio_i2c_master_handle_t::state`

**12.4.3.4.0.4.4** `flexio_i2c_master_transfer_callback_t flexio_i2c_master_handle_t::completionCallback`

Callback function called at transfer event.

## FlexIO I2C Master Driver

12.4.3.4.0.4.5 `void* flexio_i2c_master_handle_t::userData`

### 12.4.4 Macro Definition Documentation

12.4.4.1 `#define FSL_FLEXIO_I2C_MASTER_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))`

### 12.4.5 Typedef Documentation

12.4.5.1 `typedef void(* flexio_i2c_master_transfer_callback_t)(FLEXIO_I2C_Type *base, flexio_i2c_master_handle_t *handle, status_t status, void *userData)`

### 12.4.6 Enumeration Type Documentation

#### 12.4.6.1 enum \_flexio\_i2c\_status

Enumerator

*kStatus\_FLEXIO\_I2C\_Busy* I2C is busy doing transfer.

*kStatus\_FLEXIO\_I2C\_Idle* I2C is busy doing transfer.

*kStatus\_FLEXIO\_I2C\_Nak* NAK received during transfer.

#### 12.4.6.2 enum \_flexio\_i2c\_master\_interrupt

Enumerator

*kFLEXIO\_I2C\_TxEmptyInterruptEnable* Tx buffer empty interrupt enable.

*kFLEXIO\_I2C\_RxFullInterruptEnable* Rx buffer full interrupt enable.

#### 12.4.6.3 enum \_flexio\_i2c\_master\_status\_flags

Enumerator

*kFLEXIO\_I2C\_TxEmptyFlag* Tx shifter empty flag.

*kFLEXIO\_I2C\_RxFullFlag* Rx shifter full/Transfer complete flag.

*kFLEXIO\_I2C\_ReceiveNakFlag* Receive NAK flag.

#### 12.4.6.4 enum flexio\_i2c\_direction\_t

Enumerator

*kFLEXIO\_I2C\_Write* Master send to slave.

*kFLEXIO\_I2C\_Read* Master receive from slave.

## 12.4.7 Function Documentation

### 12.4.7.1 void FLEXIO\_I2C\_MasterInit ( **FLEXIO\_I2C\_Type** \* *base*, *flexio\_i2c\_master\_config\_t* \* *masterConfig*, **uint32\_t** *srcClock\_Hz* )

Example

```
FLEXIO_I2C_Type base = {
 .flexioBase = FLEXIO,
 .SDAPinIndex = 0,
 .SCLPinIndex = 1,
 .shifterIndex = {0,1},
 .timerIndex = {0,1}
};
flexio_i2c_master_config_t config = {
 .enableInDoze = false,
 .enableInDebug = true,
 .enableFastAccess = false,
 .baudRate_Bps = 100000
};
FLEXIO_I2C_MasterInit(base, &config, srcClock_Hz);
```

Parameters

|                     |                                                         |
|---------------------|---------------------------------------------------------|
| <i>base</i>         | Pointer to <b>FLEXIO_I2C_Type</b> structure.            |
| <i>masterConfig</i> | Pointer to <b>flexio_i2c_master_config_t</b> structure. |
| <i>srcClock_Hz</i>  | FlexIO source clock in Hz.                              |

### 12.4.7.2 void FLEXIO\_I2C\_MasterDeinit ( **FLEXIO\_I2C\_Type** \* *base* )

Calling this API gates the FlexIO clock and the FlexIO I2C master module can't work unless the **FLEXIO\_I2C\_MasterInit** is called.

Parameters

|             |                                              |
|-------------|----------------------------------------------|
| <i>base</i> | pointer to <b>FLEXIO_I2C_Type</b> structure. |
|-------------|----------------------------------------------|

### 12.4.7.3 void FLEXIO\_I2C\_MasterGetDefaultConfig ( **flexio\_i2c\_master\_config\_t** \* *masterConfig* )

The configuration can be used directly for calling the **FLEXIO\_I2C\_MasterInit()**.

Example:

```
flexio_i2c_master_config_t config;
FLEXIO_I2C_MasterGetDefaultConfig(&config);
```

## FlexIO I2C Master Driver

Parameters

|                     |                                                                  |
|---------------------|------------------------------------------------------------------|
| <i>masterConfig</i> | Pointer to <a href="#">flexio_i2c_master_config_t</a> structure. |
|---------------------|------------------------------------------------------------------|

**12.4.7.4 static void FLEXIO\_I2C\_MasterEnable ( FLEXIO\_I2C\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

|               |                                                       |
|---------------|-------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
| <i>enable</i> | Pass true to enable module, false to disable module.  |

**12.4.7.5 uint32\_t FLEXIO\_I2C\_MasterGetStatusFlags ( FLEXIO\_I2C\_Type \* *base* )**

Parameters

|             |                                                      |
|-------------|------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure |
|-------------|------------------------------------------------------|

Returns

Status flag, use status flag to AND [\\_flexio\\_i2c\\_master\\_status\\_flags](#) can get the related status.

**12.4.7.6 void FLEXIO\_I2C\_MasterClearStatusFlags ( FLEXIO\_I2C\_Type \* *base*, uint32\_t *mask* )**

Parameters

|             |                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.                                                                                                                                    |
| <i>mask</i> | Status flag. The parameter can be any combination of the following values: <ul style="list-style-type: none"><li>• kFLEXIO_I2C_RxFullFlag</li><li>• kFLEXIO_I2C_ReceiveNakFlag</li></ul> |

**12.4.7.7 void FLEXIO\_I2C\_MasterEnableInterrupts ( FLEXIO\_I2C\_Type \* *base*, uint32\_t *mask* )**

Parameters

|             |                                                                                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.                                                                                                        |
| <i>mask</i> | Interrupt source. Currently only one interrupt request source: <ul style="list-style-type: none"><li>• kFLEXIO_I2C_TransferCompleteInterruptEnable</li></ul> |

#### 12.4.7.8 void FLEXIO\_I2C\_MasterDisableInterrupts ( [FLEXIO\\_I2C\\_Type](#) \* *base*, [uint32\\_t](#) *mask* )

Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
| <i>mask</i> | Interrupt source.                                     |

#### 12.4.7.9 void FLEXIO\_I2C\_MasterSetBaudRate ( [FLEXIO\\_I2C\\_Type](#) \* *base*, [uint32\\_t](#) *baudRate\_Bps*, [uint32\\_t](#) *srcClock\_Hz* )

Parameters

|                     |                                                      |
|---------------------|------------------------------------------------------|
| <i>base</i>         | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure |
| <i>baudRate_Bps</i> | the baud rate value in HZ                            |
| <i>srcClock_Hz</i>  | source clock in HZ                                   |

#### 12.4.7.10 void FLEXIO\_I2C\_MasterStart ( [FLEXIO\\_I2C\\_Type](#) \* *base*, [uint8\\_t](#) *address*, [flexio\\_i2c\\_direction\\_t](#) *direction* )

Note

This API should be called when the transfer configuration is ready to send a START signal and 7-bit address to the bus. This is a non-blocking API, which returns directly after the address is put into the data register but the address transfer is not finished on the bus. Ensure that the kFLEXIO\_I2C\_RxFullFlag status is asserted before calling this API.

## FlexIO I2C Master Driver

Parameters

|                  |                                                                                                                                                                                                                                              |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>      | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.                                                                                                                                                                                        |
| <i>address</i>   | 7-bit address.                                                                                                                                                                                                                               |
| <i>direction</i> | transfer direction. This parameter is one of the values in <code>flexio_i2c_direction_t</code> : <ul style="list-style-type: none"><li>• <code>kFLEXIO_I2C_Write</code>: Transmit</li><li>• <code>kFLEXIO_I2C_Read</code>: Receive</li></ul> |

### 12.4.7.11 void FLEXIO\_I2C\_MasterStop ( [FLEXIO\\_I2C\\_Type](#) \* *base* )

Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
|-------------|-------------------------------------------------------|

### 12.4.7.12 void FLEXIO\_I2C\_MasterRepeatedStart ( [FLEXIO\\_I2C\\_Type](#) \* *base* )

Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
|-------------|-------------------------------------------------------|

### 12.4.7.13 void FLEXIO\_I2C\_MasterAbortStop ( [FLEXIO\\_I2C\\_Type](#) \* *base* )

Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
|-------------|-------------------------------------------------------|

### 12.4.7.14 void FLEXIO\_I2C\_MasterEnableAck ( [FLEXIO\\_I2C\\_Type](#) \* *base*, `bool enable` )

Parameters

|               |                                                          |
|---------------|----------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.    |
| <i>enable</i> | True to configure send ACK, false configure to send NAK. |

### 12.4.7.15 `status_t` FLEXIO\_I2C\_MasterSetTransferCount ( [FLEXIO\\_I2C\\_Type](#) \* *base*, `uint8_t count` )

## Note

Call this API before a transfer begins because the timer generates a number of clocks according to the number of bytes that need to be transferred.

## Parameters

|              |                                                                                      |
|--------------|--------------------------------------------------------------------------------------|
| <i>base</i>  | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.                                |
| <i>count</i> | Number of bytes need to be transferred from a start signal to a re-start/stop signal |

## Return values

|                                |                                    |
|--------------------------------|------------------------------------|
| <i>kStatus_Success</i>         | Successfully configured the count. |
| <i>kStatus_InvalidArgument</i> | Input argument is invalid.         |

**12.4.7.16 static void FLEXIO\_I2C\_MasterWriteByte ( [FLEXIO\\_I2C\\_Type](#) \* *base*, [uint32\\_t](#) *data* ) [inline], [static]**

## Note

This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

## Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
| <i>data</i> | a byte of data.                                       |

**12.4.7.17 static [uint8\\_t](#) FLEXIO\_I2C\_MasterReadByte ( [FLEXIO\\_I2C\\_Type](#) \* *base* ) [inline], [static]**

## Note

This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the data is ready in the register.

## FlexIO I2C Master Driver

Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
|-------------|-------------------------------------------------------|

Returns

data byte read.

### 12.4.7.18 **status\_t FLEXIO\_I2C\_MasterWriteBlocking ( [FLEXIO\\_I2C\\_Type](#) \* *base*, const [uint8\\_t](#) \* *txBuff*, [uint8\\_t](#) *txSize* )**

Note

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

Parameters

|               |                                                       |
|---------------|-------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure. |
| <i>txBuff</i> | The data bytes to send.                               |
| <i>txSize</i> | The number of data bytes to send.                     |

Return values

|                               |                                  |
|-------------------------------|----------------------------------|
| <i>kStatus_Success</i>        | Successfully write data.         |
| <i>kStatus_FLEXIO_I2C_Nak</i> | Receive NAK during writing data. |

### 12.4.7.19 **void FLEXIO\_I2C\_MasterReadBlocking ( [FLEXIO\\_I2C\\_Type](#) \* *base*, [uint8\\_t](#) \* *rxBuff*, [uint8\\_t](#) *rxSize* )**

Note

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

Parameters

|               |                                              |
|---------------|----------------------------------------------|
| <i>base</i>   | Pointer to <b>FLEXIO_I2C_Type</b> structure. |
| <i>rxBuff</i> | The buffer to store the received bytes.      |
| <i>rxSize</i> | The number of data bytes to be received.     |

#### 12.4.7.20 status\_t FLEXIO\_I2C\_MasterTransferBlocking ( **FLEXIO\_I2C\_Type \* base,** **flexio\_i2c\_master\_transfer\_t \* xfer** )

Note

The API does not return until the transfer succeeds or fails due to receiving NAK.

Parameters

|             |                                                           |
|-------------|-----------------------------------------------------------|
| <i>base</i> | pointer to <b>FLEXIO_I2C_Type</b> structure.              |
| <i>xfer</i> | pointer to <b>flexio_i2c_master_transfer_t</b> structure. |

Returns

status of **status\_t**.

#### 12.4.7.21 status\_t FLEXIO\_I2C\_MasterTransferCreateHandle ( **FLEXIO\_I2C\_Type \* base,** **flexio\_i2c\_master\_handle\_t \* handle,** **flexio\_i2c\_master\_transfer\_callback\_t** **callback,** **void \* userData** )

Parameters

|                 |                                                                                     |
|-----------------|-------------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to <b>FLEXIO_I2C_Type</b> structure.                                        |
| <i>handle</i>   | Pointer to <b>flexio_i2c_master_handle_t</b> structure to store the transfer state. |
| <i>callback</i> | Pointer to user callback function.                                                  |
| <i>userData</i> | User param passed to the callback function.                                         |

Return values

|                        |                                 |
|------------------------|---------------------------------|
| <i>kStatus_Success</i> | Successfully create the handle. |
|------------------------|---------------------------------|

## FlexIO I2C Master Driver

|                           |                                                |
|---------------------------|------------------------------------------------|
| <i>kStatus_OutOfRange</i> | The FlexIO type/handle/isr table out of range. |
|---------------------------|------------------------------------------------|

### 12.4.7.22 status\_t FLEXIO\_I2C\_MasterTransferNonBlocking ( **FLEXIO\_I2C\_Type \* base**, **flexio\_i2c\_master\_handle\_t \* handle**, **flexio\_i2c\_master\_transfer\_t \* xfer** )

Note

The API returns immediately after the transfer initiates. Call FLEXIO\_I2C\_MasterGetTransferCount to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus\_FLEXIO\_I2C\_Busy, the transfer is finished.

Parameters

|               |                                                                                                 |
|---------------|-------------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure                                            |
| <i>handle</i> | Pointer to <a href="#">flexio_i2c_master_handle_t</a> structure which stores the transfer state |
| <i>xfer</i>   | pointer to <a href="#">flexio_i2c_master_transfer_t</a> structure                               |

Return values

|                                |                                                      |
|--------------------------------|------------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully start a transfer.                       |
| <i>kStatus_FLEXIO_I2C_Busy</i> | FlexIO I2C is not idle, is running another transfer. |

### 12.4.7.23 status\_t FLEXIO\_I2C\_MasterTransferGetCount ( **FLEXIO\_I2C\_Type \* base**, **flexio\_i2c\_master\_handle\_t \* handle**, **size\_t \* count** )

Parameters

|               |                                                                                                  |
|---------------|--------------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.                                            |
| <i>handle</i> | Pointer to <a href="#">flexio_i2c_master_handle_t</a> 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. |
|------------------------|--------------------------------|

**12.4.7.24 void FLEXIO\_I2C\_MasterTransferAbort ( **FLEXIO\_I2C\_Type** \* *base*, **flexio\_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>   | Pointer to <b>FLEXIO_I2C_Type</b> structure                                            |
| <i>handle</i> | Pointer to <b>flexio_i2c_master_handle_t</b> structure which stores the transfer state |

**12.4.7.25 void FLEXIO\_I2C\_MasterTransferHandleIRQ ( **void** \* *i2cType*, **void** \* *i2cHandle* )**

Parameters

|                  |                                                          |
|------------------|----------------------------------------------------------|
| <i>i2cType</i>   | Pointer to <b>FLEXIO_I2C_Type</b> structure              |
| <i>i2cHandle</i> | Pointer to <b>flexio_i2c_master_transfer_t</b> structure |

### 12.5 FlexIO I2S Driver

#### 12.5.1 Overview

The KSDK provides a peripheral driver for I2S function using Flexible I/O module of Kinetis devices.

The FlexIO I2S driver includes functional APIs and transactional APIs.

Functional APIs target low-level APIs. Functional APIs can be used for FlexIO I2S initialization/configuration/operation for optimization/customization purpose. Using the functional API requires knowledge of the FlexIO I2S peripheral and how to organize functional APIs to meet the application requirements. All functional APIs use the peripheral base address as the first parameter. FlexIO I2S functional operation groups provide the functional APIs set.

Transactional APIs target high-level APIs. The transactional APIs can be used to enable the peripheral and can also be used 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. All transactional APIs use the the `sai_handle_t` as the first parameter. Initialize the handle by calling the `FlexIO_I2S_TransferTxCreateHandle()` or `FlexIO_I2S_TransferRxCreateHandle()` API.

Transactional APIs support asynchronous transfer. This means that the functions `FLEXIO_I2S_TransferSendNonBlocking()` and `FLEXIO_I2S_TransferReceiveNonBlocking()` set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_FLEXIO_I2S_TxIdle` and `kStatus_FLEXIO_I2S_RxIdle` status.

#### 12.5.2 Typical use case

##### 12.5.2.1 FlexIO I2S 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 FLEXIO_I2S_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
 userData = userData;

 if (kStatus_FLEXIO_I2S_TxIdle == status)
 {
 txFinished = true;
 }
}

void main(void)
{
 //...

 FLEXIO_I2S_TxGetDefaultConfig(&user_config);

 FLEXIO_I2S_TxInit(FLEXIO_I2S0, &user_config);
 FLEXIO_I2S_TransferTxCreateHandle(FLEXIO_I2S0, &g_saiHandle,
```

```

FLEXIO_I2S_UserCallback, NULL);

//Configures the SAI format.
FLEXIO_I2S_TransferTxSetTransferFormat(FLEXIO_I2S0, &g_saiHandle, mclkSource, mclk);

// Prepares to send.
sendXfer.data = sendData
sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
txFinished = false;

// Sends out.
FLEXIO_I2S_TransferSendNonBlocking(FLEXIO_I2S0, &g_saiHandle, &
sendXfer);

// Waiting to send is finished.
while (!txFinished)
{
}

// ...
}
}

```

### 12.5.2.2 FLEXIO\_I2S 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 FLEXIO_I2S_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
 userData = userData;

 if (kStatus_FLEXIO_I2S_TxIdle == status)
 {
 txFinished = true;
 }
}

void main(void)
{
 //...

 FLEXIO_I2S_TxGetDefaultConfig(&user_config);
 FLEXIO_I2S_TxInit(FLEXIO_I2S0, &user_config);

 // Sets up the DMA.
 DMAMUX_Init(DMAMUX0);
 DMAMUX_SetSource(DMAMUX0, FLEXIO_I2S_TX_DMA_CHANNEL, FLEXIO_I2S_TX_DMA_REQUEST);
 DMAMUX_EnableChannel(DMAMUX0, FLEXIO_I2S_TX_DMA_CHANNEL);

 DMA_Init(DMA0);

 /* Creates the DMA handle. */
 DMA_TransferTxCreateHandle(&g_saiTxDmaHandle, DMA0, FLEXIO_I2S_TX_DMA_CHANNEL);

 FLEXIO_I2S_TransferTxCreateHandleDMA(FLEXIO_I2S0, &g_saiTxDmaHandle
 , FLEXIO_I2S_UserCallback, NULL);

 // Prepares to send.
 sendXfer.data = sendData
 sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
}

```

## FlexIO I2S Driver

```
txFinished = false;

// Sends out.
FLEXIO_I2S_TransferSendDMA(&g_saiHandle, &sendXfer);

// Waiting to send is finished.
while (!txFinished)
{
}

// ...
}
```

## Modules

- FlexIO DMA I2S Driver
- FlexIO eDMA I2S Driver

## Data Structures

- struct **FLEXIO\_I2S\_Type**  
*Define FlexIO I2S access structure typedef.* [More...](#)
- struct **flexio\_i2s\_config\_t**  
*FlexIO I2S configure structure.* [More...](#)
- struct **flexio\_i2s\_format\_t**  
*FlexIO I2S audio format, FlexIO I2S only support the same format in Tx and Rx.* [More...](#)
- struct **flexio\_i2s\_transfer\_t**  
*Define FlexIO I2S transfer structure.* [More...](#)
- struct **flexio\_i2s\_handle\_t**  
*Define FlexIO I2S handle structure.* [More...](#)

## Macros

- #define **FLEXIO\_I2S\_XFER\_QUEUE\_SIZE** (4)  
*FlexIO I2S transfer queue size, user can refine it according to use case.*

## Typedefs

- typedef void(\* **flexio\_i2s\_callback\_t** )(FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, status\_t status, void \*userData)  
*FlexIO I2S xfer callback prototype.*

## Enumerations

- enum `_flexio_i2s_status` {
   
kStatus\_FLEXIO\_I2S\_Idle = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 0),
   
kStatus\_FLEXIO\_I2S\_TxBusy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 1),
   
kStatus\_FLEXIO\_I2S\_RxBusy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 2),
   
kStatus\_FLEXIO\_I2S\_Error = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 3),
   
kStatus\_FLEXIO\_I2S\_QueueFull = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 4) }

*FlexIO I2S transfer status.*

- enum `flexio_i2s_master_slave_t` {
   
kFLEXIO\_I2S\_Master = 0x0U,
   
kFLEXIO\_I2S\_Slave = 0x1U }

*Master or slave mode.*

- enum `_flexio_i2s_interrupt_enable` {
   
kFLEXIO\_I2S\_TxDataRegEmptyInterruptEnable = 0x1U,
   
kFLEXIO\_I2S\_RxDataRegFullInterruptEnable = 0x2U }

*Define FlexIO FlexIO I2S interrupt mask.*

- enum `_flexio_i2s_status_flags` {
   
kFLEXIO\_I2S\_TxDataRegEmptyFlag = 0x1U,
   
kFLEXIO\_I2S\_RxDataRegFullFlag = 0x2U }

*Define FlexIO FlexIO I2S status mask.*

- enum `flexio_i2s_sample_rate_t` {
   
kFLEXIO\_I2S\_SampleRate8KHz = 8000U,
   
kFLEXIO\_I2S\_SampleRate11025Hz = 11025U,
   
kFLEXIO\_I2S\_SampleRate12KHz = 12000U,
   
kFLEXIO\_I2S\_SampleRate16KHz = 16000U,
   
kFLEXIO\_I2S\_SampleRate22050Hz = 22050U,
   
kFLEXIO\_I2S\_SampleRate24KHz = 24000U,
   
kFLEXIO\_I2S\_SampleRate32KHz = 32000U,
   
kFLEXIO\_I2S\_SampleRate44100Hz = 44100U,
   
kFLEXIO\_I2S\_SampleRate48KHz = 48000U,
   
kFLEXIO\_I2S\_SampleRate96KHz = 96000U }

*Audio sample rate.*

- enum `flexio_i2s_word_width_t` {
   
kFLEXIO\_I2S\_WordWidth8bits = 8U,
   
kFLEXIO\_I2S\_WordWidth16bits = 16U,
   
kFLEXIO\_I2S\_WordWidth24bits = 24U,
   
kFLEXIO\_I2S\_WordWidth32bits = 32U }

*Audio word width.*

## Driver version

- #define `FSL_FLEXIO_I2S_DRIVER_VERSION` (MAKE\_VERSION(2, 1, 1))
- FlexIO I2S driver version 2.1.0.*

## FlexIO I2S Driver

### Initialization and deinitialization

- void **FLEXIO\_I2S\_Init** (**FLEXIO\_I2S\_Type** \*base, const **flexio\_i2s\_config\_t** \*config)  
*Initializes the FlexIO I2S.*
- void **FLEXIO\_I2S\_GetDefaultConfig** (**flexio\_i2s\_config\_t** \*config)  
*Sets the FlexIO I2S configuration structure to default values.*
- void **FLEXIO\_I2S\_Deinit** (**FLEXIO\_I2S\_Type** \*base)  
*De-initializes the FlexIO I2S.*
- static void **FLEXIO\_I2S\_Enable** (**FLEXIO\_I2S\_Type** \*base, bool enable)  
*Enables/disables the FlexIO I2S module operation.*

### Status

- uint32\_t **FLEXIO\_I2S\_GetStatusFlags** (**FLEXIO\_I2S\_Type** \*base)  
*Gets the FlexIO I2S status flags.*

### Interrupts

- void **FLEXIO\_I2S\_EnableInterrupts** (**FLEXIO\_I2S\_Type** \*base, uint32\_t mask)  
*Enables the FlexIO I2S interrupt.*
- void **FLEXIO\_I2S\_DisableInterrupts** (**FLEXIO\_I2S\_Type** \*base, uint32\_t mask)  
*Disables the FlexIO I2S interrupt.*

### DMA Control

- static void **FLEXIO\_I2S\_TxEnableDMA** (**FLEXIO\_I2S\_Type** \*base, bool enable)  
*Enables/disables the FlexIO I2S Tx DMA requests.*
- static void **FLEXIO\_I2S\_RxEnableDMA** (**FLEXIO\_I2S\_Type** \*base, bool enable)  
*Enables/disables the FlexIO I2S Rx DMA requests.*
- static uint32\_t **FLEXIO\_I2S\_TxGetDataRegisterAddress** (**FLEXIO\_I2S\_Type** \*base)  
*Gets the FlexIO I2S send data register address.*
- static uint32\_t **FLEXIO\_I2S\_RxGetDataRegisterAddress** (**FLEXIO\_I2S\_Type** \*base)  
*Gets the FlexIO I2S receive data register address.*

### Bus Operations

- void **FLEXIO\_I2S\_MasterSetFormat** (**FLEXIO\_I2S\_Type** \*base, **flexio\_i2s\_format\_t** \*format, uint32\_t srcClock\_Hz)  
*Configures the FlexIO I2S audio format in master mode.*
- void **FLEXIO\_I2S\_SlaveSetFormat** (**FLEXIO\_I2S\_Type** \*base, **flexio\_i2s\_format\_t** \*format)  
*Configures the FlexIO I2S audio format in slave mode.*
- void **FLEXIO\_I2S\_WriteBlocking** (**FLEXIO\_I2S\_Type** \*base, uint8\_t bitWidth, uint8\_t \*txData, size\_t size)  
*Sends data using a blocking method.*
- static void **FLEXIO\_I2S\_WriteData** (**FLEXIO\_I2S\_Type** \*base, uint8\_t bitWidth, uint32\_t data)

- Writes data into a data register.  
void **FLEXIO\_I2S\_ReadBlocking** (FLEXIO\_I2S\_Type \*base, uint8\_t bitWidth, uint8\_t \*rxData, size\_t size)
- Receives a piece of data using a blocking method.  
static uint32\_t **FLEXIO\_I2S\_ReadData** (FLEXIO\_I2S\_Type \*base)
- Reads a data from the data register.

## Transactional

- void **FLEXIO\_I2S\_TransferTxCreateHandle** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, flexio\_i2s\_callback\_t callback, void \*userData)  
*Initializes the FlexIO I2S handle.*
- void **FLEXIO\_I2S\_TransferSetFormat** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, flexio\_i2s\_format\_t \*format, uint32\_t srcClock\_Hz)  
*Configures the FlexIO I2S audio format.*
- void **FLEXIO\_I2S\_TransferRxCreateHandle** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, flexio\_i2s\_callback\_t callback, void \*userData)  
*Initializes the FlexIO I2S receive handle.*
- status\_t **FLEXIO\_I2S\_TransferSendNonBlocking** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, flexio\_i2s\_transfer\_t \*xfer)  
*Performs an interrupt non-blocking send transfer on FlexIO I2S.*
- status\_t **FLEXIO\_I2S\_TransferReceiveNonBlocking** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, flexio\_i2s\_transfer\_t \*xfer)  
*Performs an interrupt non-blocking receive transfer on FlexIO I2S.*
- void **FLEXIO\_I2S\_TransferAbortSend** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle)  
*Aborts the current send.*
- void **FLEXIO\_I2S\_TransferAbortReceive** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle)  
*Aborts the current receive.*
- status\_t **FLEXIO\_I2S\_TransferGetSendCount** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, size\_t \*count)  
*Gets the remaining bytes to be sent.*
- status\_t **FLEXIO\_I2S\_TransferGetReceiveCount** (FLEXIO\_I2S\_Type \*base, flexio\_i2s\_handle\_t \*handle, size\_t \*count)  
*Gets the remaining bytes to be received.*
- void **FLEXIO\_I2S\_TransferTxHandleIRQ** (void \*i2sBase, void \*i2sHandle)  
*Tx interrupt handler.*
- void **FLEXIO\_I2S\_TransferRxHandleIRQ** (void \*i2sBase, void \*i2sHandle)  
*Rx interrupt handler.*

## 12.5.3 Data Structure Documentation

### 12.5.3.1 struct FLEXIO\_I2S\_Type

#### Data Fields

- FLEXIO\_Type \* **flexioBase**  
*FlexIO base pointer.*

## FlexIO I2S Driver

- `uint8_t txPinIndex`  
*Tx data pin index in FlexIO pins.*
- `uint8_t rxPinIndex`  
*Rx data pin index.*
- `uint8_t bclkPinIndex`  
*Bit clock pin index.*
- `uint8_t fsPinIndex`  
*Frame sync pin index.*
- `uint8_t txShifterIndex`  
*Tx data shifter index.*
- `uint8_t rxShifterIndex`  
*Rx data shifter index.*
- `uint8_t bclkTimerIndex`  
*Bit clock timer index.*
- `uint8_t fsTimerIndex`  
*Frame sync timer index.*

### 12.5.3.2 struct flexio\_i2s\_config\_t

#### Data Fields

- `bool enableI2S`  
*Enable FlexIO I2S.*
- `flexio_i2s_master_slave_t masterSlave`  
*Master or slave.*

### 12.5.3.3 struct flexio\_i2s\_format\_t

#### Data Fields

- `uint8_t bitWidth`  
*Bit width of audio data, always 8/16/24/32 bits.*
- `uint32_t sampleRate_Hz`  
*Sample rate of the audio data.*

### 12.5.3.4 struct flexio\_i2s\_transfer\_t

#### Data Fields

- `uint8_t * data`  
*Data buffer start pointer.*
- `size_t dataSize`  
*Bytes to be transferred.*

#### 12.5.3.4.0.5 Field Documentation

##### 12.5.3.4.0.5.1 size\_t flexio\_i2s\_transfer\_t::dataSize

##### 12.5.3.5 struct \_flexio\_i2s\_handle

#### Data Fields

- `uint32_t state`  
*Internal state.*
- `flexio_i2s_callback_t callback`  
*Callback function called at transfer event.*
- `void *userData`  
*Callback parameter passed to callback function.*
- `uint8_t bitWidth`  
*Bit width for transfer, 8/16/24/32bits.*
- `flexio_i2s_transfer_t queue [FLEXIO_I2S_XFER_QUEUE_SIZE]`  
*Transfer queue storing queued transfer.*
- `size_t transferSize [FLEXIO_I2S_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.*

#### 12.5.4 Macro Definition Documentation

##### 12.5.4.1 #define FSL\_FLEXIO\_I2S\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 1))

##### 12.5.4.2 #define FLEXIO\_I2S\_XFER\_QUEUE\_SIZE (4)

#### 12.5.5 Enumeration Type Documentation

##### 12.5.5.1 enum \_flexio\_i2s\_status

Enumerator

- `kStatus_FLEXIO_I2S_Idle` FlexIO I2S is in idle state.
- `kStatus_FLEXIO_I2S_TxBusy` FlexIO I2S Tx is busy.
- `kStatus_FLEXIO_I2S_RxBusy` FlexIO I2S Rx is busy.
- `kStatus_FLEXIO_I2S_Error` FlexIO I2S error occurred.
- `kStatus_FLEXIO_I2S_QueueFull` FlexIO I2S transfer queue is full.

## FlexIO I2S Driver

### 12.5.5.2 enum flexio\_i2s\_master\_slave\_t

Enumerator

*kFLEXIO\_I2S\_Master* Master mode.

*kFLEXIO\_I2S\_Slave* Slave mode.

### 12.5.5.3 enum \_flexio\_i2s\_interrupt\_enable

Enumerator

*kFLEXIO\_I2S\_TxDataRegEmptyInterruptEnable* Transmit buffer empty interrupt enable.

*kFLEXIO\_I2S\_RxDataRegFullInterruptEnable* Receive buffer full interrupt enable.

### 12.5.5.4 enum \_flexio\_i2s\_status\_flags

Enumerator

*kFLEXIO\_I2S\_TxDataRegEmptyFlag* Transmit buffer empty flag.

*kFLEXIO\_I2S\_RxDataRegFullFlag* Receive buffer full flag.

### 12.5.5.5 enum flexio\_i2s\_sample\_rate\_t

Enumerator

*kFLEXIO\_I2S\_SampleRate8KHz* Sample rate 8000Hz.

*kFLEXIO\_I2S\_SampleRate11025Hz* Sample rate 11025Hz.

*kFLEXIO\_I2S\_SampleRate12KHz* Sample rate 12000Hz.

*kFLEXIO\_I2S\_SampleRate16KHz* Sample rate 16000Hz.

*kFLEXIO\_I2S\_SampleRate22050Hz* Sample rate 22050Hz.

*kFLEXIO\_I2S\_SampleRate24KHz* Sample rate 24000Hz.

*kFLEXIO\_I2S\_SampleRate32KHz* Sample rate 32000Hz.

*kFLEXIO\_I2S\_SampleRate44100Hz* Sample rate 44100Hz.

*kFLEXIO\_I2S\_SampleRate48KHz* Sample rate 48000Hz.

*kFLEXIO\_I2S\_SampleRate96KHz* Sample rate 96000Hz.

### 12.5.5.6 enum flexio\_i2s\_word\_width\_t

Enumerator

*kFLEXIO\_I2S\_WordWidth8bits* Audio data width 8 bits.

*kFLEXIO\_I2S\_WordWidth16bits* Audio data width 16 bits.

*kFLEXIO\_I2S\_WordWidth24bits* Audio data width 24 bits.

*kFLEXIO\_I2S\_WordWidth32bits* Audio data width 32 bits.

## 12.5.6 Function Documentation

### 12.5.6.1 void FLEXIO\_I2S\_Init ( FLEXIO\_I2S\_Type \* *base*, const flexio\_i2s\_config\_t \* *config* )

This API configures FlexIO pins and shifter to I2S and configures the FlexIO I2S with a configuration structure. The configuration structure can be filled by the user, or be set with default values by [FLEXIO\\_I2S\\_GetDefaultConfig\(\)](#).

#### Note

This API should be called at the beginning of the application to use the FlexIO I2S driver. Otherwise, any access to the FlexIO I2S module can cause hard fault because the clock is not enabled.

#### Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | FlexIO I2S base pointer         |
| <i>config</i> | FlexIO I2S configure structure. |

### 12.5.6.2 void FLEXIO\_I2S\_GetDefaultConfig ( flexio\_i2s\_config\_t \* *config* )

The purpose of this API is to get the configuration structure initialized for use in [FLEXIO\\_I2S\\_Init\(\)](#). Users may use the initialized structure unchanged in [FLEXIO\\_I2S\\_Init\(\)](#) or modify some fields of the structure before calling [FLEXIO\\_I2S\\_Init\(\)](#).

#### Parameters

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

### 12.5.6.3 void FLEXIO\_I2S\_Deinit ( FLEXIO\_I2S\_Type \* *base* )

Calling this API gates the FlexIO i2s clock. After calling this API, call the [FLEXIO\\_I2S\\_Init](#) to use the FlexIO I2S module.

#### Parameters

|             |                         |
|-------------|-------------------------|
| <i>base</i> | FlexIO I2S base pointer |
|-------------|-------------------------|

### 12.5.6.4 static void FLEXIO\_I2S\_Enable ( FLEXIO\_I2S\_Type \* *base*, bool *enable* ) [inline], [static]

## FlexIO I2S Driver

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a> |
| <i>enable</i> | True to enable, false to disable.          |

### **12.5.6.5 uint32\_t FLEXIO\_I2S\_GetStatusFlags ( [FLEXIO\\_I2S\\_Type](#) \* *base* )**

Parameters

|             |                                                      |
|-------------|------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure |
|-------------|------------------------------------------------------|

Returns

Status flag, which are ORed by the enumerators in the \_flexio\_i2s\_status\_flags.

### **12.5.6.6 void FLEXIO\_I2S\_EnableInterrupts ( [FLEXIO\\_I2S\\_Type](#) \* *base*, uint32\_t *mask* )**

This function enables the FlexIO UART interrupt.

Parameters

|             |                                                      |
|-------------|------------------------------------------------------|
| <i>base</i> | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure |
| <i>mask</i> | interrupt source                                     |

### **12.5.6.7 void FLEXIO\_I2S\_DisableInterrupts ( [FLEXIO\\_I2S\\_Type](#) \* *base*, uint32\_t *mask* )**

This function disables the FlexIO UART interrupt.

Parameters

|             |                                                      |
|-------------|------------------------------------------------------|
| <i>base</i> | pointer to <a href="#">FLEXIO_I2S_Type</a> structure |
| <i>mask</i> | interrupt source                                     |

### **12.5.6.8 static void FLEXIO\_I2S\_TxEnableDMA ( [FLEXIO\\_I2S\\_Type](#) \* *base*, bool *enable* ) [inline], [static]**

Parameters

|               |                                                 |
|---------------|-------------------------------------------------|
| <i>base</i>   | FlexIO I2S base pointer                         |
| <i>enable</i> | True means enable DMA, false means disable DMA. |

#### 12.5.6.9 static void FLEXIO\_I2S\_RxEnableDMA ( **FLEXIO\_I2S\_Type** \* *base*, **bool** *enable* ) [inline], [static]

Parameters

|               |                                                 |
|---------------|-------------------------------------------------|
| <i>base</i>   | FlexIO I2S base pointer                         |
| <i>enable</i> | True means enable DMA, false means disable DMA. |

#### 12.5.6.10 static uint32\_t FLEXIO\_I2S\_TxGetDataRegisterAddress ( **FLEXIO\_I2S\_Type** \* *base* ) [inline], [static]

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

|             |                                             |
|-------------|---------------------------------------------|
| <i>base</i> | Pointer to <b>FLEXIO_I2S_Type</b> structure |
|-------------|---------------------------------------------|

Returns

FlexIO i2s send data register address.

#### 12.5.6.11 static uint32\_t FLEXIO\_I2S\_RxGetDataRegisterAddress ( **FLEXIO\_I2S\_Type** \* *base* ) [inline], [static]

This function returns the I2S data register address, mainly used by DMA/eDMA.

Parameters

|             |                                             |
|-------------|---------------------------------------------|
| <i>base</i> | Pointer to <b>FLEXIO_I2S_Type</b> structure |
|-------------|---------------------------------------------|

Returns

FlexIO i2s receive data register address.

## FlexIO I2S Driver

**12.5.6.12 void FLEXIO\_I2S\_MasterSetFormat ( FLEXIO\_I2S\_Type \* *base*,  
flexio\_i2s\_format\_t \* *format*, uint32\_t *srcClock\_Hz* )**

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

|                    |                                                    |
|--------------------|----------------------------------------------------|
| <i>base</i>        | Pointer to <b>FLEXIO_I2S_Type</b> structure        |
| <i>format</i>      | Pointer to FlexIO I2S audio data format structure. |
| <i>srcClock_Hz</i> | I2S master clock source frequency in Hz.           |

#### 12.5.6.13 void **FLEXIO\_I2S\_SlaveSetFormat** ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_format\_t** \* *format* )

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

|               |                                                    |
|---------------|----------------------------------------------------|
| <i>base</i>   | Pointer to <b>FLEXIO_I2S_Type</b> structure        |
| <i>format</i> | Pointer to FlexIO I2S audio data format structure. |

#### 12.5.6.14 void **FLEXIO\_I2S\_WriteBlocking** ( **FLEXIO\_I2S\_Type** \* *base*, **uint8\_t** *bitWidth*, **uint8\_t** \* *txData*, **size\_t** *size* )

Note

This function blocks via polling until data is ready to be sent.

Parameters

|                 |                                                         |
|-----------------|---------------------------------------------------------|
| <i>base</i>     | FlexIO I2S base pointer.                                |
| <i>bitWidth</i> | How many bits in a audio word, usually 8/16/24/32 bits. |
| <i>txData</i>   | Pointer to the data to be written.                      |
| <i>size</i>     | Bytes to be written.                                    |

#### 12.5.6.15 static void **FLEXIO\_I2S\_WriteData** ( **FLEXIO\_I2S\_Type** \* *base*, **uint8\_t** *bitWidth*, **uint32\_t** *data* ) [inline], [static]

Parameters

## FlexIO I2S Driver

|                 |                                                         |
|-----------------|---------------------------------------------------------|
| <i>base</i>     | FlexIO I2S base pointer.                                |
| <i>bitWidth</i> | How many bits in a audio word, usually 8/16/24/32 bits. |
| <i>data</i>     | Data to be written.                                     |

**12.5.6.16 void FLEXIO\_I2S\_ReadBlocking ( FLEXIO\_I2S\_Type \* *base*, uint8\_t *bitWidth*, uint8\_t \* *rxData*, size\_t *size* )**

Note

This function blocks via polling until data is ready to be sent.

Parameters

|                 |                                                         |
|-----------------|---------------------------------------------------------|
| <i>base</i>     | FlexIO I2S base pointer                                 |
| <i>bitWidth</i> | How many bits in a audio word, usually 8/16/24/32 bits. |
| <i>rxData</i>   | Pointer to the data to be read.                         |
| <i>size</i>     | Bytes to be read.                                       |

**12.5.6.17 static uint32\_t FLEXIO\_I2S\_ReadData ( FLEXIO\_I2S\_Type \* *base* )  
[inline], [static]**

Parameters

|             |                         |
|-------------|-------------------------|
| <i>base</i> | FlexIO I2S base pointer |
|-------------|-------------------------|

Returns

Data read from data register.

**12.5.6.18 void FLEXIO\_I2S\_TransferTxCreateHandle ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_handle\_t \* *handle*, flexio\_i2s\_callback\_t *callback*, void \* *userData* )**

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

|                 |                                                                              |
|-----------------|------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to <b>FLEXIO_I2S_Type</b> structure                                  |
| <i>handle</i>   | Pointer to <b>flexio_i2s_handle_t</b> structure to store the transfer state. |
| <i>callback</i> | FlexIO I2S callback function, which is called while finished a block.        |
| <i>userData</i> | User parameter for the FlexIO I2S callback.                                  |

**12.5.6.19 void FLEXIO\_I2S\_TransferSetFormat ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_handle\_t** \* *handle*, **flexio\_i2s\_format\_t** \* *format*, **uint32\_t** *srcClock\_Hz* )**

Audio format can be changed at run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred.

Parameters

|                    |                                                                                              |
|--------------------|----------------------------------------------------------------------------------------------|
| <i>base</i>        | Pointer to <b>FLEXIO_I2S_Type</b> structure.                                                 |
| <i>handle</i>      | FlexIO I2S handle pointer.                                                                   |
| <i>format</i>      | Pointer to audio data format structure.                                                      |
| <i>srcClock_Hz</i> | FlexIO I2S bit clock source frequency in Hz. This parameter should be 0 while in slave mode. |

**12.5.6.20 void FLEXIO\_I2S\_TransferRxCreateHandle ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_handle\_t** \* *handle*, **flexio\_i2s\_callback\_t** *callback*, **void** \* *userData* )**

This function initializes the FlexIO I2S handle which can be used for other FlexIO I2S transactional APIs. Call this API once to get the initialized handle.

Parameters

|                 |                                                                              |
|-----------------|------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to <b>FLEXIO_I2S_Type</b> structure.                                 |
| <i>handle</i>   | Pointer to <b>flexio_i2s_handle_t</b> structure to store the transfer state. |
| <i>callback</i> | FlexIO I2S callback function, which is called while finished a block.        |
| <i>userData</i> | User parameter for the FlexIO I2S callback.                                  |

**12.5.6.21 **status\_t** FLEXIO\_I2S\_TransferSendNonBlocking ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_handle\_t** \* *handle*, **flexio\_i2s\_transfer\_t** \* *xfer* )**

## FlexIO I2S Driver

### Note

The API returns immediately after transfer initiates. Call FLEXIO\_I2S\_GetRemainingBytes to poll the transfer status and check whether the transfer is finished. If the return status is 0, the transfer is finished.

### Parameters

|               |                                                                                          |
|---------------|------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure.                                    |
| <i>handle</i> | Pointer to <a href="#">flexio_i2s_handle_t</a> structure which stores the transfer state |
| <i>xfer</i>   | Pointer to <a href="#">flexio_i2s_transfer_t</a> structure                               |

### Return values

|                                   |                                                                                    |
|-----------------------------------|------------------------------------------------------------------------------------|
| <i>kStatus_Success</i>            | Successfully start the data transmission.                                          |
| <i>kStatus_FLEXIO_I2S_Tx-Busy</i> | Previous transmission still not finished, data not all written to TX register yet. |
| <i>kStatus_InvalidArgument</i>    | The input parameter is invalid.                                                    |

### 12.5.6.22 **status\_t FLEXIO\_I2S\_TransferReceiveNonBlocking ( [FLEXIO\\_I2S\\_Type](#) \* *base*, [flexio\\_i2s\\_handle\\_t](#) \* *handle*, [flexio\\_i2s\\_transfer\\_t](#) \* *xfer* )**

### Note

The API returns immediately after transfer initiates. Call FLEXIO\_I2S\_GetRemainingBytes to poll the transfer status to check whether the transfer is finished. If the return status is 0, the transfer is finished.

### Parameters

|               |                                                                                          |
|---------------|------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure.                                    |
| <i>handle</i> | Pointer to <a href="#">flexio_i2s_handle_t</a> structure which stores the transfer state |
| <i>xfer</i>   | Pointer to <a href="#">flexio_i2s_transfer_t</a> structure                               |

### Return values

|                                  |                                      |
|----------------------------------|--------------------------------------|
| <i>kStatus_Success</i>           | Successfully start the data receive. |
| <i>kStatus_FLEXIO_I2S-RxBusy</i> | Previous receive still not finished. |
| <i>kStatus_InvalidArgument</i>   | The input parameter is invalid.      |

### 12.5.6.23 void FLEXIO\_I2S\_TransferAbortSend ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_handle\_t** \* *handle* )

Note

This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <b>FLEXIO_I2S_Type</b> structure.                                    |
| <i>handle</i> | Pointer to <b>flexio_i2s_handle_t</b> structure which stores the transfer state |

### 12.5.6.24 void FLEXIO\_I2S\_TransferAbortReceive ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_handle\_t** \* *handle* )

Note

This API can be called at any time when interrupt non-blocking transfer initiates to abort the transfer in a early time.

Parameters

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <b>FLEXIO_I2S_Type</b> structure.                                    |
| <i>handle</i> | Pointer to <b>flexio_i2s_handle_t</b> structure which stores the transfer state |

### 12.5.6.25 **status\_t** FLEXIO\_I2S\_TransferGetSendCount ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_handle\_t** \* *handle*, **size\_t** \* *count* )

Parameters

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <b>FLEXIO_I2S_Type</b> structure.                                    |
| <i>handle</i> | Pointer to <b>flexio_i2s_handle_t</b> structure which stores the transfer state |
| <i>count</i>  | Bytes 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. |

**12.5.6.26 status\_t FLEXIO\_I2S\_TransferGetReceiveCount ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

|               |                                                                                          |
|---------------|------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure.                                    |
| <i>handle</i> | Pointer to <a href="#">flexio_i2s_handle_t</a> structure which stores the transfer state |

Returns

count Bytes 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. |

#### 12.5.6.27 void [FLEXIO\\_I2S\\_TransferTxHandleIRQ](#) ( *void \* i2sBase, void \* i2sHandle* )

Parameters

|                  |                                                          |
|------------------|----------------------------------------------------------|
| <i>i2sBase</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure.    |
| <i>i2sHandle</i> | Pointer to <a href="#">flexio_i2s_handle_t</a> structure |

#### 12.5.6.28 void [FLEXIO\\_I2S\\_TransferRxHandleIRQ](#) ( *void \* i2sBase, void \* i2sHandle* )

Parameters

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| <i>i2sBase</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure.     |
| <i>i2sHandle</i> | Pointer to <a href="#">flexio_i2s_handle_t</a> structure. |

### 12.5.7 FlexIO eDMA I2S Driver

#### 12.5.7.1 Overview

#### Data Structures

- struct `flexio_i2s_edma_handle_t`

*FlexIO I2S DMA transfer handle, users should not touch the content of the handle.* [More...](#)

#### TypeDefs

- typedef void(\* `flexio_i2s_edma_callback_t`)(`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `status_t` status, `void` \*userData)

*FlexIO I2S eDMA transfer callback function for finish and error.*

#### eDMA Transactional

- void `FLEXIO_I2S_TransferTxCreateHandleEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_edma_callback_t` callback, `void` \*userData, `edma_handle_t` \*dmaHandle)

*Initializes the FlexIO I2S eDMA handle.*

- void `FLEXIO_I2S_TransferRxCreateHandleEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_edma_callback_t` callback, `void` \*userData, `edma_handle_t` \*dmaHandle)

*Initializes the FlexIO I2S Rx eDMA handle.*

- void `FLEXIO_I2S_TransferSetFormatEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_format_t` \*format, `uint32_t` srcClock\_Hz)

*Configures the FlexIO I2S Tx audio format.*

- `status_t FLEXIO_I2S_TransferSendEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_transfer_t` \*xfer)

*Performs a non-blocking FlexIO I2S transfer using DMA.*

- `status_t FLEXIO_I2S_TransferReceiveEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_transfer_t` \*xfer)

*Performs a non-blocking FlexIO I2S receive using eDMA.*

- void `FLEXIO_I2S_TransferAbortSendEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle)

*Aborts a FlexIO I2S transfer using eDMA.*

- void `FLEXIO_I2S_TransferAbortReceiveEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle)

*Aborts a FlexIO I2S receive using eDMA.*

- `status_t FLEXIO_I2S_TransferGetSendCountEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `size_t` \*count)

*Gets the remaining bytes to be sent.*

- `status_t FLEXIO_I2S_TransferGetReceiveCountEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `size_t` \*count)

*Get the remaining bytes to be received.*

## 12.5.7.2 Data Structure Documentation

### 12.5.7.2.1 struct \_flexio\_i2s\_edma\_handle

#### Data Fields

- `edma_handle_t * dmaHandle`  
*DMA handler for FlexIO I2S send.*
- `uint8_t bytesPerFrame`  
*Bytes in a frame.*
- `uint8_t nbytes`  
*eDMA minor byte transfer count initially configured.*
- `uint32_t state`  
*Internal state for FlexIO I2S eDMA transfer.*
- `flexio_i2s_edma_callback_t callback`  
*Callback for users while transfer finish or error occurred.*
- `void * userData`  
*User callback parameter.*
- `edma_tcd_t tcd [FLEXIO_I2S_XFER_QUEUE_SIZE+1U]`  
*TCD pool for eDMA transfer.*
- `flexio_i2s_transfer_t queue [FLEXIO_I2S_XFER_QUEUE_SIZE]`  
*Transfer queue storing queued transfer.*
- `size_t transferSize [FLEXIO_I2S_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.*

#### 12.5.7.2.1.1 Field Documentation

##### 12.5.7.2.1.1.1 `uint8_t flexio_i2s_edma_handle_t::nbytes`

##### 12.5.7.2.1.1.2 `edma_tcd_t flexio_i2s_edma_handle_t::tcd[FLEXIO_I2S_XFER_QUEUE_SIZE+1U]`

##### 12.5.7.2.1.1.3 `flexio_i2s_transfer_t flexio_i2s_edma_handle_t::queue[FLEXIO_I2S_XFER_QUEUE_SIZE]`

##### 12.5.7.2.1.1.4 `volatile uint8_t flexio_i2s_edma_handle_t::queueUser`

#### 12.5.7.3 Function Documentation

##### 12.5.7.3.1 `void FLEXIO_I2S_TransferTxCreateHandleEDMA ( FLEXIO_I2S_Type * base, flexio_i2s_edma_handle_t * handle, flexio_i2s_edma_callback_t callback, void * userData, edma_handle_t * dmaHandle )`

This function initializes the FlexIO I2S master DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

## FlexIO I2S Driver

Parameters

|                  |                                                                               |
|------------------|-------------------------------------------------------------------------------|
| <i>base</i>      | FlexIO I2S peripheral base address.                                           |
| <i>handle</i>    | FlexIO I2S eDMA handle pointer.                                               |
| <i>callback</i>  | FlexIO I2S eDMA callback function called while finished a block.              |
| <i>userData</i>  | User parameter for callback.                                                  |
| <i>dmaHandle</i> | eDMA handle for FlexIO I2S. This handle is a static value allocated by users. |

**12.5.7.3.2 void FLEXIO\_I2S\_TransferRxCreateHandleEDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_edma\_handle\_t \* *handle*, flexio\_i2s\_edma\_callback\_t *callback*, void \* *userData*, edma\_handle\_t \* *dmaHandle* )**

This function initializes the FlexIO I2S slave DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

|                  |                                                                               |
|------------------|-------------------------------------------------------------------------------|
| <i>base</i>      | FlexIO I2S peripheral base address.                                           |
| <i>handle</i>    | FlexIO I2S eDMA handle pointer.                                               |
| <i>callback</i>  | FlexIO I2S eDMA callback function called while finished a block.              |
| <i>userData</i>  | User parameter for callback.                                                  |
| <i>dmaHandle</i> | eDMA handle for FlexIO I2S. This handle is a static value allocated by users. |

**12.5.7.3.3 void FLEXIO\_I2S\_TransferSetFormatEDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_edma\_handle\_t \* *handle*, flexio\_i2s\_format\_t \* *format*, uint32\_t *srcClock\_Hz* )**

Audio format can be changed in run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to format.

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S eDMA handle pointer      |

|                    |                                                                              |
|--------------------|------------------------------------------------------------------------------|
| <i>format</i>      | Pointer to FlexIO I2S audio data format structure.                           |
| <i>srcClock_Hz</i> | FlexIO I2S clock source frequency in Hz, it should be 0 while in slave mode. |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | Audio format set successfully.  |
| <i>kStatus_InvalidArgument</i> | The input arguments is invalid. |

#### 12.5.7.3.4 status\_t FLEXIO\_I2S\_TransferSendEDMA ( ***base***, ***flexio\_i2s\_edma\_handle\_t \* handle***, ***flexio\_i2s\_transfer\_t \* xfer*** )

Note

This interface returned immediately after transfer initiates. Users should call FLEXIO\_I2S\_GetTransferStatus to poll the transfer status and check whether the FlexIO I2S transfer is finished.

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |
| <i>xfer</i>   | Pointer to DMA transfer structure.  |

Return values

|                                |                                            |
|--------------------------------|--------------------------------------------|
| <i>kStatus_Success</i>         | Start a FlexIO I2S eDMA send successfully. |
| <i>kStatus_InvalidArgument</i> | The input arguments is invalid.            |
| <i>kStatus_TxBusy</i>          | FlexIO I2S is busy sending data.           |

#### 12.5.7.3.5 status\_t FLEXIO\_I2S\_TransferReceiveEDMA ( ***base***, ***flexio\_i2s\_edma\_handle\_t \* handle***, ***flexio\_i2s\_transfer\_t \* xfer*** )

Note

This interface returned immediately after transfer initiates. Users should call FLEXIO\_I2S\_GetReceiveRemainingBytes to poll the transfer status and check whether the FlexIO I2S transfer is finished.

## FlexIO I2S Driver

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |
| <i>xfer</i>   | Pointer to DMA transfer structure.  |

Return values

|                                |                                               |
|--------------------------------|-----------------------------------------------|
| <i>kStatus_Success</i>         | Start a FlexIO I2S eDMA receive successfully. |
| <i>kStatus_InvalidArgument</i> | The input arguments is invalid.               |
| <i>kStatus_RxBusy</i>          | FlexIO I2S is busy receiving data.            |

**12.5.7.3.6 void FLEXIO\_I2S\_TransferAbortSendEDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_edma\_handle\_t \* *handle* )**

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |

**12.5.7.3.7 void FLEXIO\_I2S\_TransferAbortReceiveEDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_edma\_handle\_t \* *handle* )**

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |

**12.5.7.3.8 status\_t FLEXIO\_I2S\_TransferGetSendCountEDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_edma\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

|             |                                     |
|-------------|-------------------------------------|
| <i>base</i> | FlexIO I2S peripheral base address. |
|-------------|-------------------------------------|

|               |                                |
|---------------|--------------------------------|
| <i>handle</i> | FlexIO I2S DMA handle pointer. |
| <i>count</i>  | Bytes 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. |

#### 12.5.7.3.9 **status\_t FLEXIO\_I2S\_TransferGetReceiveCountEDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_edma\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |
| <i>count</i>  | Bytes 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. |

### 12.5.8 FlexIO DMA I2S Driver

#### 12.5.8.1 Overview

#### Data Structures

- struct `flexio_i2s_dma_handle_t`

*FlexIO I2S DMA transfer handle, users should not touch the content of the handle.* [More...](#)

#### TypeDefs

- typedef void(\* `flexio_i2s_dma_callback_t`)(`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `status_t` status, `void` \*userData)

*FlexIO I2S DMA transfer callback function for finish and error.*

#### DMA Transactional

- void `FLEXIO_I2S_TransferTxCreateHandleDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `flexio_i2s_dma_callback_t` callback, `void` \*userData, `dma_handle_t` \*dmaHandle)  
*Initializes the FlexIO I2S DMA handle.*
- void `FLEXIO_I2S_TransferRxCreateHandleDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `flexio_i2s_dma_callback_t` callback, `void` \*userData, `dma_handle_t` \*dmaHandle)  
*Initializes the FlexIO I2S Rx DMA handle.*
- void `FLEXIO_I2S_TransferSetFormatDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `flexio_i2s_format_t` \*format, `uint32_t` srcClock\_Hz)  
*Configures the FlexIO I2S Tx audio format.*
- `status_t FLEXIO_I2S_TransferSendDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `flexio_i2s_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO I2S transfer using DMA.*
- `status_t FLEXIO_I2S_TransferReceiveDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `flexio_i2s_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO I2S receive using DMA.*
- void `FLEXIO_I2S_TransferAbortSendDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle)  
*Aborts a FlexIO I2S transfer using DMA.*
- void `FLEXIO_I2S_TransferAbortReceiveDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle)  
*Aborts a FlexIO I2S receive using DMA.*
- `status_t FLEXIO_I2S_TransferGetSendCountDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `size_t` \*count)  
*Gets the remaining bytes to be sent.*
- `status_t FLEXIO_I2S_TransferGetReceiveCountDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_dma_handle_t` \*handle, `size_t` \*count)  
*Gets the remaining bytes to be received.*

## 12.5.8.2 Data Structure Documentation

### 12.5.8.2.1 struct \_flexio\_i2s\_dma\_handle

#### Data Fields

- **dma\_handle\_t \* dmaHandle**  
*DMA handler for FlexIO I2S send.*
- **uint8\_t bytesPerFrame**  
*Bytes in a frame.*
- **uint32\_t state**  
*Internal state for FlexIO I2S DMA transfer.*
- **flexio\_i2s\_dma\_callback\_t callback**  
*Callback for users while transfer finish or error occurred.*
- **void \* userData**  
*User callback parameter.*
- **flexio\_i2s\_transfer\_t queue [FLEXIO\_I2S\_XFER\_QUEUE\_SIZE]**  
*Transfer queue storing queued transfer.*
- **size\_t transferSize [FLEXIO\_I2S\_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.*

#### 12.5.8.2.1.1 Field Documentation

**12.5.8.2.1.1.1 flexio\_i2s\_transfer\_t flexio\_i2s\_dma\_handle\_t::queue[FLEXIO\_I2S\_XFER\_QUEUE\_SIZE]**

**12.5.8.2.1.1.2 volatile uint8\_t flexio\_i2s\_dma\_handle\_t::queueUser**

#### 12.5.8.3 Function Documentation

**12.5.8.3.1 void FLEXIO\_I2S\_TransferTxCreateHandleDMA ( FLEXIO\_I2S\_Type \* base,  
flexio\_i2s\_dma\_handle\_t \* handle, flexio\_i2s\_dma\_callback\_t callback, void \*  
userData, dma\_handle\_t \* dmaHandle )**

This function initializes the FlexIO I2S master DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

---

## FlexIO I2S Driver

|                  |                                                                              |
|------------------|------------------------------------------------------------------------------|
| <i>base</i>      | FlexIO I2S peripheral base address.                                          |
| <i>handle</i>    | FlexIO I2S DMA handle pointer.                                               |
| <i>callback</i>  | FlexIO I2S DMA callback function called while finished a block.              |
| <i>userData</i>  | User parameter for callback.                                                 |
| <i>dmaHandle</i> | DMA handle for FlexIO I2S. This handle is a static value allocated by users. |

```
12.5.8.3.2 void FLEXIO_I2S_TransferRxCreateHandleDMA (FLEXIO_I2S_Type * base,
 flexio_i2s_dma_handle_t * handle, flexio_i2s_dma_callback_t callback, void *
 userData, dma_handle_t * dmaHandle)
```

This function initializes the FlexIO I2S slave DMA handle which can be used for other FlexIO I2S master transactional APIs. Usually, for a specified FlexIO I2S instance, call this API once to get the initialized handle.

Parameters

|                  |                                                                              |
|------------------|------------------------------------------------------------------------------|
| <i>base</i>      | FlexIO I2S peripheral base address.                                          |
| <i>handle</i>    | FlexIO I2S DMA handle pointer.                                               |
| <i>callback</i>  | FlexIO I2S DMA callback function called while finished a block.              |
| <i>userData</i>  | User parameter for callback.                                                 |
| <i>dmaHandle</i> | DMA handle for FlexIO I2S. This handle is a static value allocated by users. |

```
12.5.8.3.3 void FLEXIO_I2S_TransferSetFormatDMA (FLEXIO_I2S_Type * base,
 flexio_i2s_dma_handle_t * handle, flexio_i2s_format_t * format, uint32_t srcClock_Hz
)
```

Audio format can be changed at run-time of FlexIO I2S. This function configures the sample rate and audio data format to be transferred. This function also sets the DMA parameter according to the format.

Parameters

|               |                                                    |
|---------------|----------------------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address.                |
| <i>handle</i> | FlexIO I2S DMA handle pointer                      |
| <i>format</i> | Pointer to FlexIO I2S audio data format structure. |

|                    |                                                                              |
|--------------------|------------------------------------------------------------------------------|
| <i>srcClock_Hz</i> | FlexIO I2S clock source frequency in Hz. It should be 0 while in slave mode. |
|--------------------|------------------------------------------------------------------------------|

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | Audio format set successfully.  |
| <i>kStatus_InvalidArgument</i> | The input arguments is invalid. |

#### 12.5.8.3.4 status\_t FLEXIO\_I2S\_TransferSendDMA ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_dma\_handle\_t** \* *handle*, **flexio\_i2s\_transfer\_t** \* *xfer* )

Note

This interface returns immediately after transfer initiates. Call FLEXIO\_I2S\_GetTransferStatus to poll the transfer status and check whether FLEXIO I2S transfer finished.

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |
| <i>xfer</i>   | Pointer to DMA transfer structure.  |

Return values

|                                |                                           |
|--------------------------------|-------------------------------------------|
| <i>kStatus_Success</i>         | Start a FlexIO I2S DMA send successfully. |
| <i>kStatus_InvalidArgument</i> | The input arguments is invalid.           |
| <i>kStatus_TxBusy</i>          | FlexIO I2S is busy sending data.          |

#### 12.5.8.3.5 status\_t FLEXIO\_I2S\_TransferReceiveDMA ( **FLEXIO\_I2S\_Type** \* *base*, **flexio\_i2s\_dma\_handle\_t** \* *handle*, **flexio\_i2s\_transfer\_t** \* *xfer* )

Note

This interface returns immediately after transfer initiates. Call FLEXIO\_I2S\_GetReceiveRemainingBytes to poll the transfer status to check whether the FlexIO I2S transfer is finished.

Parameters

## FlexIO I2S Driver

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |
| <i>xfer</i>   | Pointer to DMA transfer structure.  |

Return values

|                                |                                              |
|--------------------------------|----------------------------------------------|
| <i>kStatus_Success</i>         | Start a FlexIO I2S DMA receive successfully. |
| <i>kStatus_InvalidArgument</i> | The input arguments is invalid.              |
| <i>kStatus_RxBusy</i>          | FlexIO I2S is busy receiving data.           |

**12.5.8.3.6 void FLEXIO\_I2S\_TransferAbortSendDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_dma\_handle\_t \* *handle* )**

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |

**12.5.8.3.7 void FLEXIO\_I2S\_TransferAbortReceiveDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_dma\_handle\_t \* *handle* )**

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |

**12.5.8.3.8 status\_t FLEXIO\_I2S\_TransferGetSendCountDMA ( FLEXIO\_I2S\_Type \* *base*, flexio\_i2s\_dma\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |

|              |             |
|--------------|-------------|
| <i>count</i> | Bytes sent. |
|--------------|-------------|

Return values

|                                      |                                                                |
|--------------------------------------|----------------------------------------------------------------|
| <i>kStatus_Success</i>               | Succeed get the transfer count.                                |
| <i>kStatus_NoTransferIn-Progress</i> | There is not a non-blocking transaction currently in progress. |

#### 12.5.8.3.9 status\_t FLEXIO\_I2S\_TransferGetReceiveCountDMA ( **FLEXIO\_I2S\_Type \* base,** **flexio\_i2s\_dma\_handle\_t \* handle, size\_t \* count** )

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | FlexIO I2S peripheral base address. |
| <i>handle</i> | FlexIO I2S DMA handle pointer.      |
| <i>count</i>  | Bytes received.                     |

Return values

|                                      |                                                                |
|--------------------------------------|----------------------------------------------------------------|
| <i>kStatus_Success</i>               | Succeed get the transfer count.                                |
| <i>kStatus_NoTransferIn-Progress</i> | There is not a non-blocking transaction currently in progress. |

### 12.6 FlexIO SPI Driver

#### 12.6.1 Overview

The KSDK provides a peripheral driver for an SPI function using the Flexible I/O module of Kinetis devices.

FlexIO SPI driver includes functional APIs and transactional APIs.

Functional APIs target low-level APIs. Functional APIs can be used for FlexIO SPI initialization/configuration/operation for optimization/customization purpose. Using the functional API requires the knowledge of the FlexIO SPI peripheral and how to organize functional APIs to meet the application requirements. All functional API use the [FLEXIO\\_SPI\\_Type](#) \*base as the first parameter. FlexIO SPI functional operation groups provide the functional API set.

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and also in the application if the code size and performance of transactional APIs can satisfy requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the flexio\_spi\_master\_handle\_t/flexio\_spi\_slave\_handle\_t as the second parameter. Initialize the handle by calling the [FLEXIO\\_SPI\\_MasterTransferCreateHandle\(\)](#) or [FLEXIO\\_SPI\\_SlaveTransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer. This means that the functions [FLEXIO\\_SPI\\_MasterTransferNonBlocking\(\)](#)/[FLEXIO\\_SPI\\_SlaveTransferNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer is complete, the upper layer is notified through a callback function with the kStatus\_FLEXIO\_SPI\_Idle status. Note that the FlexIO SPI slave driver only supports discontinuous PCS access, which is a limitation. The FlexIO SPI slave driver can support continuous PCS, but the slave can't adapt discontinuous and continuous PCS automatically. Users can change the timer disable mode in [FLEXIO\\_SPI\\_SlaveInit](#) manually, from kFLEXIO\_TimerDisableOnTimerCompare to kFLEXIO\_TimerDisableNever to enable a discontinuous PCS access. Only CPHA = 0 is supported.

#### 12.6.2 Typical use case

##### 12.6.2.1 FlexIO SPI send/receive using an interrupt method

```
flexio_spi_master_handle_t g_spiHandle;
FLEXIO_SPI_Type spiDev;
volatile bool txFinished;
static uint8_t srcBuff[BUFFER_SIZE];
static uint8_t destBuff[BUFFER_SIZE];

void FLEXIO_SPI_MasterUserCallback(FLEXIO_SPI_Type *base, flexio_spi_master_handle_t *handle
 , status_t status, void *userData)
{
 userData = userData;

 if (kStatus_FLEXIO_SPI_Idle == status)
 {
 txFinished = true;
 }
}

void main(void)
```

```
{
//...
flexio_spi_transfer_t xfer = {0};
flexio_spi_master_config_t userConfig;

FLEXIO_SPI_MasterGetDefaultConfig(&userConfig);
userConfig.baudRate_Bps = 500000U;

spiDev.flexioBase = BOARD_FLEXIO_BASE;
spiDev.SDOPinIndex = FLEXIO_SPI_MOSI_PIN;
spiDev.SDIPinIndex = FLEXIO_SPI_MISO_PIN;
spiDev.SCKPinIndex = FLEXIO_SPI_SCK_PIN;
spiDev.CSnPinIndex = FLEXIO_SPI_CSn_PIN;
spiDev.shifterIndex[0] = 0U;
spiDev.shifterIndex[1] = 1U;
spiDev.timerIndex[0] = 0U;
spiDev.timerIndex[1] = 1U;

FLEXIO_SPI_MasterInit(&spiDev, &userConfig, FLEXIO_CLOCK_FREQUENCY);

xfer.txData = srcBuff;
xfer.rxData = destBuff;
xfer.dataSize = BUFFER_SIZE;
xfer.flags = kFLEXIO_SPI_8bitMsb;
FLEXIO_SPI_MasterTransferCreateHandle(&spiDev, &g_spiHandle,
 FLEXIO_SPI_MasterUserCallback, NULL);
FLEXIO_SPI_MasterTransferNonBlocking(&spiDev, &g_spiHandle, &xfer);

// Send finished.
while (!txFinished)
{
}

// ...
}
```

### 12.6.2.2 FlexIO\_SPI Send/Receive using a DMA method

```
dma_handle_t g_spiTxDmaHandle;
dma_handle_t g_spiRxDmaHandle;
flexio_spi_master_handle_t g_spiHandle;
FLEXIO_SPI_Type spiDev;
volatile bool txFinished;
static uint8_t srcBuff[BUFFER_SIZE];
static uint8_t destBuff[BUFFER_SIZE];
void FLEXIO_SPI_MasterUserCallback(FLEXIO_SPI_Type *base, flexio_spi_master_dma_handle_t *
 handle, status_t status, void *userData)
{
 userData = userData;

 if (kStatus_FLEXIO_SPI_Idle == status)
 {
 txFinished = true;
 }
}

void main(void)
{
 flexio_spi_transfer_t xfer = {0};
 flexio_spi_master_config_t userConfig;

 FLEXIO_SPI_MasterGetDefaultConfig(&userConfig);
 userConfig.baudRate_Bps = 500000U;

 spiDev.flexioBase = BOARD_FLEXIO_BASE;
```

## FlexIO SPI Driver

```
spiDev.SDOPinIndex = FLEXIO_SPI_MOSI_PIN;
spiDev.SDIPinIndex = FLEXIO_SPI_MISO_PIN;
spiDev.SCKPinIndex = FLEXIO_SPI_SCK_PIN;
spiDev.CSnPinIndex = FLEXIO_SPI_CSn_PIN;
spiDev.shifterIndex[0] = 0U;
spiDev.shifterIndex[1] = 1U;
spiDev.timerIndex[0] = 0U;
spiDev.timerIndex[1] = 1U;

/*Initializes the DMA for the example.*/
DMAMGR_Init();

dma_request_source_tx = (dma_request_source_t)(FLEXIO_DMA_REQUEST_BASE + spiDev.
 shifterIndex[0]);
dma_request_source_rx = (dma_request_source_t)(FLEXIO_DMA_REQUEST_BASE + spiDev.
 shifterIndex[1]);

/* Requests DMA channels for transmit and receive.*/
DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_tx, 0, &txHandle);
DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_rx, 1, &rxHandle);

FLEXIO_SPI_MasterInit(&spiDev, &userConfig, FLEXIO_CLOCK_FREQUENCY);

/* Initializes the buffer.*/
for (i = 0; i < BUFFER_SIZE; i++)
{
 srcBuff[i] = i;
}

/* Sends to the slave.*/
xfer.txData = srcBuff;
xfer.rxData = destBuff;
xfer.dataSize = BUFFER_SIZE;
xfer.flags = kFLEXIO_SPI_8bitMsb;
FLEXIO_SPI_MasterTransferCreateHandleDMA(&spiDev, &g_spiHandle,
 FLEXIO_SPI_MasterUserCallback, NULL, &g_spiTxDmaHandle, &g_spiRxDmaHandle);
FLEXIO_SPI_MasterTransferDMA(&spiDev, &g_spiHandle, &xfer);

// Send finished.
while (!txFinished)
{
}

// ...
}
```

## Modules

- [FlexIO DMA SPI Driver](#)
- [FlexIO eDMA SPI Driver](#)

## Data Structures

- struct [FLEXIO\\_SPI\\_Type](#)  
*Define FlexIO SPI access structure typedef.* [More...](#)
- struct [flexio\\_spi\\_master\\_config\\_t](#)  
*Define FlexIO SPI master configuration structure.* [More...](#)
- struct [flexio\\_spi\\_slave\\_config\\_t](#)  
*Define FlexIO SPI slave configuration structure.* [More...](#)

- struct **flexio\_spi\_transfer\_t**  
*Define FlexIO SPI transfer structure. [More...](#)*
- struct **flexio\_spi\_master\_handle\_t**  
*Define FlexIO SPI handle structure. [More...](#)*

## Macros

- #define **FLEXIO\_SPI\_DUMMYDATA** (0xFFFFU)  
*FlexIO SPI dummy transfer data, the data is sent while txData is NULL.*

## Typedefs

- typedef flexio\_spi\_master\_handle\_t **flexio\_spi\_slave\_handle\_t**  
*Slave handle is the same with master handle.*
- typedef void(\* **flexio\_spi\_master\_transfer\_callback\_t** )(FLEXIO\_SPI\_Type \*base, flexio\_spi\_master\_handle\_t \*handle, status\_t status, void \*userData)  
*FlexIO SPI master callback for finished transmit.*
- typedef void(\* **flexio\_spi\_slave\_transfer\_callback\_t** )(FLEXIO\_SPI\_Type \*base, **flexio\_spi\_slave\_handle\_t** \*handle, status\_t status, void \*userData)  
*FlexIO SPI slave callback for finished transmit.*

## Enumerations

- enum **\_flexio\_spi\_status** {
   
*kStatus\_FLEXIO\_SPI\_Busy* = MAKE\_STATUS(kStatusGroup\_FLEXIO\_SPI, 1),
   
*kStatus\_FLEXIO\_SPI\_Idle* = MAKE\_STATUS(kStatusGroup\_FLEXIO\_SPI, 2),
   
*kStatus\_FLEXIO\_SPI\_Error* = MAKE\_STATUS(kStatusGroup\_FLEXIO\_SPI, 3) }
   
*Error codes for the FlexIO SPI driver.*
- enum **flexio\_spi\_clock\_phase\_t** {
   
*kFLEXIO\_SPI\_ClockPhaseFirstEdge* = 0x0U,
   
*kFLEXIO\_SPI\_ClockPhaseSecondEdge* = 0x1U }
   
*FlexIO SPI clock phase configuration.*
- enum **flexio\_spi\_shift\_direction\_t** {
   
*kFLEXIO\_SPI\_MsbFirst* = 0,
   
*kFLEXIO\_SPI\_LsbFirst* = 1 }
   
*FlexIO SPI data shifter direction options.*
- enum **flexio\_spi\_data\_bitcount\_mode\_t** {
   
*kFLEXIO\_SPI\_8BitMode* = 0x08U,
   
*kFLEXIO\_SPI\_16BitMode* = 0x10U }
   
*FlexIO SPI data length mode options.*
- enum **\_flexio\_spi\_interrupt\_enable** {
   
*kFLEXIO\_SPI\_TxEmptyInterruptEnable* = 0x1U,
   
*kFLEXIO\_SPI\_RxFullInterruptEnable* = 0x2U }
   
*Define FlexIO SPI interrupt mask.*

## FlexIO SPI Driver

- enum \_flexio\_spi\_status\_flags {  
    kFLEXIO\_SPI\_TxBufferEmptyFlag = 0x1U,  
    kFLEXIO\_SPI\_RxBufferFullFlag = 0x2U }  
    *Define FlexIO SPI status mask.*
- enum \_flexio\_spi\_dma\_enable {  
    kFLEXIO\_SPI\_TxDmaEnable = 0x1U,  
    kFLEXIO\_SPI\_RxDmaEnable = 0x2U,  
    kFLEXIO\_SPI\_DmaAllEnable = 0x3U }  
    *Define FlexIO SPI DMA mask.*
- enum \_flexio\_spi\_transfer\_flags {  
    kFLEXIO\_SPI\_8bitMsb = 0x1U,  
    kFLEXIO\_SPI\_8bitLsb = 0x2U,  
    kFLEXIO\_SPI\_16bitMsb = 0x9U,  
    kFLEXIO\_SPI\_16bitLsb = 0xaU }  
    *Define FlexIO SPI transfer flags.*

## Driver version

- #define FSL\_FLEXIO\_SPI\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 0))  
    *FlexIO SPI driver version 2.1.0.*

## FlexIO SPI Configuration

- void FLEXIO\_SPI\_MasterInit (FLEXIO\_SPI\_Type \*base, flexio\_spi\_master\_config\_t \*masterConfig, uint32\_t srcClock\_Hz)  
    *Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI master hardware, and configures the FlexIO SPI with FlexIO SPI master configuration.*
- void FLEXIO\_SPI\_MasterDeinit (FLEXIO\_SPI\_Type \*base)  
    *Gates the FlexIO clock.*
- void FLEXIO\_SPI\_MasterGetDefaultConfig (flexio\_spi\_master\_config\_t \*masterConfig)  
    *Gets the default configuration to configure the FlexIO SPI master.*
- void FLEXIO\_SPI\_SlaveInit (FLEXIO\_SPI\_Type \*base, flexio\_spi\_slave\_config\_t \*slaveConfig)  
    *Ungates the FlexIO clock, resets the FlexIO module, configures the FlexIO SPI slave hardware configuration, and configures the FlexIO SPI with FlexIO SPI slave configuration.*
- void FLEXIO\_SPI\_SlaveDeinit (FLEXIO\_SPI\_Type \*base)  
    *Gates the FlexIO clock.*
- void FLEXIO\_SPI\_SlaveGetDefaultConfig (flexio\_spi\_slave\_config\_t \*slaveConfig)  
    *Gets the default configuration to configure the FlexIO SPI slave.*

## Status

- uint32\_t FLEXIO\_SPI\_GetStatusFlags (FLEXIO\_SPI\_Type \*base)  
    *Gets FlexIO SPI status flags.*
- void FLEXIO\_SPI\_ClearStatusFlags (FLEXIO\_SPI\_Type \*base, uint32\_t mask)  
    *Clears FlexIO SPI status flags.*

## Interrupts

- void **FLEXIO\_SPI\_EnableInterrupts** (**FLEXIO\_SPI\_Type** \*base, **uint32\_t** mask)  
*Enables the FlexIO SPI interrupt.*
- void **FLEXIO\_SPI\_DisableInterrupts** (**FLEXIO\_SPI\_Type** \*base, **uint32\_t** mask)  
*Disables the FlexIO SPI interrupt.*

## DMA Control

- void **FLEXIO\_SPI\_EnableDMA** (**FLEXIO\_SPI\_Type** \*base, **uint32\_t** mask, **bool** enable)  
*Enables/disables the FlexIO SPI transmit DMA.*
- static **uint32\_t FLEXIO\_SPI\_GetTxDataRegisterAddress** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_shift\_direction\_t** direction)  
*Gets the FlexIO SPI transmit data register address for MSB first transfer.*
- static **uint32\_t FLEXIO\_SPI\_GetRxDataRegisterAddress** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_shift\_direction\_t** direction)  
*Gets the FlexIO SPI receive data register address for the MSB first transfer.*

## Bus Operations

- static void **FLEXIO\_SPI\_Enable** (**FLEXIO\_SPI\_Type** \*base, **bool** enable)  
*Enables/disables the FlexIO SPI module operation.*
- void **FLEXIO\_SPI\_MasterSetBaudRate** (**FLEXIO\_SPI\_Type** \*base, **uint32\_t** baudRate\_Bps, **uint32\_t** srcClockHz)  
*Sets baud rate for the FlexIO SPI transfer, which is only used for the master.*
- static void **FLEXIO\_SPI\_WriteData** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_shift\_direction\_t** direction, **uint16\_t** data)  
*Writes one byte of data, which is sent using the MSB method.*
- static **uint16\_t FLEXIO\_SPI\_ReadData** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_shift\_direction\_t** direction)  
*Reads 8 bit/16 bit data.*
- void **FLEXIO\_SPI\_WriteBlocking** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_shift\_direction\_t** direction, const **uint8\_t** \*buffer, **size\_t** size)  
*Sends a buffer of data bytes.*
- void **FLEXIO\_SPI\_ReadBlocking** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_shift\_direction\_t** direction, **uint8\_t** \*buffer, **size\_t** size)  
*Receives a buffer of bytes.*
- void **FLEXIO\_SPI\_MasterTransferBlocking** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_transfer\_t** \*xfer)  
*Receives a buffer of bytes.*

## Transactional

- **status\_t FLEXIO\_SPI\_MasterTransferCreateHandle** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_master\_handle\_t** \*handle, **flexio\_spi\_master\_transfer\_callback\_t** callback, **void** \*userData)  
*Initializes the FlexIO SPI Master handle, which is used in transactional functions.*

## FlexIO SPI Driver

- status\_t **FLEXIO\_SPI\_MasterTransferNonBlocking** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_master\_handle\_t \*handle, flexio\_spi\_transfer\_t \*xfer)  
*Master transfer data using IRQ.*
- void **FLEXIO\_SPI\_MasterTransferAbort** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_master\_handle\_t \*handle)  
*Aborts the master data transfer, which used IRQ.*
- status\_t **FLEXIO\_SPI\_MasterTransferGetCount** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_master\_handle\_t \*handle, size\_t \*count)  
*Gets the data transfer status which used IRQ.*
- void **FLEXIO\_SPI\_MasterTransferHandleIRQ** (void \*spiType, void \*spiHandle)  
*FlexIO SPI master IRQ handler function.*
- status\_t **FLEXIO\_SPI\_SlaveTransferCreateHandle** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_slave\_handle\_t \*handle, flexio\_spi\_slave\_transfer\_callback\_t callback, void \*userData)  
*Initializes the FlexIO SPI Slave handle, which is used in transactional functions.*
- status\_t **FLEXIO\_SPI\_SlaveTransferNonBlocking** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_slave\_handle\_t \*handle, flexio\_spi\_transfer\_t \*xfer)  
*Slave transfer data using IRQ.*
- static void **FLEXIO\_SPI\_SlaveTransferAbort** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_slave\_handle\_t \*handle)  
*Aborts the slave data transfer which used IRQ, share same API with master.*
- static status\_t **FLEXIO\_SPI\_SlaveTransferGetCount** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_slave\_handle\_t \*handle, size\_t \*count)  
*Gets the data transfer status which used IRQ, share same API with master.*
- void **FLEXIO\_SPI\_SlaveTransferHandleIRQ** (void \*spiType, void \*spiHandle)  
*FlexIO SPI slave IRQ handler function.*

### 12.6.3 Data Structure Documentation

#### 12.6.3.1 struct FLEXIO\_SPI\_Type

##### Data Fields

- FLEXIO\_Type \* **flexioBase**  
*FlexIO base pointer.*
- uint8\_t **SDOPinIndex**  
*Pin select for data output.*
- uint8\_t **SDIPinIndex**  
*Pin select for data input.*
- uint8\_t **SCKPinIndex**  
*Pin select for clock.*
- uint8\_t **CSnPinIndex**  
*Pin select for enable.*
- uint8\_t **shifterIndex** [2]  
*Shifter index used in FlexIO SPI.*
- uint8\_t **timerIndex** [2]  
*Timer index used in FlexIO SPI.*

### 12.6.3.1.0.1 Field Documentation

12.6.3.1.0.1.1 `FLEXIO_Type* FLEXIO_SPI_Type::flexioBase`

12.6.3.1.0.1.2 `uint8_t FLEXIO_SPI_Type::SDOPinIndex`

12.6.3.1.0.1.3 `uint8_t FLEXIO_SPI_Type::SDIPinIndex`

12.6.3.1.0.1.4 `uint8_t FLEXIO_SPI_Type::SCKPinIndex`

12.6.3.1.0.1.5 `uint8_t FLEXIO_SPI_Type::CSnPinIndex`

12.6.3.1.0.1.6 `uint8_t FLEXIO_SPI_Type::shifterIndex[2]`

12.6.3.1.0.1.7 `uint8_t FLEXIO_SPI_Type::timerIndex[2]`

### 12.6.3.2 `struct flexio_spi_master_config_t`

#### Data Fields

- bool `enableMaster`  
*Enable/disable FlexIO SPI master after configuration.*
- bool `enableInDoze`  
*Enable/disable FlexIO operation in doze mode.*
- bool `enableInDebug`  
*Enable/disable FlexIO operation in debug mode.*
- bool `enableFastAccess`  
*Enable/disable fast access to FlexIO registers,  
fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.*
- `uint32_t baudRate_Bps`  
*Baud rate in Bps.*
- `flexio_spi_clock_phase_t phase`  
*Clock phase.*
- `flexio_spi_data_bitcount_mode_t dataMode`  
*8bit or 16bit mode.*

### 12.6.3.2.0.2 Field Documentation

12.6.3.2.0.2.1 `bool flexio_spi_master_config_t::enableMaster`

12.6.3.2.0.2.2 `bool flexio_spi_master_config_t::enableInDoze`

12.6.3.2.0.2.3 `bool flexio_spi_master_config_t::enableInDebug`

12.6.3.2.0.2.4 `bool flexio_spi_master_config_t::enableFastAccess`

12.6.3.2.0.2.5 `uint32_t flexio_spi_master_config_t::baudRate_Bps`

12.6.3.2.0.2.6 `flexio_spi_clock_phase_t flexio_spi_master_config_t::phase`

12.6.3.2.0.2.7 `flexio_spi_data_bitcount_mode_t flexio_spi_master_config_t::dataMode`

### 12.6.3.3 `struct flexio_spi_slave_config_t`

#### Data Fields

- `bool enableSlave`  
*Enable/disable FlexIO SPI slave after configuration.*
- `bool enableInDoze`  
*Enable/disable FlexIO operation in doze mode.*
- `bool enableInDebug`  
*Enable/disable FlexIO operation in debug mode.*
- `bool enableFastAccess`  
*Enable/disable fast access to FlexIO registers,  
fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.*
- `flexio_spi_clock_phase_t phase`  
*Clock phase.*
- `flexio_spi_data_bitcount_mode_t dataMode`  
*8bit or 16bit mode.*

### 12.6.3.3.0.3 Field Documentation

**12.6.3.3.0.3.1 bool flexio\_spi\_slave\_config\_t::enableSlave**

**12.6.3.3.0.3.2 bool flexio\_spi\_slave\_config\_t::enableInDoze**

**12.6.3.3.0.3.3 bool flexio\_spi\_slave\_config\_t::enableInDebug**

**12.6.3.3.0.3.4 bool flexio\_spi\_slave\_config\_t::enableFastAccess**

**12.6.3.3.0.3.5 flexio\_spi\_clock\_phase\_t flexio\_spi\_slave\_config\_t::phase**

**12.6.3.3.0.3.6 flexio\_spi\_data\_bitcount\_mode\_t flexio\_spi\_slave\_config\_t::dataMode**

### 12.6.3.4 struct flexio\_spi\_transfer\_t

#### Data Fields

- `uint8_t * txData`  
*Send buffer.*
- `uint8_t * rxData`  
*Receive buffer.*
- `size_t dataSize`  
*Transfer bytes.*
- `uint8_t flags`  
*FlexIO SPI control flag, MSB first or LSB first.*

### 12.6.3.4.0.4 Field Documentation

**12.6.3.4.0.4.1 uint8\_t\* flexio\_spi\_transfer\_t::txData**

**12.6.3.4.0.4.2 uint8\_t\* flexio\_spi\_transfer\_t::rxData**

**12.6.3.4.0.4.3 size\_t flexio\_spi\_transfer\_t::dataSize**

**12.6.3.4.0.4.4 uint8\_t flexio\_spi\_transfer\_t::flags**

### 12.6.3.5 struct \_flexio\_spi\_master\_handle

typedef for `flexio_spi_master_handle_t` in advance.

#### Data Fields

- `uint8_t * txData`  
*Transfer buffer.*
- `uint8_t * rxData`  
*Receive buffer.*
- `size_t transferSize`  
*Total bytes to be transferred.*
- `volatile size_t txRemainingBytes`

## FlexIO SPI Driver

- volatile size\_t **rxRemainingBytes**  
*Send data remaining in bytes.*
- volatile uint32\_t **state**  
*Receive data remaining in bytes.*
- uint8\_t **bytePerFrame**  
*FlexIO SPI internal state.*
- flexio\_spi\_shift\_direction\_t **direction**  
*SPI mode, 2bytes or 1byte in a frame.*
- flexio\_spi\_master\_transfer\_callback\_t **callback**  
*Shift direction.*
- void \* **userData**  
*FlexIO SPI callback.*
- Callback parameter.

### 12.6.3.5.0.5 Field Documentation

- 12.6.3.5.0.5.1 `uint8_t* flexio_spi_master_handle_t::txData`
- 12.6.3.5.0.5.2 `uint8_t* flexio_spi_master_handle_t::rxData`
- 12.6.3.5.0.5.3 `size_t flexio_spi_master_handle_t::transferSize`
- 12.6.3.5.0.5.4 `volatile size_t flexio_spi_master_handle_t::txRemainingBytes`
- 12.6.3.5.0.5.5 `volatile size_t flexio_spi_master_handle_t::rxRemainingBytes`
- 12.6.3.5.0.5.6 `volatile uint32_t flexio_spi_master_handle_t::state`
- 12.6.3.5.0.5.7 `flexio_spi_shift_direction_t flexio_spi_master_handle_t::direction`
- 12.6.3.5.0.5.8 `flexio_spi_master_transfer_callback_t flexio_spi_master_handle_t::callback`
- 12.6.3.5.0.5.9 `void* flexio_spi_master_handle_t::userData`

### 12.6.4 Macro Definition Documentation

- 12.6.4.1 `#define FSL_FLEXIO_SPI_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`
- 12.6.4.2 `#define FLEXIO_SPI_DUMMYDATA (0xFFFFU)`

### 12.6.5 Typedef Documentation

- 12.6.5.1 `typedef flexio_spi_master_handle_t flexio_spi_slave_handle_t`

### 12.6.6 Enumeration Type Documentation

#### 12.6.6.1 enum \_flexio\_spi\_status

Enumerator

- kStatus\_FLEXIO\_SPI\_Busy* FlexIO SPI is busy.
- kStatus\_FLEXIO\_SPI\_Idle* SPI is idle.
- kStatus\_FLEXIO\_SPI\_Error* FlexIO SPI error.

#### 12.6.6.2 enum flexio\_spi\_clock\_phase\_t

Enumerator

- kFLEXIO\_SPI\_ClockPhaseFirstEdge* First edge on SPSCK occurs at the middle of the first cycle of a data transfer.

***kFLEXIO\_SPI\_ClockPhaseSecondEdge*** First edge on SPSCK occurs at the start of the first cycle of a data transfer.

### 12.6.6.3 enum flexio\_spi\_shift\_direction\_t

Enumerator

***kFLEXIO\_SPI\_MsbFirst*** Data transfers start with most significant bit.

***kFLEXIO\_SPI\_LsbFirst*** Data transfers start with least significant bit.

### 12.6.6.4 enum flexio\_spi\_data\_bitcount\_mode\_t

Enumerator

***kFLEXIO\_SPI\_8BitMode*** 8-bit data transmission mode.

***kFLEXIO\_SPI\_16BitMode*** 16-bit data transmission mode.

### 12.6.6.5 enum \_flexio\_spi\_interrupt\_enable

Enumerator

***kFLEXIO\_SPI\_TxEmptyInterruptEnable*** Transmit buffer empty interrupt enable.

***kFLEXIO\_SPI\_RxFullInterruptEnable*** Receive buffer full interrupt enable.

### 12.6.6.6 enum \_flexio\_spi\_status\_flags

Enumerator

***kFLEXIO\_SPI\_TxBufferEmptyFlag*** Transmit buffer empty flag.

***kFLEXIO\_SPI\_RxBufferFullFlag*** Receive buffer full flag.

### 12.6.6.7 enum \_flexio\_spi\_dma\_enable

Enumerator

***kFLEXIO\_SPI\_TxDmaEnable*** Tx DMA request source.

***kFLEXIO\_SPI\_RxDmaEnable*** Rx DMA request source.

***kFLEXIO\_SPI\_DmaAllEnable*** All DMA request source.

### 12.6.6.8 enum \_flexio\_spi\_transfer\_flags

Enumerator

**kFLEXIO\_SPI\_8bitMsb** FlexIO SPI 8-bit MSB first.  
**kFLEXIO\_SPI\_8bitLsb** FlexIO SPI 8-bit LSB first.  
**kFLEXIO\_SPI\_16bitMsb** FlexIO SPI 16-bit MSB first.  
**kFLEXIO\_SPI\_16bitLsb** FlexIO SPI 16-bit LSB first.

### 12.6.7 Function Documentation

#### 12.6.7.1 void FLEXIO\_SPI\_MasterInit ( **FLEXIO\_SPI\_Type** \* *base*, **flexio\_spi\_master\_config\_t** \* *masterConfig*, **uint32\_t** *srcClock\_Hz* )

The configuration structure can be filled by the user, or be set with default values by the [FLEXIO\\_SPI\\_MasterGetDefaultConfig\(\)](#).

Note

FlexIO SPI master only support CPOL = 0, which means clock inactive low.

Example

```
FLEXIO_SPI_Type spiDev = {
 .flexioBase = FLEXIO,
 .SDOPinIndex = 0,
 .SDIPinIndex = 1,
 .SCKPinIndex = 2,
 .CSnPinIndex = 3,
 .shifterIndex = {0,1},
 .timerIndex = {0,1}
};
flexio_spi_master_config_t config = {
 .enableMaster = true,
 .enableInDoze = false,
 .enableInDebug = true,
 .enableFastAccess = false,
 .baudRate_Bps = 500000,
 .phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
 .direction = kFLEXIO_SPI_MsbFirst,
 .dataMode = kFLEXIO_SPI_8BitMode
};
FLEXIO_SPI_MasterInit(&spiDev, &config, srcClock_Hz);
```

Parameters

---

## FlexIO SPI Driver

|                     |                                                                      |
|---------------------|----------------------------------------------------------------------|
| <i>base</i>         | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.            |
| <i>masterConfig</i> | Pointer to the <a href="#">flexio_spi_master_config_t</a> structure. |
| <i>srcClock_Hz</i>  | FlexIO source clock in Hz.                                           |

### 12.6.7.2 void FLEXIO\_SPI\_MasterDeinit ( [FLEXIO\\_SPI\\_Type](#) \* *base* )

Parameters

|             |                                                  |
|-------------|--------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> . |
|-------------|--------------------------------------------------|

### 12.6.7.3 void FLEXIO\_SPI\_MasterGetDefaultConfig ( [flexio\\_spi\\_master\\_config\\_t](#) \* *masterConfig* )

The configuration can be used directly by calling the [FLEXIO\\_SPI\\_MasterConfigure\(\)](#). Example:

```
flexio_spi_master_config_t masterConfig;
FLEXIO_SPI_MasterGetDefaultConfig(&masterConfig);
```

Parameters

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

### 12.6.7.4 void FLEXIO\_SPI\_SlaveInit ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_slave\\_config\\_t](#) \* *slaveConfig* )

The configuration structure can be filled by the user, or be set with default values by the [FLEXIO\\_SPI\\_SlaveGetDefaultConfig\(\)](#).

Note

Only one timer is needed in the FlexIO SPI slave. As a result, the second timer index is ignored. FlexIO SPI slave only support CPOL = 0, which means clock inactive low. Example

```
FLEXIO_SPI_Type spiDev = {
 .flexioBase = FLEXIO,
 .SDOPinIndex = 0,
 .SDIPinIndex = 1,
 .SCKPinIndex = 2,
 .CSnPinIndex = 3,
 .shifterIndex = {0,1},
 .timerIndex = {0}
};
flexio_spi_slave_config_t config = {
 .enableSlave = true,
 .enableInDoze = false,
```

```

.enableInDebug = true,
.enableFastAccess = false,
.phase = kFLEXIO_SPI_ClockPhaseFirstEdge,
.direction = kFLEXIO_SPI_MsbFirst,
.dataMode = kFLEXIO_SPI_8BitMode
};

FLEXIO_SPI_SlaveInit(&spiDev, &config);

```

## Parameters

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

**12.6.7.5 void FLEXIO\_SPI\_SlaveDeinit ( [FLEXIO\\_SPI\\_Type](#) \* *base* )**

## Parameters

|             |                                                  |
|-------------|--------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> . |
|-------------|--------------------------------------------------|

**12.6.7.6 void FLEXIO\_SPI\_SlaveGetDefaultConfig ( [flexio\\_spi\\_slave\\_config\\_t](#) \* *slaveConfig* )**

The configuration can be used directly for calling the [FLEXIO\\_SPI\\_SlaveConfigure\(\)](#). Example:

```

flexio_spi_slave_config_t slaveConfig;
FLEXIO_SPI_SlaveGetDefaultConfig(&slaveConfig);

```

## Parameters

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

**12.6.7.7 uint32\_t FLEXIO\_SPI\_GetStatusFlags ( [FLEXIO\\_SPI\\_Type](#) \* *base* )**

## Parameters

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

## Returns

status flag; Use the status flag to AND the following flag mask and get the status.

- [kFLEXIO\\_SPI\\_TxEmptyFlag](#)
- [kFLEXIO\\_SPI\\_RxEmptyFlag](#)

```
12.6.7.8 void FLEXIO_SPI_ClearStatusFlags (FLEXIO_SPI_Type * base, uint32_t mask
)
```

Parameters

|             |                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                                                                                                                |
| <i>mask</i> | status flag The parameter can be any combination of the following values: <ul style="list-style-type: none"> <li>• kFLEXIO_SPI_TxEmptyFlag</li> <li>• kFLEXIO_SPI_RxEmptyFlag</li> </ul> |

#### **12.6.7.9 void FLEXIO\_SPI\_EnableInterrupts ( FLEXIO\_SPI\_Type \* *base*, uint32\_t *mask* )**

This function enables the FlexIO SPI interrupt.

Parameters

|             |                                                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                                                                                                                                           |
| <i>mask</i> | interrupt source. The parameter can be any combination of the following values: <ul style="list-style-type: none"> <li>• kFLEXIO_SPI_RxFullInterruptEnable</li> <li>• kFLEXIO_SPI_TxEmptyInterruptEnable</li> </ul> |

#### **12.6.7.10 void FLEXIO\_SPI\_DisableInterrupts ( FLEXIO\_SPI\_Type \* *base*, uint32\_t *mask* )**

This function disables the FlexIO SPI interrupt.

Parameters

|             |                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                                                                                                                                          |
| <i>mask</i> | interrupt source The parameter can be any combination of the following values: <ul style="list-style-type: none"> <li>• kFLEXIO_SPI_RxFullInterruptEnable</li> <li>• kFLEXIO_SPI_TxEmptyInterruptEnable</li> </ul> |

#### **12.6.7.11 void FLEXIO\_SPI\_EnableDMA ( FLEXIO\_SPI\_Type \* *base*, uint32\_t *mask*, bool *enable* )**

This function enables/disables the FlexIO SPI Tx DMA, which means that asserting the kFLEXIO\_SPI\_TxEmptyFlag does/doesn't trigger the DMA request.

## FlexIO SPI Driver

Parameters

|               |                                                           |
|---------------|-----------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>mask</i>   | SPI DMA source.                                           |
| <i>enable</i> | True means enable DMA, false means disable DMA.           |

### **12.6.7.12 static uint32\_t FLEXIO\_SPI\_GetTxDataRegisterAddress ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_shift\_direction\_t *direction* ) [inline], [static]**

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

Parameters

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| <i>base</i>      | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>direction</i> | Shift direction of MSB first or LSB first.                |

Returns

FlexIO SPI transmit data register address.

### **12.6.7.13 static uint32\_t FLEXIO\_SPI\_GetRxDataRegisterAddress ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_shift\_direction\_t *direction* ) [inline], [static]**

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

Parameters

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| <i>base</i>      | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>direction</i> | Shift direction of MSB first or LSB first.                |

Returns

FlexIO SPI receive data register address.

### **12.6.7.14 static void FLEXIO\_SPI\_Enable ( FLEXIO\_SPI\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

|               |                                                  |
|---------------|--------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> . |
| <i>enable</i> | True to enable, false to disable.                |

#### 12.6.7.15 void FLEXIO\_SPI\_MasterSetBaudRate ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [uint32\\_t](#) *baudRate\_Bps*, [uint32\\_t](#) *srcClockHz* )

Parameters

|                     |                                                           |
|---------------------|-----------------------------------------------------------|
| <i>base</i>         | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>baudRate_Bps</i> | Baud Rate needed in Hz.                                   |
| <i>srcClockHz</i>   | SPI source clock frequency in Hz.                         |

#### 12.6.7.16 static void FLEXIO\_SPI\_WriteData ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_shift\\_direction\\_t](#) *direction*, [uint16\\_t](#) *data* ) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is put into the data register but the data transfer is not finished on the bus. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| <i>base</i>      | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>direction</i> | Shift direction of MSB first or LSB first.                |
| <i>data</i>      | 8 bit/16 bit data.                                        |

#### 12.6.7.17 static [uint16\\_t](#) FLEXIO\_SPI\_ReadData ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_shift\\_direction\\_t](#) *direction* ) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

## FlexIO SPI Driver

Parameters

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| <i>base</i>      | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>direction</i> | Shift direction of MSB first or LSB first.                |

Returns

8 bit/16 bit data received.

**12.6.7.18 void FLEXIO\_SPI\_WriteBlocking ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_shift\_direction\_t *direction*, const uint8\_t \* *buffer*, size\_t *size* )**

Note

This function blocks using the polling method until all bytes have been sent.

Parameters

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| <i>base</i>      | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>direction</i> | Shift direction of MSB first or LSB first.                |
| <i>buffer</i>    | The data bytes to send.                                   |
| <i>size</i>      | The number of data bytes to send.                         |

**12.6.7.19 void FLEXIO\_SPI\_ReadBlocking ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_shift\_direction\_t *direction*, uint8\_t \* *buffer*, size\_t *size* )**

Note

This function blocks using the polling method until all bytes have been received.

Parameters

|                  |                                                           |
|------------------|-----------------------------------------------------------|
| <i>base</i>      | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>direction</i> | Shift direction of MSB first or LSB first.                |
| <i>buffer</i>    | The buffer to store the received bytes.                   |

|                  |                                            |
|------------------|--------------------------------------------|
| <i>size</i>      | The number of data bytes to be received.   |
| <i>direction</i> | Shift direction of MSB first or LSB first. |

### 12.6.7.20 void FLEXIO\_SPI\_MasterTransferBlocking ( **FLEXIO\_SPI\_Type** \* *base*, **flexio\_spi\_transfer\_t** \* *xfer* )

Note

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

Parameters

|             |                                                                   |
|-------------|-------------------------------------------------------------------|
| <i>base</i> | pointer to <b>FLEXIO_SPI_Type</b> structure                       |
| <i>xfer</i> | FlexIO SPI transfer structure, see <b>flexio_spi_transfer_t</b> . |

### 12.6.7.21 status\_t FLEXIO\_SPI\_MasterTransferCreateHandle ( **FLEXIO\_SPI\_Type** \* *base*, **flexio\_spi\_master\_handle\_t** \* *handle*, **flexio\_spi\_master\_transfer\_callback\_t** *callback*, **void** \* *userData* )

Parameters

|                 |                                                                                         |
|-----------------|-----------------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to the <b>FLEXIO_SPI_Type</b> structure.                                        |
| <i>handle</i>   | Pointer to the <b>flexio_spi_master_handle_t</b> structure to store the transfer state. |
| <i>callback</i> | The callback function.                                                                  |
| <i>userData</i> | The parameter of the callback function.                                                 |

Return values

|                           |                                                |
|---------------------------|------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                |
| <i>kStatus_OutOfRange</i> | The FlexIO type/handle/ISR table out of range. |

### 12.6.7.22 status\_t FLEXIO\_SPI\_MasterTransferNonBlocking ( **FLEXIO\_SPI\_Type** \* *base*, **flexio\_spi\_master\_handle\_t** \* *handle*, **flexio\_spi\_transfer\_t** \* *xfer* )

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

## FlexIO SPI Driver

Parameters

|               |                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                     |
| <i>handle</i> | Pointer to the <code>flexio_spi_master_handle_t</code> structure to store the transfer state. |
| <i>xfer</i>   | FlexIO SPI transfer structure. See <a href="#">flexio_spi_transfer_t</a> .                    |

Return values

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

**12.6.7.23 void FLEXIO\_SPI\_MasterTransferAbort ( `FLEXIO_SPI_Type * base,`  
`flexio_spi_master_handle_t * handle` )**

Parameters

|               |                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                     |
| <i>handle</i> | Pointer to the <code>flexio_spi_master_handle_t</code> structure to store the transfer state. |

**12.6.7.24 status\_t FLEXIO\_SPI\_MasterTransferGetCount ( `FLEXIO_SPI_Type * base,`  
`flexio_spi_master_handle_t * handle, size_t * count` )**

Parameters

|               |                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                     |
| <i>handle</i> | Pointer to the <code>flexio_spi_master_handle_t</code> structure to store 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. |

**12.6.7.25 void FLEXIO\_SPI\_MasterTransferHandleIRQ ( `void * spiType, void * spiHandle`  
)**

Parameters

|                  |                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------|
| <i>spiType</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                        |
| <i>spiHandle</i> | Pointer to the <a href="#">flexio_spi_master_handle_t</a> structure to store the transfer state. |

**12.6.7.26 status\_t FLEXIO\_SPI\_SlaveTransferCreateHandle ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_slave\\_handle\\_t](#) \* *handle*, [flexio\\_spi\\_slave\\_transfer\\_callback\\_t](#) *callback*, [void](#) \* *userData* )**

Parameters

|                 |                                                                                                 |
|-----------------|-------------------------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                       |
| <i>handle</i>   | Pointer to the <a href="#">flexio_spi_slave_handle_t</a> structure to store the transfer state. |
| <i>callback</i> | The callback function.                                                                          |
| <i>userData</i> | The parameter of the callback function.                                                         |

Return values

|                           |                                                |
|---------------------------|------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                |
| <i>kStatus_OutOfRange</i> | The FlexIO type/handle/ISR table out of range. |

**12.6.7.27 status\_t FLEXIO\_SPI\_SlaveTransferNonBlocking ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_slave\\_handle\\_t](#) \* *handle*, [flexio\\_spi\\_transfer\\_t](#) \* *xfer* )**

This function sends data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, the callback function is called.

Parameters

|               |                                                                                                 |
|---------------|-------------------------------------------------------------------------------------------------|
| <i>handle</i> | Pointer to the <a href="#">flexio_spi_slave_handle_t</a> structure to store the transfer state. |
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                       |
| <i>xfer</i>   | FlexIO SPI transfer structure. See <a href="#">flexio_spi_transfer_t</a> .                      |

Return values

|                                |                                                  |
|--------------------------------|--------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully start a transfer.                   |
| <i>kStatus_InvalidArgument</i> | Input argument is invalid.                       |
| <i>kStatus_FLEXIO_SPI_Busy</i> | SPI is not idle; it is running another transfer. |

**12.6.7.28 static void FLEXIO\_SPI\_SlaveTransferAbort ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_slave\_handle\_t \* *handle* ) [inline], [static]**

Parameters

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                       |
| <i>handle</i> | Pointer to the flexio_spi_slave_handle_t structure to store the transfer state. |

**12.6.7.29 static status\_t FLEXIO\_SPI\_SlaveTransferGetCount ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_slave\_handle\_t \* *handle*, size\_t \* *count* ) [inline], [static]**

Parameters

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                       |
| <i>handle</i> | Pointer to the flexio_spi_slave_handle_t structure to store 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. |

**12.6.7.30 void FLEXIO\_SPI\_SlaveTransferHandleIRQ ( void \* *spiType*, void \* *spiHandle* )**

Parameters

|                  |                                                                                 |
|------------------|---------------------------------------------------------------------------------|
| <i>spiType</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                       |
| <i>spiHandle</i> | Pointer to the flexio_spi_slave_handle_t structure to store the transfer state. |

### 12.6.8 FlexIO eDMA SPI Driver

#### 12.6.8.1 Overview

#### Data Structures

- struct `flexio_spi_master_edma_handle_t`

*FlexIO SPI eDMA transfer handle, users should not touch the content of the handle. [More...](#)*

#### TypeDefs

- typedef `flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t`  
*Slave handle is the same with master handle.*
- typedef void(\* `flexio_spi_master_edma_transfer_callback_t`)(`FLEXIO_SPI_Type` \*base, `flexio_spi_master_edma_handle_t` \*handle, `status_t` status, void \*userData)  
*FlexIO SPI master callback for finished transmit.*
- typedef void(\* `flexio_spi_slave_edma_transfer_callback_t`)(`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_edma_handle_t` \*handle, `status_t` status, void \*userData)  
*FlexIO SPI slave callback for finished transmit.*

#### eDMA Transactional

- `status_t FLEXIO_SPI_MasterTransferCreateHandleEDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_edma_handle_t` \*handle, `flexio_spi_master_edma_transfer_callback_t` callback, void \*userData, `edma_handle_t` \*txHandle, `edma_handle_t` \*rxHandle)  
*Initializes the FlexIO SPI master eDMA handle.*
- `status_t FLEXIO_SPI_MasterTransferEDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_edma_handle_t` \*handle, `flexio_spi_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO SPI transfer using eDMA.*
- `void FLEXIO_SPI_MasterTransferAbortEDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_edma_handle_t` \*handle)  
*Aborts a FlexIO SPI transfer using eDMA.*
- `status_t FLEXIO_SPI_MasterTransferGetCountEDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_edma_handle_t` \*handle, `size_t` \*count)  
*Gets the remaining bytes for FlexIO SPI eDMA transfer.*
- `static void FLEXIO_SPI_SlaveTransferCreateHandleEDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_edma_handle_t` \*handle, `flexio_spi_slave_edma_transfer_callback_t` callback, void \*userData, `edma_handle_t` \*txHandle, `edma_handle_t` \*rxHandle)  
*Initializes the FlexIO SPI slave eDMA handle.*
- `status_t FLEXIO_SPI_SlaveTransferEDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_edma_handle_t` \*handle, `flexio_spi_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO SPI transfer using eDMA.*
- `static void FLEXIO_SPI_SlaveTransferAbortEDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_edma_handle_t` \*handle)  
*Aborts a FlexIO SPI transfer using eDMA.*

- static status\_t **FLEXIO\_SPI\_SlaveTransferGetCountEDMA** (**FLEXIO\_SPI\_Type** \*base, **flexio\_spi\_slave\_edma\_handle\_t** \*handle, **size\_t** \*count)  
*Gets the remaining bytes to be transferred for FlexIO SPI eDMA.*

## 12.6.8.2 Data Structure Documentation

### 12.6.8.2.1 **struct \_flexio\_spi\_master\_edma\_handle**

typedef for **flexio\_spi\_master\_edma\_handle\_t** in advance.

#### Data Fields

- **size\_t transferSize**  
*Total bytes to be transferred.*
- **uint8\_t nbytes**  
*eDMA minor byte transfer count initially configured.*
- **bool txInProgress**  
*Send transfer in progress.*
- **bool rxInProgress**  
*Receive transfer in progress.*
- **edma\_handle\_t \* txHandle**  
*DMA handler for SPI send.*
- **edma\_handle\_t \* rxHandle**  
*DMA handler for SPI receive.*
- **flexio\_spi\_master\_edma\_transfer\_callback\_t callback**  
*Callback for SPI DMA transfer.*
- **void \* userData**  
*User Data for SPI DMA callback.*

## FlexIO SPI Driver

### 12.6.8.2.1.1 Field Documentation

12.6.8.2.1.1.1 `size_t flexio_spi_master_edma_handle_t::transferSize`

12.6.8.2.1.1.2 `uint8_t flexio_spi_master_edma_handle_t::nbytes`

### 12.6.8.3 Typedef Documentation

12.6.8.3.1 `typedef flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t`

### 12.6.8.4 Function Documentation

12.6.8.4.1 `status_t FLEXIO_SPI_MasterTransferCreateHandleEDMA ( FLEXIO_SPI_Type * base, flexio_spi_master_edma_handle_t * handle, flexio_spi_master_edma_transfer_callback_t callback, void * userData, edma_handle_t * txHandle, edma_handle_t * rxHandle )`

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

Parameters

|                 |                                                                                                |
|-----------------|------------------------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                          |
| <i>handle</i>   | Pointer to <code>flexio_spi_master_edma_handle_t</code> structure to store the transfer state. |
| <i>callback</i> | SPI callback, NULL means no callback.                                                          |
| <i>userData</i> | callback function parameter.                                                                   |
| <i>txHandle</i> | User requested eDMA handle for FlexIO SPI RX eDMA transfer.                                    |
| <i>rxHandle</i> | User requested eDMA handle for FlexIO SPI TX eDMA transfer.                                    |

Return values

|                           |                                                     |
|---------------------------|-----------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                     |
| <i>kStatus_OutOfRange</i> | The FlexIO SPI eDMA type/handle table out of range. |

12.6.8.4.2 `status_t FLEXIO_SPI_MasterTransferEDMA ( FLEXIO_SPI_Type * base, flexio_spi_master_edma_handle_t * handle, flexio_spi_transfer_t * xfer )`

Note

This interface returns immediately after transfer initiates. Call `FLEXIO_SPI_MasterGetTransferCountEDMA` to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

|               |                                                                                                   |
|---------------|---------------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                             |
| <i>handle</i> | Pointer to <a href="#">flexio_spi_master_edma_handle_t</a> structure to store the transfer state. |
| <i>xfer</i>   | Pointer to FlexIO SPI transfer structure.                                                         |

Return values

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

**12.6.8.4.3 void FLEXIO\_SPI\_MasterTransferAbortEDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_master\\_edma\\_handle\\_t](#) \* *handle* )**

Parameters

|               |                                                       |
|---------------|-------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>handle</i> | FlexIO SPI eDMA handle pointer.                       |

**12.6.8.4.4 status\_t FLEXIO\_SPI\_MasterTransferGetCountEDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_master\\_edma\\_handle\\_t](#) \* *handle*, [size\\_t](#) \* *count* )**

Parameters

|               |                                                                     |
|---------------|---------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.               |
| <i>handle</i> | FlexIO SPI eDMA handle pointer.                                     |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction. |

**12.6.8.4.5 static void FLEXIO\_SPI\_SlaveTransferCreateHandleEDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_slave\\_edma\\_handle\\_t](#) \* *handle*, [flexio\\_spi\\_slave\\_edma\\_transfer\\_callback\\_t](#) *callback*, [void](#) \* *userData*, [edma\\_handle\\_t](#) \* *txHandle*, [edma\\_handle\\_t](#) \* *rxHandle* ) [inline], [static]**

This function initializes the FlexIO SPI slave eDMA handle.

## FlexIO SPI Driver

Parameters

|                 |                                                                                               |
|-----------------|-----------------------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                         |
| <i>handle</i>   | Pointer to <code>flexio_spi_slave_edma_handle_t</code> structure to store the transfer state. |
| <i>callback</i> | SPI callback, NULL means no callback.                                                         |
| <i>userData</i> | callback function parameter.                                                                  |
| <i>txHandle</i> | User requested eDMA handle for FlexIO SPI TX eDMA transfer.                                   |
| <i>rxHandle</i> | User requested eDMA handle for FlexIO SPI RX eDMA transfer.                                   |

**12.6.8.4.6 `status_t FLEXIO_SPI_SlaveTransferEDMA ( FLEXIO_SPI_Type * base,  
flexio_spi_slave_edma_handle_t * handle, flexio_spi_transfer_t * xfer )`**

Note

This interface returns immediately after transfer initiates. Call `FLEXIO_SPI_SlaveGetTransferCountEDMA` to poll the transfer status and check whether the FlexIO SPI transfer is finished.

Parameters

|               |                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                         |
| <i>handle</i> | Pointer to <code>flexio_spi_slave_edma_handle_t</code> structure to store the transfer state. |
| <i>xfer</i>   | Pointer to FlexIO SPI transfer structure.                                                     |

Return values

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

**12.6.8.4.7 `static void FLEXIO_SPI_SlaveTransferAbortEDMA ( FLEXIO_SPI_Type * base,  
flexio_spi_slave_edma_handle_t * handle ) [inline], [static]`**

Parameters

|               |                                                                                               |
|---------------|-----------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                         |
| <i>handle</i> | Pointer to <code>flexio_spi_slave_edma_handle_t</code> structure to store the transfer state. |

**12.6.8.4.8 static status\_t FLEXIO\_SPI\_SlaveTransferGetCountEDMA ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_slave\_edma\_handle\_t \* *handle*, size\_t \* *count* ) [inline], [static]**

Parameters

|               |                                                                     |
|---------------|---------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.               |
| <i>handle</i> | FlexIO SPI eDMA handle pointer.                                     |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction. |

### 12.6.9 FlexIO DMA SPI Driver

#### 12.6.9.1 Overview

#### Data Structures

- struct `flexio_spi_master_dma_handle_t`

*FlexIO SPI DMA transfer handle, users should not touch the content of the handle.* [More...](#)

#### TypeDefs

- typedef `flexio_spi_master_dma_handle_t flexio_spi_slave_dma_handle_t`  
*Slave handle is the same with master handle.*
- typedef void(\* `flexio_spi_master_dma_transfer_callback_t`)(`FLEXIO_SPI_Type` \*base, `flexio_spi_master_dma_handle_t` \*handle, `status_t` status, void \*userData)  
*FlexIO SPI master callback for finished transmit.*
- typedef void(\* `flexio_spi_slave_dma_transfer_callback_t`)(`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_dma_handle_t` \*handle, `status_t` status, void \*userData)  
*FlexIO SPI slave callback for finished transmit.*

#### DMA Transactional

- `status_t FLEXIO_SPI_MasterTransferCreateHandleDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_dma_handle_t` \*handle, `flexio_spi_master_dma_transfer_callback_t` callback, void \*userData, `dma_handle_t` \*txHandle, `dma_handle_t` \*rxHandle)  
*Initializes the FLEXIO SPI master DMA handle.*
- `status_t FLEXIO_SPI_MasterTransferDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_dma_handle_t` \*handle, `flexio_spi_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO SPI transfer using DMA.*
- `void FLEXIO_SPI_MasterTransferAbortDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_dma_handle_t` \*handle)  
*Aborts a FlexIO SPI transfer using DMA.*
- `status_t FLEXIO_SPI_MasterTransferGetCountDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_master_dma_handle_t` \*handle, `size_t` \*count)  
*Gets the remaining bytes for FlexIO SPI DMA transfer.*
- `static void FLEXIO_SPI_SlaveTransferCreateHandleDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_dma_handle_t` \*handle, `flexio_spi_slave_dma_transfer_callback_t` callback, void \*userData, `dma_handle_t` \*txHandle, `dma_handle_t` \*rxHandle)  
*Initializes the FlexIO SPI slave DMA handle.*
- `status_t FLEXIO_SPI_SlaveTransferDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_dma_handle_t` \*handle, `flexio_spi_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO SPI transfer using DMA.*
- `static void FLEXIO_SPI_SlaveTransferAbortDMA` (`FLEXIO_SPI_Type` \*base, `flexio_spi_slave_dma_handle_t` \*handle)  
*Aborts a FlexIO SPI transfer using DMA.*

- static status\_t **FLEXIO\_SPI\_SlaveTransferGetCountDMA** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_slave\_dma\_handle\_t \*handle, size\_t \*count)  
*Gets the remaining bytes to be transferred for FlexIO SPI DMA.*

## 12.6.9.2 Data Structure Documentation

### 12.6.9.2.1 struct \_flexio\_spi\_master\_dma\_handle

typedef for flexio\_spi\_master\_dma\_handle\_t in advance.

#### Data Fields

- size\_t **transferSize**  
*Total bytes to be transferred.*
- bool **txInProgress**  
*Send transfer in progress.*
- bool **rxInProgress**  
*Receive transfer in progress.*
- dma\_handle\_t \* **txHandle**  
*DMA handler for SPI send.*
- dma\_handle\_t \* **rxHandle**  
*DMA handler for SPI receive.*
- flexio\_spi\_master\_dma\_transfer\_callback\_t **callback**  
*Callback for SPI DMA transfer.*
- void \* **userData**  
*User Data for SPI DMA callback.*

#### 12.6.9.2.1.1 Field Documentation

##### 12.6.9.2.1.1.1 size\_t flexio\_spi\_master\_dma\_handle\_t::transferSize

### 12.6.9.3 Typedef Documentation

#### 12.6.9.3.1 typedef flexio\_spi\_master\_dma\_handle\_t flexio\_spi\_slave\_dma\_handle\_t

### 12.6.9.4 Function Documentation

#### 12.6.9.4.1 status\_t FLEXIO\_SPI\_MasterTransferCreateHandleDMA ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_master\_dma\_handle\_t \* *handle*, flexio\_spi\_master\_dma\_transfer\_callback\_t *callback*, void \* *userData*, dma\_handle\_t \* *txHandle*, dma\_handle\_t \* *rxHandle* )

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

## FlexIO SPI Driver

Parameters

|                 |                                                                                                  |
|-----------------|--------------------------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                            |
| <i>handle</i>   | Pointer to <a href="#">flexio_spi_master_dma_handle_t</a> structure to store the transfer state. |
| <i>callback</i> | SPI callback, NULL means no callback.                                                            |
| <i>userData</i> | callback function parameter.                                                                     |
| <i>txHandle</i> | User requested DMA handle for FlexIO SPI RX DMA transfer.                                        |
| <i>rxHandle</i> | User requested DMA handle for FlexIO SPI TX DMA transfer.                                        |

Return values

|                           |                                                    |
|---------------------------|----------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                    |
| <i>kStatus_OutOfRange</i> | The FlexIO SPI DMA type/handle table out of range. |

**12.6.9.4.2 status\_t FLEXIO\_SPI\_MasterTransferDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_master\\_dma\\_handle\\_t](#) \* *handle*, [flexio\\_spi\\_transfer\\_t](#) \* *xfer* )**

Note

This interface returned immediately after transfer initiates. Call [FLEXIO\\_SPI\\_MasterGetTransferCountDMA](#) to poll the transfer status to check whether the FlexIO SPI transfer is finished.

Parameters

|               |                                                                                                  |
|---------------|--------------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                            |
| <i>handle</i> | Pointer to <a href="#">flexio_spi_master_dma_handle_t</a> structure to store the transfer state. |
| <i>xfer</i>   | Pointer to FlexIO SPI transfer structure.                                                        |

Return values

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

**12.6.9.4.3 void FLEXIO\_SPI\_MasterTransferAbortDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_master\\_dma\\_handle\\_t](#) \* *handle* )**

Parameters

|               |                                                       |
|---------------|-------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>handle</i> | FlexIO SPI DMA handle pointer.                        |

#### 12.6.9.4.4 **status\_t FLEXIO\_SPI\_MasterTransferGetCountDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_master\\_dma\\_handle\\_t](#) \* *handle*, [size\\_t](#) \* *count* )**

Parameters

|               |                                                                     |
|---------------|---------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.               |
| <i>handle</i> | FlexIO SPI DMA handle pointer.                                      |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction. |

#### 12.6.9.4.5 **static void FLEXIO\_SPI\_SlaveTransferCreateHandleDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_slave\\_dma\\_handle\\_t](#) \* *handle*, [flexio\\_spi\\_slave\\_dma\\_transfer\\_callback\\_t](#) *callback*, [void](#) \* *userData*, [dma\\_handle\\_t](#) \* *txHandle*, [dma\\_handle\\_t](#) \* *rxHandle* ) [inline], [**static**]**

This function initializes the FlexIO SPI slave DMA handle.

Parameters

|                 |                                                                                                 |
|-----------------|-------------------------------------------------------------------------------------------------|
| <i>base</i>     | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                           |
| <i>handle</i>   | Pointer to <a href="#">flexio_spi_slave_dma_handle_t</a> structure to store the transfer state. |
| <i>callback</i> | SPI callback, NULL means no callback.                                                           |
| <i>userData</i> | callback function parameter.                                                                    |
| <i>txHandle</i> | User requested DMA handle for FlexIO SPI TX DMA transfer.                                       |
| <i>rxHandle</i> | User requested DMA handle for FlexIO SPI RX DMA transfer.                                       |

#### 12.6.9.4.6 **status\_t FLEXIO\_SPI\_SlaveTransferDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, [flexio\\_spi\\_slave\\_dma\\_handle\\_t](#) \* *handle*, [flexio\\_spi\\_transfer\\_t](#) \* *xfer* )**

Note

This interface returns immediately after transfer initiates. Call [FLEXIO\\_SPI\\_SlaveGetTransferCountDMA](#) to poll the transfer status and check whether the FlexIO SPI transfer is finished.

## FlexIO SPI Driver

Parameters

|               |                                                                                              |
|---------------|----------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                        |
| <i>handle</i> | Pointer to <code>flexio_spi_slave_dma_handle_t</code> structure to store the transfer state. |
| <i>xfer</i>   | Pointer to FlexIO SPI transfer structure.                                                    |

Return values

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

**12.6.9.4.7 static void FLEXIO\_SPI\_SlaveTransferAbortDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, `flexio_spi_slave_dma_handle_t` \* *handle* ) [inline], [static]**

Parameters

|               |                                                                                              |
|---------------|----------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.                                        |
| <i>handle</i> | Pointer to <code>flexio_spi_slave_dma_handle_t</code> structure to store the transfer state. |

**12.6.9.4.8 static status\_t FLEXIO\_SPI\_SlaveTransferGetCountDMA ( [FLEXIO\\_SPI\\_Type](#) \* *base*, `flexio_spi_slave_dma_handle_t` \* *handle*, `size_t` \* *count* ) [inline], [static]**

Parameters

|               |                                                                     |
|---------------|---------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_SPI_Type</a> structure.               |
| <i>handle</i> | FlexIO SPI DMA handle pointer.                                      |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction. |

## 12.7 FlexIO UART Driver

### 12.7.1 Overview

The KSDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) function using the Flexible I/O.

FlexIO UART driver includes functional APIs and transactional APIs. Functional APIs target low-level APIs. Functional APIs can be used for the FlexIO UART initialization/configuration/operation for optimization/customization purpose. Using the functional APIs requires the knowledge of the FlexIO UART peripheral and how to organize functional APIs to meet the application requirements. All functional API use the [FLEXIO\\_UART\\_Type](#) \* as the first parameter. FlexIO UART functional operation groups provide the functional APIs set.

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and also in the application if the code size and performance of transactional APIs satisfy requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the `flexio_uart_handle_t` as the second parameter. Initialize the handle by calling the [FLEXIO\\_UART\\_TransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer. This means that the functions `FLEXIO_UART_SendNonBlocking()` and `FLEXIO_UART_ReceiveNonBlocking()` set up an interrupt for data transfer. When the transfer is complete, the upper layer is notified through a callback function with the `kStatus_FLEXIO_UART_TxIdle` and `kStatus_FLEXIO_UART_RxIdle` status.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size through calling the `FLEXIO_UART_InstallRingBuffer()`. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The function `FLEXIO_UART_ReceiveNonBlocking()` first gets data from the ring buffer. If ring buffer does not have enough data, the function returns the data to the ring buffer and saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the `kStatus_FLEXIO_UART_RxIdle` status.

If the receive ring buffer is full, the upper layer is informed through a callback with status `kStatus_FLEXIO_UART_RxRingBufferOverrun`. In the callback function, the upper layer reads data from the ring buffer. If not, the oldest data is overwritten by the new data.

The ring buffer size is specified when calling the `FLEXIO_UART_InstallRingBuffer`. Note that one byte is reserved for the ring buffer maintenance. Create a handle as follows.

```
FLEXIO_UART_InstallRingBuffer(&uartDev, &handle, &ringBuffer, 32);
```

In this example, the buffer size is 32. However, only 31 bytes are used for saving data.

### 12.7.2 Typical use case

#### 12.7.2.1 FlexIO UART send/receive using a polling method

```
uint8_t ch;
```

## FlexIO UART Driver

```
FLEXIO_UART_Type uartDev;
status_t result = kStatus_Success;
flexio_uart_user_config user_config;
FLEXIO_UART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableUart = true;

uartDev.flexioBase = BOARD_FLEXIO_BASE;
uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
uartDev.shifterIndex[0] = 0U;
uartDev.shifterIndex[1] = 1U;
uartDev.timerIndex[0] = 0U;
uartDev.timerIndex[1] = 1U;

result = FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);
//Check if configuration is correct.
if(result != kStatus_Success)
{
 return;
}
FLEXIO_UART_WriteBlocking(&uartDev, txbuff, sizeof(txbuff));

while(1)
{
 FLEXIO_UART_ReadBlocking(&uartDev, &ch, 1);
 FLEXIO_UART_WriteBlocking(&uartDev, &ch, 1);
}
```

### 12.7.2.2 FlexIO UART send/receive using an interrupt method

```
FLEXIO_UART_Type uartDev;
flexio_uart_handle_t g_uartHandle;
flexio_uart_config_t user_config;
flexio_uart_transfer_t sendXfer;
flexio_uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void FLEXIO_UART_UserCallback(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle,
 status_t status, void *userData)
{
 userData = userData;

 if (kStatus_FLEXIO_UART_TxIdle == status)
 {
 txFinished = true;
 }

 if (kStatus_FLEXIO_UART_RxIdle == status)
 {
 rxFinished = true;
 }
}

void main(void)
{
 //...

 FLEXIO_UART_GetDefaultConfig(&user_config);
 user_config.baudRate_Bps = 115200U;
 user_config.enableUart = true;
```

```

uartDev.flexioBase = BOARD_FLEXIO_BASE;
uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
uartDev.shifterIndex[0] = 0U;
uartDev.shifterIndex[1] = 1U;
uartDev.timerIndex[0] = 0U;
uartDev.timerIndex[1] = 1U;

result = FLEXIO_UART_Init(&uartDev, &user_config, 1200000000U);
//Check if configuration is correct.
if(result != kStatus_Success)
{
 return;
}

FLEXIO_UART_TransferCreateHandle(&uartDev, &g_uartHandle,
 FLEXIO_UART_UserCallback, NULL);

// Prepares to send.
sendXfer.data = sendData;
sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
txFinished = false;

// Sends out.
FLEXIO_UART_SendNonBlocking(&uartDev, &g_uartHandle, &sendXfer);

// Send finished.
while (!txFinished)
{
}

// Prepares to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
rxFinished = false;

// Receives.
FLEXIO_UART_ReceiveNonBlocking(&uartDev, &g_uartHandle, &receiveXfer, NULL);

// Receive finished.
while (!rxFinished)
{
}

// ...
}

```

### 12.7.2.3 FlexIO UART receive using the ringbuffer feature

```

#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE 32

FLEXIO_UART_Type uartDev;
flexio_uart_handle_t g_uartHandle;
flexio_uart_config_t user_config;
flexio_uart_transfer_t sendXfer;
flexio_uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void FLEXIO_UART_UserCallback(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle,
 status_t status, void *userData)
{

```

## FlexIO UART Driver

```
userData = userData;

if (kStatus_FLEXIO_UART_RxIdle == status)
{
 rxFinished = true;
}
}

void main(void)
{
 size_t bytesRead;
//...

FLEXIO_UART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableUart = true;

uartDev.flexioBase = BOARD_FLEXIO_BASE;
uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
uartDev.shifterIndex[0] = 0U;
uartDev.shifterIndex[1] = 1U;
uartDev.timerIndex[0] = 0U;
uartDev.timerIndex[1] = 1U;

result = FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);
//Check if configuration is correct.
if(result != kStatus_Success)
{
 return;
}

FLEXIO_UART_TransferCreateHandle(&uartDev, &g_uartHandle,
 FLEXIO_UART_UserCallback, NULL);
FLEXIO_UART_InstallRingBuffer(&uartDev, &g_uartHandle, ringBuffer, RING_BUFFER_SIZE);

// Receive is working in the background to the ring buffer.

// Prepares to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = RX_DATA_SIZE;
rxFinished = false;

// Receives.
FLEXIO_UART_ReceiveNonBlocking(&uartDev, &g_uartHandle, &receiveXfer, &bytesRead);

if (bytesRead == RX_DATA_SIZE) /* Have read enough data. */
{
 ;
}
else
{
 if (bytesRead) /* Received some data, process first. */
 {
 ;
 }

 // Receive finished.
 while (!rxFinished)
 {
 }
}

// ...
}
```

### 12.7.2.4 FlexIO UART send/receive using a DMA method

```

FLEXIO_UART_Type uartDev;
flexio_uart_handle_t g_uartHandle;
dma_handle_t g_uartTxDmaHandle;
dma_handle_t g_uartRxDmaHandle;
flexio_uart_config_t user_config;
flexio_uart_transfer_t sendXfer;
flexio_uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void FLEXIO_UART_UserCallback(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle,
 status_t status, void *userData)
{
 userData = userData;

 if (kStatus_FLEXIO_UART_TxIdle == status)
 {
 txFinished = true;
 }

 if (kStatus_FLEXIO_UART_RxIdle == status)
 {
 rxFinished = true;
 }
}

void main(void)
{
 //...

 FLEXIO_UART_GetDefaultConfig(&user_config);
 user_config.baudRate_Bps = 115200U;
 user_config.enableUart = true;

 uartDev.flexioBase = BOARD_FLEXIO_BASE;
 uartDev.TxPinIndex = FLEXIO_UART_TX_PIN;
 uartDev.RxPinIndex = FLEXIO_UART_RX_PIN;
 uartDev.shifterIndex[0] = 0U;
 uartDev.shifterIndex[1] = 1U;
 uartDev.timerIndex[0] = 0U;
 uartDev.timerIndex[1] = 1U;
 result = FLEXIO_UART_Init(&uartDev, &user_config, 48000000U);
 //Check if configuration is correct.
 if(result != kStatus_Success)
 {
 return;
 }

 /*Initializes the DMA for the example*/
 DMAMGR_Init();

 dma_request_source_tx = (dma_request_source_t) (FLEXIO_DMA_REQUEST_BASE + uartDev.
 shifterIndex[0]);
 dma_request_source_rx = (dma_request_source_t) (FLEXIO_DMA_REQUEST_BASE + uartDev.
 shifterIndex[1]);

 /* Requests DMA channels for transmit and receive. */
 DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_tx, 0, &g_uartTxDmaHandle);
 DMAMGR_RequestChannel((dma_request_source_t)dma_request_source_rx, 1, &g_uartRxDmaHandle);

 FLEXIO_UART_TransferCreateHandleDMA(&uartDev, &g_uartHandle,
 FLEXIO_UART_UserCallback, NULL, &g_uartTxDmaHandle, &g_uartRxDmaHandle);
}

```

## FlexIO UART Driver

```
// Prepares to send.
sendXfer.data = sendData
sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
txFinished = false;

// Sends out.
FLEXIO_UART_SendDMA(&uartDev, &g_uartHandle, &sendXfer);

// Send finished.
while (!txFinished)
{
}

// Prepares to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
rxFinished = false;

// Receives.
FLEXIO_UART_ReceiveDMA(&uartDev, &g_uartHandle, &receiveXfer, NULL);

// Receive finished.
while (!rxFinished)
{
}

// ...
}
```

## Modules

- [FlexIO DMA UART Driver](#)
- [FlexIO eDMA UART Driver](#)

## Data Structures

- struct [FLEXIO\\_UART\\_Type](#)  
*Define FlexIO UART access structure typedef.* [More...](#)
- struct [flexio\\_uart\\_config\\_t](#)  
*Define FlexIO UART user configuration structure.* [More...](#)
- struct [flexio\\_uart\\_transfer\\_t](#)  
*Define FlexIO UART transfer structure.* [More...](#)
- struct [flexio\\_uart\\_handle\\_t](#)  
*Define FLEXIO UART handle structure.* [More...](#)

## Typedefs

- [typedef void\(\\* flexio\\_uart\\_transfer\\_callback\\_t \)\(FLEXIO\\_UART\\_Type \\*base, flexio\\_uart\\_handle\\_t \\*handle, status\\_t status, void \\*userData\)](#)  
*FlexIO UART transfer callback function.*

## Enumerations

- enum `_flexio_uart_status` {
   
kStatus\_FLEXIO\_UART\_TxBusy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_UART, 0),
   
kStatus\_FLEXIO\_UART\_RxBusy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_UART, 1),
   
kStatus\_FLEXIO\_UART\_TxIdle = MAKE\_STATUS(kStatusGroup\_FLEXIO\_UART, 2),
   
kStatus\_FLEXIO\_UART\_RxIdle = MAKE\_STATUS(kStatusGroup\_FLEXIO\_UART, 3),
   
kStatus\_FLEXIO\_UART\_ERROR = MAKE\_STATUS(kStatusGroup\_FLEXIO\_UART, 4),
   
kStatus\_FLEXIO\_UART\_RxRingBufferOverrun,
   
kStatus\_FLEXIO\_UART\_RxHardwareOverrun = MAKE\_STATUS(kStatusGroup\_FLEXIO\_UART, 6)
 }

*Error codes for the UART driver.*

- enum `flexio_uart_bit_count_per_char_t` {
   
kFLEXIO\_UART\_7BitsPerChar = 7U,
   
kFLEXIO\_UART\_8BitsPerChar = 8U,
   
kFLEXIO\_UART\_9BitsPerChar = 9U
 }

*FlexIO UART bit count per char.*

- enum `_flexio_uart_interrupt_enable` {
   
kFLEXIO\_UART\_TxDataRegEmptyInterruptEnable = 0x1U,
   
kFLEXIO\_UART\_RxDataRegFullInterruptEnable = 0x2U
 }

*Define FlexIO UART interrupt mask.*

- enum `_flexio_uart_status_flags` {
   
kFLEXIO\_UART\_TxDataRegEmptyFlag = 0x1U,
   
kFLEXIO\_UART\_RxDataRegFullFlag = 0x2U,
   
kFLEXIO\_UART\_RxOverRunFlag = 0x4U
 }

*Define FlexIO UART status mask.*

## Driver version

- #define `FSL_FLEXIO_UART_DRIVER_VERSION` (MAKE\_VERSION(2, 1, 2))  
*FlexIO UART driver version 2.1.2.*

## Initialization and deinitialization

- `status_t FLEXIO_UART_Init (FLEXIO_UART_Type *base, const flexio_uart_config_t *userConfig, uint32_t srcClock_Hz)`  
*Ungates the FlexIO clock, resets the FlexIO module, configures FlexIO UART hardware, and configures the FlexIO UART with FlexIO UART configuration.*
- `void FLEXIO_UART_Deinit (FLEXIO_UART_Type *base)`  
*Disables the FlexIO UART and gates the FlexIO clock.*
- `void FLEXIO_UART_GetDefaultConfig (flexio_uart_config_t *userConfig)`  
*Gets the default configuration to configure the FlexIO UART.*

## FlexIO UART Driver

### Status

- `uint32_t FLEXIO_UART_GetStatusFlags (FLEXIO_UART_Type *base)`  
*Gets the FlexIO UART status flags.*
- `void FLEXIO_UART_ClearStatusFlags (FLEXIO_UART_Type *base, uint32_t mask)`  
*Gets the FlexIO UART status flags.*

### Interrupts

- `void FLEXIO_UART_EnableInterrupts (FLEXIO_UART_Type *base, uint32_t mask)`  
*Enables the FlexIO UART interrupt.*
- `void FLEXIO_UART_DisableInterrupts (FLEXIO_UART_Type *base, uint32_t mask)`  
*Disables the FlexIO UART interrupt.*

### DMA Control

- `static uint32_t FLEXIO_UART_GetTxDataRegisterAddress (FLEXIO_UART_Type *base)`  
*Gets the FlexIO UART transmit data register address.*
- `static uint32_t FLEXIO_UART_GetRxDataRegisterAddress (FLEXIO_UART_Type *base)`  
*Gets the FlexIO UART receive data register address.*
- `static void FLEXIO_UART_EnableTxDMA (FLEXIO_UART_Type *base, bool enable)`  
*Enables/disables the FlexIO UART transmit DMA.*
- `static void FLEXIO_UART_EnableRxDMA (FLEXIO_UART_Type *base, bool enable)`  
*Enables/disables the FlexIO UART receive DMA.*

### Bus Operations

- `static void FLEXIO_UART_Enable (FLEXIO_UART_Type *base, bool enable)`  
*Enables/disables the FlexIO UART module operation.*
- `static void FLEXIO_UART_WriteByte (FLEXIO_UART_Type *base, const uint8_t *buffer)`  
*Writes one byte of data.*
- `static void FLEXIO_UART_ReadByte (FLEXIO_UART_Type *base, uint8_t *buffer)`  
*Reads one byte of data.*
- `void FLEXIO_UART_WriteBlocking (FLEXIO_UART_Type *base, const uint8_t *txData, size_t txSize)`  
*Sends a buffer of data bytes.*
- `void FLEXIO_UART_ReadBlocking (FLEXIO_UART_Type *base, uint8_t *rxData, size_t rxSize)`  
*Receives a buffer of bytes.*

### Transactional

- `status_t FLEXIO_UART_TransferCreateHandle (FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, flexio_uart_transfer_callback_t callback, void *userData)`  
*Initializes the UART handle.*

- void **FLEXIO\_UART\_TransferStartRingBuffer** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle, uint8\_t \*ringBuffer, size\_t ringBufferSize)  
*Sets up the RX ring buffer.*
- void **FLEXIO\_UART\_TransferStopRingBuffer** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle)  
*Aborts the background transfer and uninstalls the ring buffer.*
- status\_t **FLEXIO\_UART\_TransferSendNonBlocking** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle, **flexio\_uart\_transfer\_t** \*xfer)  
*Transmits a buffer of data using the interrupt method.*
- void **FLEXIO\_UART\_TransferAbortSend** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle)  
*Aborts the interrupt-driven data transmit.*
- status\_t **FLEXIO\_UART\_TransferGetSendCount** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle, size\_t \*count)  
*Gets the number of bytes sent.*
- status\_t **FLEXIO\_UART\_TransferReceiveNonBlocking** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle, **flexio\_uart\_transfer\_t** \*xfer, size\_t \*receivedBytes)  
*Receives a buffer of data using the interrupt method.*
- void **FLEXIO\_UART\_TransferAbortReceive** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle)  
*Aborts the receive data which was using IRQ.*
- status\_t **FLEXIO\_UART\_TransferGetReceiveCount** (**FLEXIO\_UART\_Type** \*base, flexio\_uart\_handle\_t \*handle, size\_t \*count)  
*Gets the number of bytes received.*
- void **FLEXIO\_UART\_TransferHandleIRQ** (void \*uartType, void \*uartHandle)  
*FlexIO UART IRQ handler function.*

## 12.7.3 Data Structure Documentation

### 12.7.3.1 struct FLEXIO\_UART\_Type

#### Data Fields

- **FLEXIO\_Type** \* **flexioBase**  
*FlexIO base pointer.*
- uint8\_t **TxPinIndex**  
*Pin select for UART\_Tx.*
- uint8\_t **RxPinIndex**  
*Pin select for UART\_Rx.*
- uint8\_t **shifterIndex** [2]  
*Shifter index used in FlexIO UART.*
- uint8\_t **timerIndex** [2]  
*Timer index used in FlexIO UART.*

## FlexIO UART Driver

### 12.7.3.1.0.1 Field Documentation

12.7.3.1.0.1.1 `FLEXIO_Type* FLEXIO_UART_Type::flexioBase`

12.7.3.1.0.1.2 `uint8_t FLEXIO_UART_Type::TxPinIndex`

12.7.3.1.0.1.3 `uint8_t FLEXIO_UART_Type::RxPinIndex`

12.7.3.1.0.1.4 `uint8_t FLEXIO_UART_Type::shifterIndex[2]`

12.7.3.1.0.1.5 `uint8_t FLEXIO_UART_Type::timerIndex[2]`

### 12.7.3.2 `struct flexio_uart_config_t`

#### Data Fields

- `bool enableUart`  
*Enable/disable FlexIO UART TX & RX.*
- `bool enableInDoze`  
*Enable/disable FlexIO operation in doze mode.*
- `bool enableInDebug`  
*Enable/disable FlexIO operation in debug mode.*
- `bool enableFastAccess`  
*Enable/disable fast access to FlexIO registers,  
fast access requires the FlexIO clock to be at least twice the frequency of the bus clock.*
- `uint32_t baudRate_Bps`  
*Baud rate in Bps.*
- `flexio_uart_bit_count_per_char_t bitCountPerChar`  
*number of bits, 7/8/9 -bit*

### 12.7.3.2.0.2 Field Documentation

12.7.3.2.0.2.1 `bool flexio_uart_config_t::enableUart`

12.7.3.2.0.2.2 `bool flexio_uart_config_t::enableFastAccess`

12.7.3.2.0.2.3 `uint32_t flexio_uart_config_t::baudRate_Bps`

### 12.7.3.3 `struct flexio_uart_transfer_t`

#### Data Fields

- `uint8_t * data`  
*Transfer buffer.*
- `size_t dataSize`  
*Transfer size.*

### 12.7.3.4 struct \_flexio\_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.*
- `uint8_t *volatile rxData`  
*Address of remaining data to receive.*
- `volatile size_t rxDataSize`  
*Size of the remaining data to receive.*
- `size_t txDataSizeAll`  
*Total bytes to be sent.*
- `size_t rxDataSizeAll`  
*Total bytes to be received.*
- `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.*
- `flexio_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.*

## FlexIO UART Driver

### 12.7.3.4.0.3 Field Documentation

- 12.7.3.4.0.3.1 `uint8_t* volatile flexio_uart_handle_t::txData`
- 12.7.3.4.0.3.2 `volatile size_t flexio_uart_handle_t::txDataSize`
- 12.7.3.4.0.3.3 `uint8_t* volatile flexio_uart_handle_t::rxData`
- 12.7.3.4.0.3.4 `volatile size_t flexio_uart_handle_t::rxDataSize`
- 12.7.3.4.0.3.5 `size_t flexio_uart_handle_t::txDataSizeAll`
- 12.7.3.4.0.3.6 `size_t flexio_uart_handle_t::rxDataSizeAll`
- 12.7.3.4.0.3.7 `uint8_t* flexio_uart_handle_t::rxRingBuffer`
- 12.7.3.4.0.3.8 `size_t flexio_uart_handle_t::rxRingBufferSize`
- 12.7.3.4.0.3.9 `volatile uint16_t flexio_uart_handle_t::rxRingBufferHead`
- 12.7.3.4.0.3.10 `volatile uint16_t flexio_uart_handle_t::rxRingBufferTail`
- 12.7.3.4.0.3.11 `flexio_uart_transfer_callback_t flexio_uart_handle_t::callback`
- 12.7.3.4.0.3.12 `void* flexio_uart_handle_t::userData`
- 12.7.3.4.0.3.13 `volatile uint8_t flexio_uart_handle_t::txState`

### 12.7.4 Macro Definition Documentation

- 12.7.4.1 `#define FSL_FLEXIO_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))`

### 12.7.5 Typedef Documentation

- 12.7.5.1 `typedef void(* flexio_uart_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, status_t status, void *userData)`

### 12.7.6 Enumeration Type Documentation

#### 12.7.6.1 enum \_flexio\_uart\_status

Enumerator

- `kStatus_FLEXIO_UART_TxBusy` Transmitter is busy.
- `kStatus_FLEXIO_UART_RxBusy` Receiver is busy.
- `kStatus_FLEXIO_UART_TxIdle` UART transmitter is idle.
- `kStatus_FLEXIO_UART_RxIdle` UART receiver is idle.
- `kStatus_FLEXIO_UART_ERROR` ERROR happens on UART.

*kStatus\_FLEXIO\_UART\_RxRingBufferOverrun*  UART RX software ring buffer overrun.  
*kStatus\_FLEXIO\_UART\_RxHardwareOverrun*  UART RX receiver overrun.

### 12.7.6.2 enum flexio\_uart\_bit\_count\_per\_char\_t

Enumerator

*kFLEXIO\_UART\_7BitsPerChar*  7-bit data characters  
*kFLEXIO\_UART\_8BitsPerChar*  8-bit data characters  
*kFLEXIO\_UART\_9BitsPerChar*  9-bit data characters

### 12.7.6.3 enum \_flexio\_uart\_interrupt\_enable

Enumerator

*kFLEXIO\_UART\_TxDataRegEmptyInterruptEnable*  Transmit buffer empty interrupt enable.  
*kFLEXIO\_UART\_RxDataRegFullInterruptEnable*  Receive buffer full interrupt enable.

### 12.7.6.4 enum \_flexio\_uart\_status\_flags

Enumerator

*kFLEXIO\_UART\_TxDataRegEmptyFlag*  Transmit buffer empty flag.  
*kFLEXIO\_UART\_RxDataRegFullFlag*  Receive buffer full flag.  
*kFLEXIO\_UART\_RxOverRunFlag*  Receive buffer over run flag.

## 12.7.7 Function Documentation

### 12.7.7.1 status\_t FLEXIO\_UART\_Init ( FLEXIO\_UART\_Type \* *base*, const flexio\_uart\_config\_t \* *userConfig*, uint32\_t *srcClock\_Hz* )

The configuration structure can be filled by the user or be set with default values by [FLEXIO\\_UART\\_GetDefaultConfig\(\)](#).

Example

```
FLEXIO_UART_Type base = {
 .flexioBase = FLEXIO,
 .TxPinIndex = 0,
 .RxPinIndex = 1,
 .shifterIndex = {0,1},
 .timerIndex = {0,1}
};
flexio_uart_config_t config = {
 .enableInDoze = false,
```

## FlexIO UART Driver

```
.enableInDebug = true,
.enableFastAccess = false,
.baudRate_Bps = 115200U,
.bitCountPerChar = 8
};
FLEXIO_UART_Init(base, &config, srcClock_Hz);
```

Parameters

|                    |                                                       |
|--------------------|-------------------------------------------------------|
| <i>base</i>        | Pointer to the <b>FLEXIO_UART_Type</b> structure.     |
| <i>userConfig</i>  | Pointer to the <b>flexio_uart_config_t</b> structure. |
| <i>srcClock_Hz</i> | FlexIO source clock in Hz.                            |

Return values

|                                |                                     |
|--------------------------------|-------------------------------------|
| <i>kStatus_Success</i>         | Configuration success               |
| <i>kStatus_InvalidArgument</i> | Baudrate configuration out of range |

### 12.7.7.2 void **FLEXIO\_UART\_Deinit**( **FLEXIO\_UART\_Type** \* *base* )

Note

After calling this API, call the **FLEXIO\_UART\_Init** to use the FlexIO UART module.

Parameters

|             |                                              |
|-------------|----------------------------------------------|
| <i>base</i> | Pointer to <b>FLEXIO_UART_Type</b> structure |
|-------------|----------------------------------------------|

### 12.7.7.3 void **FLEXIO\_UART\_GetDefaultConfig**( **flexio\_uart\_config\_t** \* *userConfig* )

The configuration can be used directly for calling the **FLEXIO\_UART\_Init()**. Example:

```
flexio_uart_config_t config;
FLEXIO_UART_GetDefaultConfig(&userConfig);
```

Parameters

|                   |                                                       |
|-------------------|-------------------------------------------------------|
| <i>userConfig</i> | Pointer to the <b>flexio_uart_config_t</b> structure. |
|-------------------|-------------------------------------------------------|

### 12.7.7.4 uint32\_t **FLEXIO\_UART\_GetStatusFlags**( **FLEXIO\_UART\_Type** \* *base* )

Parameters

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

Returns

FlexIO UART status flags.

#### 12.7.7.5 void FLEXIO\_UART\_ClearStatusFlags ( [FLEXIO\\_UART\\_Type](#) \* *base*, [uint32\\_t](#) *mask* )

Parameters

|             |                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                                                                                                                                                                                                                               |
| <i>mask</i> | Status flag. The parameter can be any combination of the following values: <ul style="list-style-type: none"> <li>• <a href="#">kFLEXIO_UART_TxDataRegEmptyFlag</a></li> <li>• <a href="#">kFLEXIO_UART_RxEmptyFlag</a></li> <li>• <a href="#">kFLEXIO_UART_RxOverRunFlag</a></li> </ul> |

#### 12.7.7.6 void FLEXIO\_UART\_EnableInterrupts ( [FLEXIO\\_UART\\_Type](#) \* *base*, [uint32\\_t](#) *mask* )

This function enables the FlexIO UART interrupt.

Parameters

|             |                                                            |
|-------------|------------------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>mask</i> | Interrupt source.                                          |

#### 12.7.7.7 void FLEXIO\_UART\_DisableInterrupts ( [FLEXIO\\_UART\\_Type](#) \* *base*, [uint32\\_t](#) *mask* )

This function disables the FlexIO UART interrupt.

Parameters

## FlexIO UART Driver

|             |                                                            |
|-------------|------------------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>mask</i> | Interrupt source.                                          |

### **12.7.7.8 static uint32\_t FLEXIO\_UART\_GetTxDataRegisterAddress ( FLEXIO\_UART\_Type \* *base* ) [inline], [static]**

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

Parameters

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

Returns

FlexIO UART transmit data register address.

### **12.7.7.9 static uint32\_t FLEXIO\_UART\_GetRxDataRegisterAddress ( FLEXIO\_UART\_Type \* *base* ) [inline], [static]**

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

Parameters

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

Returns

FlexIO UART receive data register address.

### **12.7.7.10 static void FLEXIO\_UART\_EnableTxDMA ( FLEXIO\_UART\_Type \* *base*, bool *enable* ) [inline], [static]**

This function enables/disables the FlexIO UART Tx DMA, which means asserting the kFLEXIO\_UART\_TxDataRegEmptyFlag does/doesn't trigger the DMA request.

Parameters

|               |                                                            |
|---------------|------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>enable</i> | True to enable, false to disable.                          |

**12.7.7.11 static void FLEXIO\_UART\_EnableRxDMA ( FLEXIO\_UART\_Type \* *base*, bool *enable* ) [inline], [static]**

This function enables/disables the FlexIO UART Rx DMA, which means asserting kFLEXIO\_UART\_RxDataRegFullFlag does/doesn't trigger the DMA request.

## FlexIO UART Driver

Parameters

|               |                                                            |
|---------------|------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>enable</i> | True to enable, false to disable.                          |

### 12.7.7.12 static void FLEXIO\_UART\_Enable ( [FLEXIO\\_UART\\_Type](#) \* *base*, *bool enable* ) [inline], [static]

Parameters

|               |                                                   |
|---------------|---------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> . |
| <i>enable</i> | True to enable, false to disable.                 |

### 12.7.7.13 static void FLEXIO\_UART\_WriteByte ( [FLEXIO\\_UART\\_Type](#) \* *base*, *const uint8\_t* \* *buffer* ) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is put into the data register. Ensure that the TxEmptyFlag is asserted before calling this API.

Parameters

|               |                                                            |
|---------------|------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>buffer</i> | The data bytes to send.                                    |

### 12.7.7.14 static void FLEXIO\_UART\_ReadByte ( [FLEXIO\\_UART\\_Type](#) \* *base*, *uint8\_t* \* *buffer* ) [inline], [static]

Note

This is a non-blocking API, which returns directly after the data is read from the data register. Ensure that the RxFullFlag is asserted before calling this API.

Parameters

|               |                                                            |
|---------------|------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>buffer</i> | The buffer to store the received bytes.                    |

#### 12.7.7.15 void FLEXIO\_UART\_WriteBlocking ( [FLEXIO\\_UART\\_Type](#) \* *base*, const [uint8\\_t](#) \* *txData*, [size\\_t](#) *txSize* )

Note

This function blocks using the polling method until all bytes have been sent.

Parameters

|               |                                                            |
|---------------|------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>txData</i> | The data bytes to send.                                    |
| <i>txSize</i> | The number of data bytes to send.                          |

#### 12.7.7.16 void FLEXIO\_UART\_ReadBlocking ( [FLEXIO\\_UART\\_Type](#) \* *base*, [uint8\\_t](#) \* *rxData*, [size\\_t](#) *rxSize* )

Note

This function blocks using the polling method until all bytes have been received.

Parameters

|               |                                                            |
|---------------|------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure. |
| <i>rxData</i> | The buffer to store the received bytes.                    |
| <i>rxSize</i> | The number of data bytes to be received.                   |

#### 12.7.7.17 status\_t FLEXIO\_UART\_TransferCreateHandle ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_handle\\_t](#) \* *handle*, [flexio\\_uart\\_transfer\\_callback\\_t](#) *callback*, [void](#) \* *userData* )

This function initializes the FlexIO UART handle, which can be used for other FlexIO UART transactional APIs. Call this API once to get the initialized handle.

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

## FlexIO UART Driver

Parameters

|                 |                                                                            |
|-----------------|----------------------------------------------------------------------------|
| <i>base</i>     | to <a href="#">FLEXIO_UART_Type</a> structure.                             |
| <i>handle</i>   | Pointer to the flexio_uart_handle_t structure to store the transfer state. |
| <i>callback</i> | The callback function.                                                     |
| <i>userData</i> | The parameter of the callback function.                                    |

Return values

|                           |                                                |
|---------------------------|------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                |
| <i>kStatus_OutOfRange</i> | The FlexIO type/handle/ISR table out of range. |

### 12.7.7.18 void FLEXIO\_UART\_TransferStartRingBuffer ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_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 is stored into the ring buffer even when the user doesn't call the [UART\\_ReceiveNonBlocking\(\)](#) API. If there is already data received in the ring buffer, users 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>           | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                                   |
| <i>handle</i>         | Pointer to the flexio_uart_handle_t structure to store the transfer state.                   |
| <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.                                                                     |

### 12.7.7.19 void FLEXIO\_UART\_TransferStopRingBuffer ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_handle\\_t](#) \* *handle* )

This function aborts the background transfer and uninstalls the ring buffer.

## Parameters

|               |                                                                                            |
|---------------|--------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                                 |
| <i>handle</i> | Pointer to the <a href="#">flexio_uart_handle_t</a> structure to store the transfer state. |

**12.7.7.20 status\_t FLEXIO\_UART\_TransferSendNonBlocking ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_handle\\_t](#) \* *handle*, [flexio\\_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 ISR, the FlexIO UART driver calls the callback function and passes the [kStatus\\_FLEXIO\\_UART\\_TxIdle](#) as status parameter.

## Note

The [kStatus\\_FLEXIO\\_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.

## Parameters

|               |                                                                                            |
|---------------|--------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                                 |
| <i>handle</i> | Pointer to the <a href="#">flexio_uart_handle_t</a> structure to store the transfer state. |
| <i>xfer</i>   | FlexIO UART transfer structure. See <a href="#">flexio_uart_transfer_t</a> .               |

## Return values

|                            |                                                                                |
|----------------------------|--------------------------------------------------------------------------------|
| <i>kStatus_Success</i>     | Successfully starts the data transmission.                                     |
| <i>kStatus_UART_TxBusy</i> | Previous transmission still not finished, data not written to the TX register. |

**12.7.7.21 void FLEXIO\_UART\_TransferAbortSend ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_handle\\_t](#) \* *handle* )**

This function aborts the interrupt-driven data sending. Get the *remainBytes* to find out how many bytes are still not sent out.

## Parameters

## FlexIO UART Driver

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                 |
| <i>handle</i> | Pointer to the flexio_uart_handle_t structure to store the transfer state. |

### 12.7.7.22 [status\\_t FLEXIO\\_UART\\_TransferGetSendCount \( FLEXIO\\_UART\\_Type \\* base, flexio\\_uart\\_handle\\_t \\* handle, size\\_t \\* count \)](#)

This function gets the number of bytes sent driven by interrupt.

Parameters

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                 |
| <i>handle</i> | Pointer to the flexio_uart_handle_t structure to store the transfer state. |
| <i>count</i>  | Number of bytes sent so far by the non-blocking transaction.               |

Return values

|                                     |                                                   |
|-------------------------------------|---------------------------------------------------|
| <i>kStatus_NoTransferInProgress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>              | Successfully return the count.                    |

### 12.7.7.23 [status\\_t FLEXIO\\_UART\\_TransferReceiveNonBlocking \( FLEXIO\\_UART\\_Type \\* base, flexio\\_uart\\_handle\\_t \\* handle, flexio\\_uart\\_transfer\\_t \\* xfer, size\\_t \\* receivedBytes \)](#)

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 UART driver. When 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, if the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer, the 5 bytes are copied to `xfer->data`. This function returns with the parameter `receivedBytes` set to 5. For the last 5 bytes, newly arrived data is saved from the `xfer->data[5]`. When 5 bytes are received, the UART driver notifies upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to `xfer->data`. When all data is received, the upper layer is notified.

Parameters

|                      |                                                                            |
|----------------------|----------------------------------------------------------------------------|
| <i>base</i>          | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                 |
| <i>handle</i>        | Pointer to the flexio_uart_handle_t structure to store the transfer state. |
| <i>xfer</i>          | UART transfer structure. See <a href="#">flexio_uart_transfer_t</a> .      |
| <i>receivedBytes</i> | Bytes received from the ring buffer directly.                              |

Return values

|                                   |                                                          |
|-----------------------------------|----------------------------------------------------------|
| <i>kStatus_Success</i>            | Successfully queue the transfer into the transmit queue. |
| <i>kStatus_FLEXIO_UART_RxBusy</i> | Previous receive request is not finished.                |

#### 12.7.7.24 void FLEXIO\_UART\_TransferAbortReceive ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_handle\\_t](#) \* *handle* )

This function aborts the receive data which was using IRQ.

Parameters

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                 |
| <i>handle</i> | Pointer to the flexio_uart_handle_t structure to store the transfer state. |

#### 12.7.7.25 [status\\_t](#) FLEXIO\_UART\_TransferGetReceiveCount ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_handle\\_t](#) \* *handle*, [size\\_t](#) \* *count* )

This function gets the number of bytes received driven by interrupt.

Parameters

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                 |
| <i>handle</i> | Pointer to the flexio_uart_handle_t structure to store the transfer state. |
| <i>count</i>  | Number of bytes received so far by the non-blocking transaction.           |

Return values

|                                     |                                                   |
|-------------------------------------|---------------------------------------------------|
| <i>kStatus_NoTransferInProgress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>              | Successfully return the count.                    |

### 12.7.7.26 void FLEXIO\_UART\_TransferHandleIRQ ( void \* *uartType*, void \* *uartHandle* )

This function processes the FlexIO UART transmit and receives the IRQ request.

## Parameters

|                   |                                                                                         |
|-------------------|-----------------------------------------------------------------------------------------|
| <i>uartType</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                              |
| <i>uartHandle</i> | Pointer to the <code>flexio_uart_handle_t</code> structure to store the transfer state. |

### 12.7.8 FlexIO eDMA UART Driver

#### 12.7.8.1 Overview

#### Data Structures

- struct `flexio_uart_edma_handle_t`  
*UART eDMA handle.* [More...](#)

#### TypeDefs

- typedef void(\* `flexio_uart_edma_transfer_callback_t`)(`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle, `status_t` status, `void` \*userData)  
*UART transfer callback function.*

#### eDMA transactional

- `status_t FLEXIO_UART_TransferCreateHandleEDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle, `flexio_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 FLEXIO_UART_TransferSendEDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle, `flexio_uart_transfer_t` \*xfer)  
*Sends data using eDMA.*
- `status_t FLEXIO_UART_TransferReceiveEDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle, `flexio_uart_transfer_t` \*xfer)  
*Receives data using eDMA.*
- `void FLEXIO_UART_TransferAbortSendEDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle)  
*Aborts the sent data which using eDMA.*
- `void FLEXIO_UART_TransferAbortReceiveEDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle)  
*Aborts the receive data which using eDMA.*
- `status_t FLEXIO_UART_TransferGetSendCountEDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle, `size_t` \*count)  
*Gets the number of bytes sent out.*
- `status_t FLEXIO_UART_TransferGetReceiveCountEDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_edma_handle_t` \*handle, `size_t` \*count)  
*Gets the number of bytes received.*

### 12.7.8.2 Data Structure Documentation

#### 12.7.8.2.1 struct \_flexio\_uart\_edma\_handle

##### Data Fields

- **flexio\_uart\_edma\_transfer\_callback\_t callback**  
*Callback function.*
- **void \*userData**  
*UART callback function parameter.*
- **size\_t txDataSizeAll**  
*Total bytes to be sent.*
- **size\_t rxDataSizeAll**  
*Total bytes to be received.*
- **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.*

## FlexIO UART Driver

### 12.7.8.2.1.1 Field Documentation

- 12.7.8.2.1.1.1 `flexio_uart_edma_transfer_callback_t flexio_uart_edma_handle_t::callback`
- 12.7.8.2.1.1.2 `void* flexio_uart_edma_handle_t::userData`
- 12.7.8.2.1.1.3 `size_t flexio_uart_edma_handle_t::txDataSizeAll`
- 12.7.8.2.1.1.4 `size_t flexio_uart_edma_handle_t::rxDataSizeAll`
- 12.7.8.2.1.1.5 `edma_handle_t* flexio_uart_edma_handle_t::txEdmaHandle`
- 12.7.8.2.1.1.6 `edma_handle_t* flexio_uart_edma_handle_t::rxEdmaHandle`
- 12.7.8.2.1.1.7 `uint8_t flexio_uart_edma_handle_t::nbytes`
- 12.7.8.2.1.1.8 `volatile uint8_t flexio_uart_edma_handle_t::txState`

### 12.7.8.3 Typedef Documentation

- 12.7.8.3.1 `typedef void(* flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_edma_handle_t *handle, status_t status, void *userData)`

### 12.7.8.4 Function Documentation

- 12.7.8.4.1 `status_t FLEXIO_UART_TransferCreateHandleEDMA ( FLEXIO_UART_Type * base, flexio_uart_edma_handle_t * handle, flexio_uart_edma_transfer_callback_t callback, void * userData, edma_handle_t * txEdmaHandle, edma_handle_t * rxEdmaHandle )`

Parameters

|                     |                                                                 |
|---------------------|-----------------------------------------------------------------|
| <i>base</i>         | Pointer to <a href="#">FLEXIO_UART_Type</a> .                   |
| <i>handle</i>       | Pointer to <a href="#">flexio_uart_edma_handle_t</a> structure. |
| <i>callback</i>     | The callback function.                                          |
| <i>userData</i>     | The parameter of the callback function.                         |
| <i>rxEdmaHandle</i> | User requested DMA handle for RX DMA transfer.                  |
| <i>txEdmaHandle</i> | User requested DMA handle for TX DMA transfer.                  |

Return values

|                           |                                                     |
|---------------------------|-----------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                     |
| <i>kStatus_OutOfRange</i> | The FlexIO SPI eDMA type/handle table out of range. |

#### 12.7.8.4.2 `status_t FLEXIO_UART_TransferSendEDMA ( FLEXIO_UART_Type * base, flexio_uart_edma_handle_t * handle, flexio_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 out, the send callback function is called.

Parameters

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                                |
| <i>handle</i> | UART handle pointer.                                                       |
| <i>xfer</i>   | UART eDMA transfer structure, see <a href="#">flexio_uart_transfer_t</a> . |

Return values

|                                   |                             |
|-----------------------------------|-----------------------------|
| <i>kStatus_Success</i>            | if succeed, others failed.  |
| <i>kStatus_FLEXIO_UART_TxBusy</i> | Previous transfer on going. |

#### 12.7.8.4.3 `status_t FLEXIO_UART_TransferReceiveEDMA ( FLEXIO_UART_Type * base, flexio_uart_edma_handle_t * handle, flexio_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.

## FlexIO UART Driver

Parameters

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                                |
| <i>handle</i> | Pointer to <a href="#">flexio_uart_edma_handle_t</a> structure             |
| <i>xfer</i>   | UART eDMA transfer structure, see <a href="#">flexio_uart_transfer_t</a> . |

Return values

|                            |                             |
|----------------------------|-----------------------------|
| <i>kStatus_Success</i>     | if succeed, others failed.  |
| <i>kStatus_UART_RxBusy</i> | Previous transfer on going. |

### **12.7.8.4.4 void FLEXIO\_UART\_TransferAbortSendEDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_edma\\_handle\\_t](#) \* *handle* )**

This function aborts sent data which using eDMA.

Parameters

|               |                                                                |
|---------------|----------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                    |
| <i>handle</i> | Pointer to <a href="#">flexio_uart_edma_handle_t</a> structure |

### **12.7.8.4.5 void FLEXIO\_UART\_TransferAbortReceiveEDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_edma\\_handle\\_t](#) \* *handle* )**

This function aborts the receive data which using eDMA.

Parameters

|               |                                                                |
|---------------|----------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                    |
| <i>handle</i> | Pointer to <a href="#">flexio_uart_edma_handle_t</a> structure |

### **12.7.8.4.6 status\_t FLEXIO\_UART\_TransferGetSendCountEDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_edma\\_handle\\_t](#) \* *handle*, [size\\_t](#) \* *count* )**

This function gets the number of bytes sent out.

Parameters

|               |                                                              |
|---------------|--------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                  |
| <i>handle</i> | Pointer to <code>flexio_uart_edma_handle_t</code> structure  |
| <i>count</i>  | Number of bytes sent so far by the non-blocking transaction. |

Return values

|                                      |                                                   |
|--------------------------------------|---------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>               | Successfully return the count.                    |

#### **12.7.8.4.7 `status_t FLEXIO_UART_TransferGetReceiveCountEDMA ( FLEXIO_UART_Type * base, flexio_uart_edma_handle_t * handle, size_t * count )`**

This function gets the number of bytes received.

Parameters

|               |                                                                  |
|---------------|------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                      |
| <i>handle</i> | Pointer to <code>flexio_uart_edma_handle_t</code> structure      |
| <i>count</i>  | Number of bytes received so far by the non-blocking transaction. |

Return values

|                                      |                                                   |
|--------------------------------------|---------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>               | Successfully return the count.                    |

### 12.7.9 FlexIO DMA UART Driver

#### 12.7.9.1 Overview

#### Data Structures

- struct `flexio_uart_dma_handle_t`  
*UART DMA handle.* [More...](#)

#### TypeDefs

- typedef void(\* `flexio_uart_dma_transfer_callback_t` )(`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle, `status_t` status, `void` \*userData)  
*UART transfer callback function.*

#### eDMA transactional

- `status_t FLEXIO_UART_TransferCreateHandleDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle, `flexio_uart_dma_transfer_callback_t` callback, `void` \*userData, `dma_handle_t` \*txDmaHandle, `dma_handle_t` \*rxDmaHandle)  
*Initializes the FLEXIO\_UART handle which is used in transactional functions.*
- `status_t FLEXIO_UART_TransferSendDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle, `flexio_uart_transfer_t` \*xfer)  
*Sends data using DMA.*
- `status_t FLEXIO_UART_TransferReceiveDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle, `flexio_uart_transfer_t` \*xfer)  
*Receives data using DMA.*
- `void FLEXIO_UART_TransferAbortSendDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle)  
*Aborts the sent data which using DMA.*
- `void FLEXIO_UART_TransferAbortReceiveDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle)  
*Aborts the receive data which using DMA.*
- `status_t FLEXIO_UART_TransferGetSendCountDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle, `size_t` \*count)  
*Gets the number of bytes sent out.*
- `status_t FLEXIO_UART_TransferGetReceiveCountDMA` (`FLEXIO_UART_Type` \*base, `flexio_uart_dma_handle_t` \*handle, `size_t` \*count)  
*Gets the number of bytes received.*

## 12.7.9.2 Data Structure Documentation

### 12.7.9.2.1 struct \_flexio\_uart\_dma\_handle

#### Data Fields

- `flexio_uart_dma_transfer_callback_t callback`  
*Callback function.*
- `void *userData`  
*UART callback function parameter.*
- `size_t txDataSizeAll`  
*Total bytes to be sent.*
- `size_t rxDataSizeAll`  
*Total bytes to be received.*
- `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.*

#### 12.7.9.2.1.1 Field Documentation

12.7.9.2.1.1.1 `flexio_uart_dma_transfer_callback_t flexio_uart_dma_handle_t::callback`

12.7.9.2.1.1.2 `void* flexio_uart_dma_handle_t::userData`

12.7.9.2.1.1.3 `size_t flexio_uart_dma_handle_t::txDataSizeAll`

12.7.9.2.1.1.4 `size_t flexio_uart_dma_handle_t::rxDataSizeAll`

12.7.9.2.1.1.5 `dma_handle_t* flexio_uart_dma_handle_t::txDmaHandle`

12.7.9.2.1.1.6 `dma_handle_t* flexio_uart_dma_handle_t::rxDmaHandle`

12.7.9.2.1.1.7 `volatile uint8_t flexio_uart_dma_handle_t::txState`

#### 12.7.9.3 Typedef Documentation

12.7.9.3.1 `typedef void(* flexio_uart_dma_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_dma_handle_t *handle, status_t status, void *userData)`

#### 12.7.9.4 Function Documentation

12.7.9.4.1 `status_t FLEXIO_UART_TransferCreateHandleDMA ( FLEXIO_UART_Type * base, flexio_uart_dma_handle_t * handle, flexio_uart_dma_transfer_callback_t callback, void * userData, dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle )`

## FlexIO UART Driver

Parameters

|                    |                                                                |
|--------------------|----------------------------------------------------------------|
| <i>base</i>        | Pointer to <a href="#">FLEXIO_UART_Type</a> structure.         |
| <i>handle</i>      | Pointer to <a href="#">flexio_uart_dma_handle_t</a> structure. |
| <i>callback</i>    | FlexIO UART callback, NULL means no callback.                  |
| <i>userData</i>    | User callback function data.                                   |
| <i>txDmaHandle</i> | User requested DMA handle for TX DMA transfer.                 |
| <i>rxDmaHandle</i> | User requested DMA handle for RX DMA transfer.                 |

Return values

|                           |                                                     |
|---------------------------|-----------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                     |
| <i>kStatus_OutOfRange</i> | The FlexIO UART DMA type/handle table out of range. |

### 12.7.9.4.2 status\_t FLEXIO\_UART\_TransferSendDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_dma\\_handle\\_t](#) \* *handle*, [flexio\\_uart\\_transfer\\_t](#) \* *xfer* )

This function send data using DMA. This is non-blocking function, which returns right away. When all data is sent out, the send callback function is called.

Parameters

|               |                                                                                  |
|---------------|----------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a> structure                            |
| <i>handle</i> | Pointer to <a href="#">flexio_uart_dma_handle_t</a> structure                    |
| <i>xfer</i>   | FLEXIO_UART DMA transfer structure, see <a href="#">flexio_uart_transfer_t</a> . |

Return values

|                                   |                             |
|-----------------------------------|-----------------------------|
| <i>kStatus_Success</i>            | if succeed, others failed.  |
| <i>kStatus_FLEXIO_UART_TxBusy</i> | Previous transfer on going. |

### 12.7.9.4.3 status\_t FLEXIO\_UART\_TransferReceiveDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_dma\\_handle\\_t](#) \* *handle*, [flexio\\_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

|               |                                                                                  |
|---------------|----------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a> structure                            |
| <i>handle</i> | Pointer to <a href="#">flexio_uart_dma_handle_t</a> structure                    |
| <i>xfer</i>   | FLEXIO_UART DMA transfer structure, see <a href="#">flexio_uart_transfer_t</a> . |

Return values

|                                   |                             |
|-----------------------------------|-----------------------------|
| <i>kStatus_Success</i>            | if succeed, others failed.  |
| <i>kStatus_FLEXIO_UART_RxBusy</i> | Previous transfer on going. |

#### **12.7.9.4.4 void FLEXIO\_UART\_TransferAbortSendDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_dma\\_handle\\_t](#) \* *handle* )**

This function aborts the sent data which using DMA.

Parameters

|               |                                                               |
|---------------|---------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a> structure         |
| <i>handle</i> | Pointer to <a href="#">flexio_uart_dma_handle_t</a> structure |

#### **12.7.9.4.5 void FLEXIO\_UART\_TransferAbortReceiveDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_dma\\_handle\\_t](#) \* *handle* )**

This function aborts the receive data which using DMA.

Parameters

|               |                                                               |
|---------------|---------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a> structure         |
| <i>handle</i> | Pointer to <a href="#">flexio_uart_dma_handle_t</a> structure |

#### **12.7.9.4.6 status\_t FLEXIO\_UART\_TransferGetSendCountDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_dma\\_handle\\_t](#) \* *handle*, [size\\_t](#) \* *count* )**

This function gets the number of bytes sent out.

## FlexIO UART Driver

Parameters

|               |                                                              |
|---------------|--------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a> structure        |
| <i>handle</i> | Pointer to <code>flexio_uart_dma_handle_t</code> structure   |
| <i>count</i>  | Number of bytes sent so far by the non-blocking transaction. |

Return values

|                                      |                                                   |
|--------------------------------------|---------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>               | Successfully return the count.                    |

### **12.7.9.4.7 `status_t FLEXIO_UART_TransferGetReceiveCountDMA ( FLEXIO_UART_Type * base, flexio_uart_dma_handle_t * handle, size_t * count )`**

This function gets the number of bytes received.

Parameters

|               |                                                                  |
|---------------|------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a> structure            |
| <i>handle</i> | Pointer to <code>flexio_uart_dma_handle_t</code> structure       |
| <i>count</i>  | Number of bytes received so far by the non-blocking transaction. |

Return values

|                                      |                                                   |
|--------------------------------------|---------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>               | Successfully return the count.                    |

# Chapter 13

## GPIO: General-Purpose Input/Output Driver

### 13.1 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.*

### 13.2 Data Structure Documentation

#### 13.2.1 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

### 13.3 Macro Definition Documentation

13.3.1 `#define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))`

### 13.4 Enumeration Type Documentation

#### 13.4.1 `enum gpio_pin_direction_t`

Enumerator

*kGPIO\_DigitalInput* Set current pin as digital input.

*kGPIO\_DigitalOutput* Set current pin as digital output.

## 13.5 GPIO Driver

### 13.5.1 Overview

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

### 13.5.2 Typical use case

#### 13.5.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
 kGpioDigitalOutput,
 1,
};

/* Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

#### 13.5.2.2 Input Operation

```
/* Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_GPIO_PIN,
 kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
 kGpioDigitalInput,
 0,
};

/* Sets the input pin configuration */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```

## GPIO Configuration

- void **GPIO\_PinInit** (GPIO\_Type \*base, uint32\_t pin, const **gpio\_pin\_config\_t** \*config)  
*Initializes a GPIO pin used by the board.*

## GPIO Output Operations

- static void **GPIO\_WritePinOutput** (GPIO\_Type \*base, uint32\_t pin, uint8\_t output)  
*Sets the output level of the multiple GPIO pins to the logic 1 or 0.*
- static void **GPIO\_SetPinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 1.*
- static void **GPIO\_ClearPinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 0.*
- static void **GPIO\_TogglePinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Reverses the current output logic of the multiple GPIO pins.*

### GPIO Input Operations

- static uint32\_t [GPIO\\_ReadPinInput](#) (GPIO\_Type \*base, uint32\_t pin)  
*Reads the current input value of the GPIO port.*

### GPIO Interrupt

- uint32\_t [GPIO\\_GetPinsInterruptFlags](#) (GPIO\_Type \*base)  
*Reads the GPIO port interrupt status flag.*
- void [GPIO\\_ClearPinsInterruptFlags](#) (GPIO\_Type \*base, uint32\_t mask)  
*Clears multiple GPIO pin interrupt status flags.*

### 13.5.3 Function Documentation

#### 13.5.3.1 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                                 |

#### 13.5.3.2 static void [GPIO\\_WritePinOutput](#) ( **GPIO\_Type** \* *base*, **uint32\_t** *pin*, **uint8\_t** *output* ) [inline], [static]

Parameters

|               |                                                                                                                                                                                        |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)                                                                                                                         |
| <i>pin</i>    | GPIO pin number                                                                                                                                                                        |
| <i>output</i> | GPIO pin output logic level. <ul style="list-style-type: none"> <li>• 0: corresponding pin output low-logic level.</li> <li>• 1: corresponding pin output high-logic level.</li> </ul> |

### 13.5.3.3 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                                          |

### 13.5.3.4 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                                          |

### 13.5.3.5 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                                          |

### 13.5.3.6 static **uint32\_t** GPIO\_ReadPinInput ( **GPIO\_Type** \* *base*, **uint32\_t** *pin* ) [**inline**], [**static**]

## GPIO Driver

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>pin</i>  | GPIO pin number                                                |

Return values

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

### 13.5.3.7 `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. |
|------------|-------------------------------------------------------------------------------------------------------------|

### 13.5.3.8 `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                                          |

## 13.6 FGPIO Driver

### 13.6.1 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.

### 13.6.2 Typical use case

#### 13.6.2.1 Output Operation

```
/* Output pin configuration */
gpio_pin_config_t led_config =
{
 kGpioDigitalOutput,
 1,
};

/* Sets the configuration */
GPIO_PinInit(GPIO_LED, LED_PINNUM, &led_config);
```

#### 13.6.2.2 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 */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```

## FGPIO Configuration

- void **GPIO\_PinInit** (GPIO\_Type \*base, uint32\_t pin, const gpio\_pin\_config\_t \*config)  
*Initializes a FGPIO pin used by the board.*

## FGPIO Output Operations

- static void **GPIO\_WritePinOutput** (GPIO\_Type \*base, uint32\_t pin, uint8\_t output)  
*Sets the output level of the multiple FGPIO pins to the logic 1 or 0.*

## GPIO Driver

- static void **GPIO\_SetPinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 1.*
- static void **GPIO\_ClearPinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 0.*
- static void **GPIO\_TogglePinsOutput** (GPIO\_Type \*base, uint32\_t mask)  
*Reverses the current output logic of the multiple GPIO pins.*

## GPIO Input Operations

- static uint32\_t **GPIO\_ReadPinInput** (GPIO\_Type \*base, uint32\_t pin)  
*Reads the current input value of the GPIO port.*

## GPIO Interrupt

- uint32\_t **GPIO\_GetPinsInterruptFlags** (GPIO\_Type \*base)  
*Reads the GPIO port interrupt status flag.*
- void **GPIO\_ClearPinsInterruptFlags** (GPIO\_Type \*base, uint32\_t mask)  
*Clears the multiple GPIO pin interrupt status flag.*

### 13.6.3 Function Documentation

#### 13.6.3.1 void **GPIO\_PinInit** ( **GPIO\_Type** \* *base*, **uint32\_t** *pin*, **const gpio\_pin\_config\_t** \* *config* )

To initialize the GPIO driver, 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                                 |

### 13.6.3.2 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> | GPIOpin output logic level. <ul style="list-style-type: none"><li>• 0: corresponding pin output low-logic level.</li><li>• 1: corresponding pin output high-logic level.</li></ul> |

### 13.6.3.3 static void GPIO\_Set PinsOutput ( **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                                          |

### 13.6.3.4 static void GPIO\_Clear PinsOutput ( **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                                          |

### 13.6.3.5 static void GPIO\_Toggle PinsOutput ( **GPIO\_Type** \* *base*, **uint32\_t** *mask* ) [inline], [static]

## GPIO Driver

Parameters

|             |                                                                |
|-------------|----------------------------------------------------------------|
| <i>base</i> | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <i>mask</i> | GPIO pin number macro                                          |

### 13.6.3.6 static uint32\_t GPIO\_ReadPinInput ( **GPIO\_Type** \* *base*, **uint32\_t** *pin* ) [**inline**], [**static**]

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<br>• 0: corresponding pin input low-logic level.<br>• 1: corresponding pin input high-logic level. |
|-------------|---------------------------------------------------------------------------------------------------------------------|

### 13.6.3.7 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 flags, for example, 0x00010001 means the pin 0 and 17 have the interrupt. |
|------------|--------------------------------------------------------------------------------------------------------------|

### 13.6.3.8 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                                          |



# **Chapter 14**

## **I2C: Inter-Integrated Circuit Driver**

### **14.1 Overview**

#### **Modules**

- I2C DMA Driver
- I2C Driver
- I2C FreeRTOS Driver
- I2C eDMA Driver
- I2C μCOS/II Driver
- I2C μCOS/III Driver

### 14.2 I2C Driver

#### 14.2.1 Overview

The KSDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of Kinetis 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.

#### 14.2.2 Typical use case

##### 14.2.2.1 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);

if(result)
{
 /* If an error occurs, send STOP. */
```

```

 I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);
 return result;
}

while (!(I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR) & kI2C_IntPendingFlag))
{
}

/* Wait for all data to be sent out and sends STOP. */
I2C_MasterStop(EXAMPLE_I2C_MASTER_BASEADDR, kI2CStop);

```

#### 14.2.2.2 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;

```

#### 14.2.2.3 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;

```

## I2C Driver

```
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)
 {
 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;
```

### 14.2.2.4 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);

/* 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);
}
else
{
```

```

 I2C_SlaveReadBlocking(EXAMPLE_I2C_SLAVE_BASEADDR);
}

return result;
}

14.2.2.5 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_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;

```

## I2C Driver

### 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 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_SlaveGenaralcallEvent` = 0x40U,
   
  `kI2C_SlaveAllEvents` }
   
*Set of events sent to the callback for nonblocking slave transfers.*

## Driver version

- #define `FSL_I2C_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 2))
   
*I2C driver version 2.0.2.*

## 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)

## I2C Driver

- **void I2C\_MasterGetDefaultConfig (i2c\_master\_config\_t \*masterConfig)**  
*De-initializes the I2C slave peripheral.*  
*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.*

## 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.*

### 14.2.3 Data Structure Documentation

#### 14.2.3.1 struct i2c\_master\_config\_t

##### Data Fields

- bool **enableMaster**  
*Enables the I2C peripheral at initialization time.*
- bool **enableHighDrive**  
*Controls the drive capability of the I2C pads.*

## I2C Driver

- 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.*

### 14.2.3.1.0.1 Field Documentation

#### 14.2.3.1.0.1.1 bool i2c\_master\_config\_t::enableMaster

#### 14.2.3.1.0.1.2 bool i2c\_master\_config\_t::enableHighDrive

#### 14.2.3.1.0.1.3 bool i2c\_master\_config\_t::enableStopHold

#### 14.2.3.1.0.1.4 uint32\_t i2c\_master\_config\_t::baudRate\_Bps

#### 14.2.3.1.0.1.5 uint8\_t i2c\_master\_config\_t::glitchFilterWidth

### 14.2.3.2 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 `enableHighDrive`  
*Controls the drive capability of the I2C pads.*
- 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.*

#### 14.2.3.2.0.2 Field Documentation

- 14.2.3.2.0.2.1 `bool i2c_slave_config_t::enableSlave`
- 14.2.3.2.0.2.2 `bool i2c_slave_config_t::enableGeneralCall`
- 14.2.3.2.0.2.3 `bool i2c_slave_config_t::enableWakeUp`
- 14.2.3.2.0.2.4 `bool i2c_slave_config_t::enableHighDrive`
- 14.2.3.2.0.2.5 `bool i2c_slave_config_t::enableBaudRateCtl`
- 14.2.3.2.0.2.6 `uint16_t i2c_slave_config_t::slaveAddress`
- 14.2.3.2.0.2.7 `uint16_t i2c_slave_config_t::upperAddress`
- 14.2.3.2.0.2.8 `i2c_slave_address_mode_t i2c_slave_config_t::addressingMode`
- 14.2.3.2.0.2.9 `uint32_t i2c_slave_config_t::sclStopHoldTime_ns`

#### 14.2.3.3 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.*

#### 14.2.3.3.0.3 Field Documentation

- 14.2.3.3.0.3.1 `uint32_t i2c_master_transfer_t::flags`
- 14.2.3.3.0.3.2 `uint8_t i2c_master_transfer_t::slaveAddress`
- 14.2.3.3.0.3.3 `i2c_direction_t i2c_master_transfer_t::direction`
- 14.2.3.3.0.3.4 `uint32_t i2c_master_transfer_t::subaddress`

Transferred MSB first.

## I2C Driver

**14.2.3.3.0.3.5 uint8\_t i2c\_master\_transfer\_t::subaddressSize**

**14.2.3.3.0.3.6 uint8\_t\* volatile i2c\_master\_transfer\_t::data**

**14.2.3.3.0.3.7 volatile size\_t i2c\_master\_transfer\_t::dataSize**

### 14.2.3.4 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.*

#### 14.2.3.4.0.4 Field Documentation

**14.2.3.4.0.4.1 i2c\_master\_transfer\_t i2c\_master\_handle\_t::transfer**

**14.2.3.4.0.4.2 size\_t i2c\_master\_handle\_t::transferSize**

**14.2.3.4.0.4.3 uint8\_t i2c\_master\_handle\_t::state**

**14.2.3.4.0.4.4 i2c\_master\_transfer\_callback\_t i2c\_master\_handle\_t::completionCallback**

**14.2.3.4.0.4.5 void\* i2c\_master\_handle\_t::userData**

### 14.2.3.5 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.*

#### 14.2.3.5.0.5 Field Documentation

14.2.3.5.0.5.1 `i2c_slave_transfer_event_t i2c_slave_transfer_t::event`

14.2.3.5.0.5.2 `uint8_t* volatile i2c_slave_transfer_t::data`

14.2.3.5.0.5.3 `volatile size_t i2c_slave_transfer_t::dataSize`

14.2.3.5.0.5.4 `status_t i2c_slave_transfer_t::completionStatus`

Only applies for [kI2C\\_SlaveCompletionEvent](#).

14.2.3.5.0.5.5 `size_t i2c_slave_transfer_t::transferredCount`

#### 14.2.3.6 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

### 14.2.3.6.0.6 Field Documentation

14.2.3.6.0.6.1 `volatile bool i2c_slave_handle_t::isBusy`

14.2.3.6.0.6.2 `i2c_slave_transfer_t i2c_slave_handle_t::transfer`

14.2.3.6.0.6.3 `uint32_t i2c_slave_handle_t::eventMask`

14.2.3.6.0.6.4 `i2c_slave_transfer_callback_t i2c_slave_handle_t::callback`

14.2.3.6.0.6.5 `void* i2c_slave_handle_t::userData`

### 14.2.4 Macro Definition Documentation

14.2.4.1 `#define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))`

### 14.2.5 Typedef Documentation

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

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

### 14.2.6 Enumeration Type Documentation

#### 14.2.6.1 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.

#### 14.2.6.2 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.

**14.2.6.3 enum \_i2c\_interrupt\_enable**

## Enumerator

- kI2C\_GlobalInterruptEnable* I2C global interrupt.
- kI2C\_StartStopDetectInterruptEnable* I2C start&stop detect interrupt.

**14.2.6.4 enum i2c\_direction\_t**

## Enumerator

- kI2C\_Write* Master transmits to the slave.
- kI2C\_Read* Master receives from the slave.

**14.2.6.5 enum i2c\_slave\_address\_mode\_t**

## Enumerator

- kI2C\_Address7bit* 7-bit addressing mode.
- kI2C\_RangeMatch* Range address match addressing mode.

**14.2.6.6 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.

### 14.2.6.7 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.

### 14.2.7 Function Documentation

#### 14.2.7.1 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            |

#### 14.2.7.2 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           |

#### 14.2.7.3 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 |
|-------------|------------------|

### 14.2.7.4 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 |
|-------------|------------------|

### 14.2.7.5 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. |
|---------------------|--------------------------------------------------|

### 14.2.7.6 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. |
|--------------------|-------------------------------------------------|

#### 14.2.7.7 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. |

#### 14.2.7.8 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.

#### 14.2.7.9 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.

#### 14.2.7.10 static void I2C\_MasterClearStatusFlags ( I2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared kI2C\_ArbitrationLostFlag and kI2C\_IntPendingFlag.

## I2C Driver

Parameters

|                   |                                                                                                                                                                                                                                                                                                                                |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | I2C base pointer                                                                                                                                                                                                                                                                                                               |
| <i>statusMask</i> | The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values: <ul style="list-style-type: none"><li>• kI2C_StartDetectFlag (if available)</li><li>• kI2C_StopDetectFlag (if available)</li><li>• kI2C_ArbitrationLostFlag</li><li>• kI2C_IntPendingFlagFlag</li></ul> |

### 14.2.7.11 static void I2C\_SlaveClearStatusFlags ( I2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared kI2C\_ArbitrationLostFlag and kI2C\_IntPendingFlag

Parameters

|                   |                                                                                                                                                                                                                                                                                                                                |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | I2C base pointer                                                                                                                                                                                                                                                                                                               |
| <i>statusMask</i> | The status flag mask, defined in type i2c_status_flag_t. The parameter can be any combination of the following values: <ul style="list-style-type: none"><li>• kI2C_StartDetectFlag (if available)</li><li>• kI2C_StopDetectFlag (if available)</li><li>• kI2C_ArbitrationLostFlag</li><li>• kI2C_IntPendingFlagFlag</li></ul> |

### 14.2.7.12 void I2C\_EnableInterrupts ( I2C\_Type \* *base*, uint32\_t *mask* )

Parameters

|             |                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | I2C base pointer                                                                                                                                                                                                                                                                     |
| <i>mask</i> | interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"><li>• kI2C_GlobalInterruptEnable</li><li>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</li><li>• kI2C_SdaTimeoutInterruptEnable</li></ul> |

### 14.2.7.13 void I2C\_DisableInterrupts ( I2C\_Type \* *base*, uint32\_t *mask* )

Parameters

|             |                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | I2C base pointer                                                                                                                                                                                                                                                                     |
| <i>mask</i> | interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"><li>• kI2C_GlobalInterruptEnable</li><li>• kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</li><li>• kI2C_SdaTimeoutInterruptEnable</li></ul> |

#### 14.2.7.14 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 |

#### 14.2.7.15 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

#### 14.2.7.16 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               |

### 14.2.7.17 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.                |

#### 14.2.7.18 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.  |

#### 14.2.7.19 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. |

#### 14.2.7.20 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_ArbitrationLost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>             | Transfer error, receive NAK during transfer. |

### 14.2.7.21 **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.            |

### 14.2.7.22 **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_ArbitrationLost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>             | Transfer error, receive NAK during transfer. |

#### 14.2.7.23 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.     |

#### 14.2.7.24 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_ArbitrationLost</i> | Transfer error, arbitration lost.            |
| <i>kStatus_I2C_Nak</i>             | Transfer error, receive NAK during transfer. |

**14.2.7.25 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 <i>i2c_master_handle_t</i> structure to store the transfer state. |
| <i>callback</i> | pointer to user callback function.                                           |
| <i>userData</i> | user parameter passed to the callback function.                              |

**14.2.7.26 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 <i>i2c_master_handle_t</i> structure which stores the transfer state. |
| <i>xfer</i>   | pointer to <i>i2c_master_transfer_t</i> 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.      |

14.2.7.27 **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. |

### 14.2.7.28 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 |

### 14.2.7.29 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. |

### 14.2.7.30 void I2C\_SlaveTransferCreateHandle ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, i2c\_slave\_transfer\_callback\_t *callback*, void \* *userData* )

Parameters

|                 |                                                                      |
|-----------------|----------------------------------------------------------------------|
| <i>base</i>     | I2C base pointer.                                                    |
| <i>handle</i>   | pointer to i2c_slave_handle_t structure to store the transfer state. |
| <i>callback</i> | pointer to user callback function.                                   |
| <i>userData</i> | user parameter passed to the callback function.                      |

#### 14.2.7.31 **status\_t I2C\_SlaveTransferNonBlocking ( I2C\_Type \* *base*, i2c\_slave\_handle\_t \* *handle*, uint32\_t *eventMask* )**

Call this API after calling the [I2C\\_SlaveInit\(\)](#) and [I2C\\_SlaveTransferCreateHandle\(\)](#) to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to [I2C\\_SlaveTransferCreateHandle\(\)](#). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of [i2c\\_slave\\_transfer\\_event\\_t](#) enumerators for the events you wish to receive. The [k\\_I2C\\_SlaveTransmitEvent](#) and #[kLPI2C\\_SlaveReceiveEvent](#) events are always enabled and do not need to 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 # <a href="#">i2c_slave_handle_t</a> structure which stores the transfer state.                                                                                                                                                                                                         |
| <i>eventMask</i> | Bit mask formed by OR'ing together <a href="#">i2c_slave_transfer_event_t</a> enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and <a href="#">kI2C_SlaveAllEvents</a> to enable all events. |

Return values

|                                  |                                                           |
|----------------------------------|-----------------------------------------------------------|
| <a href="#">#kStatus_Success</a> | Slave transfers were successfully started.                |
| <a href="#">kStatus_I2C_Busy</a> | Slave transfers have already been started on this handle. |

#### 14.2.7.32 **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. |

### 14.2.7.33 **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. |

### 14.2.7.34 **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 |

## 14.3 I2C eDMA Driver

### 14.3.1 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.*

### 14.3.2 Data Structure Documentation

#### 14.3.2.1 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.*

### 14.3.2.1.0.7 Field Documentation

**14.3.2.1.0.7.1 `i2c_master_transfer_t i2c_master_edma_handle_t::transfer`**

**14.3.2.1.0.7.2 `size_t i2c_master_edma_handle_t::transferSize`**

**14.3.2.1.0.7.3 `uint8_t i2c_master_edma_handle_t::nbytes`**

**14.3.2.1.0.7.4 `uint8_t i2c_master_edma_handle_t::state`**

**14.3.2.1.0.7.5 `edma_handle_t* i2c_master_edma_handle_t::dmaHandle`**

**14.3.2.1.0.7.6 `i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completionCallback`**

**14.3.2.1.0.7.7 `void* i2c_master_edma_handle_t::userData`**

### 14.3.3 Typedef Documentation

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

### 14.3.4 Function Documentation

**14.3.4.1 `void I2C_MasterCreateEDMAHandle ( I2C_Type * base, i2c_master_edma_handle_t * handle, i2c_master_edma_transfer_callback_t callback, void * userData, edma_handle_t * edmaHandle )`**

Parameters

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

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

**14.3.4.2 status\_t I2C\_MasterTransferEDMA ( *I2C\_Type* \* *base*, *i2c\_master\_edma\_handle\_t* \* *handle*, *i2c\_master\_transfer\_t* \* *xfer* )**

Parameters

|               |                                                                       |
|---------------|-----------------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address.                                          |
| <i>handle</i> | A pointer to the <i>i2c_master_edma_handle_t</i> structure.           |
| <i>xfer</i>   | A pointer to the transfer structure of <i>i2c_master_transfer_t</i> . |

Return values

|                                     |                                                |
|-------------------------------------|------------------------------------------------|
| <i>kStatus_Success</i>              | Sucessfully completed the data transmission.   |
| <i>kStatus_I2C_Busy</i>             | A previous transmission is still not finished. |
| <i>kStatus_I2C_Timeout</i>          | Transfer error, waits for a signal timeout.    |
| <i>kStatus_I2C_Arbitration_Lost</i> | Transfer error, arbitration lost.              |
| <i>kStatus_I2C_Nak</i>              | Transfer error, receive NAK during transfer.   |

**14.3.4.3 status\_t I2C\_MasterTransferGetCountEDMA ( *I2C\_Type* \* *base*, *i2c\_master\_edma\_handle\_t* \* *handle*, *size\_t* \* *count* )**

Parameters

|               |                                                                |
|---------------|----------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address.                                   |
| <i>handle</i> | A pointer to the <i>i2c_master_edma_handle_t</i> structure.    |
| <i>count</i>  | A number of bytes transferred by the non-blocking transaction. |

**14.3.4.4 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 i2c_master_edma_handle_t structure. |

## 14.4 I2C DMA Driver

### 14.4.1 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.*

### 14.4.2 Data Structure Documentation

#### 14.4.2.1 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.*

### 14.4.2.1.0.8 Field Documentation

**14.4.2.1.0.8.1 i2c\_master\_transfer\_t i2c\_master\_dma\_handle\_t::transfer**

**14.4.2.1.0.8.2 size\_t i2c\_master\_dma\_handle\_t::transferSize**

**14.4.2.1.0.8.3 uint8\_t i2c\_master\_dma\_handle\_t::state**

**14.4.2.1.0.8.4 dma\_handle\_t\* i2c\_master\_dma\_handle\_t::dmaHandle**

**14.4.2.1.0.8.5 i2c\_master\_dma\_transfer\_callback\_t i2c\_master\_dma\_handle\_t::completionCallback**

**14.4.2.1.0.8.6 void\* i2c\_master\_dma\_handle\_t::userData**

### 14.4.3 Typedef Documentation

**14.4.3.1 typedef void(\* i2c\_master\_dma\_transfer\_callback\_t)(I2C\_Type \*base, i2c\_master\_dma\_handle\_t \*handle, status\_t status, void \*userData)**

### 14.4.4 Function Documentation

**14.4.4.1 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                               |

**14.4.4.2 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 i2c_master_dma_handle_t structure               |
| <i>xfer</i>   | A pointer to the transfer structure of the i2c_master_transfer_t |

Return values

|                                     |                                                 |
|-------------------------------------|-------------------------------------------------|
| <i>kStatus_Success</i>              | Sucessfully completes the data transmission.    |
| <i>kStatus_I2C_Busy</i>             | A previous transmission is still not finished.  |
| <i>kStatus_I2C_Timeout</i>          | A transfer error, waits for the signal timeout. |
| <i>kStatus_I2C_Arbitration-Lost</i> | A transfer error, arbitration lost.             |
| <i>kStatus_I2C_Nak</i>              | A transfer error, receives NAK during transfer. |

#### 14.4.4.3 status\_t I2C\_MasterTransferGetCountDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

|               |                                                                       |
|---------------|-----------------------------------------------------------------------|
| <i>base</i>   | I2C peripheral base address                                           |
| <i>handle</i> | A pointer to the i2c_master_dma_handle_t structure                    |
| <i>count</i>  | A number of bytes transferred so far by the non-blocking transaction. |

#### 14.4.4.4 void I2C\_MasterTransferAbortDMA ( I2C\_Type \* *base*, i2c\_master\_dma\_handle\_t \* *handle* )

Parameters

|               |                                                     |
|---------------|-----------------------------------------------------|
| <i>base</i>   | I2C peripheral base address                         |
| <i>handle</i> | A pointer to the i2c_master_dma_handle_t structure. |

## I2C FreeRTOS Driver

### 14.5 I2C FreeRTOS Driver

#### 14.5.1 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.*

#### 14.5.2 Function Documentation

##### 14.5.2.1 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.

##### 14.5.2.2 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. |
|---------------|----------------------|

#### 14.5.2.3 **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.

### 14.6 I2C μCOS/II Driver

#### 14.6.1 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 the 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.*

#### 14.6.2 Function Documentation

##### 14.6.2.1 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> | A configuration structure to set-up the I2C in master mode.              |
| <i>srcClock_Hz</i>  | A frequency of the input clock of the I2C module.                        |

Returns

status of the operation.

##### 14.6.2.2 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. |
|---------------|----------------------|

#### 14.6.2.3 **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.

### 14.7 I2C μCOS/III Driver

#### 14.7.1 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 the 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.*

#### 14.7.2 Function Documentation

##### 14.7.2.1 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 the RTOS context. |
| <i>base</i>         | The pointer base address of the I2C instance to initialize.                  |
| <i>masterConfig</i> | A configuration structure to set-up the I2C in master mode.                  |
| <i>srcClock_Hz</i>  | A frequency of the input clock of the I2C module.                            |

Returns

status of the operation.

##### 14.7.2.2 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. |
|---------------|----------------------|

#### 14.7.2.3 **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 15

## LLWU: Low-Leakage Wakeup Unit Driver

### 15.1 Overview

The KSDK provides a peripheral driver for the Low-Leakage Wakeup Unit (LLWU) module of Kinetis 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.

### 15.2 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.

### 15.3 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.

### 15.4 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\_EnableInternalModuleInterruptWakeup** (LLWU\_Type \*base, uint32\_t moduleIndex, bool enable)  
*Enables/disables the internal module source.*
- static bool **LLWU\_GetInternalWakeUpModuleFlag** (LLWU\_Type \*base, uint32\_t moduleIndex)  
*Gets the external wakeup source flag.*
- void **LLWU\_SetPinFilterMode** (LLWU\_Type \*base, uint32\_t filterIndex, **llwu\_external\_pin\_filter\_mode\_t** filterMode)  
*Sets the pin filter configuration.*
- bool **LLWU\_GetPinFilterFlag** (LLWU\_Type \*base, uint32\_t filterIndex)  
*Gets the pin filter configuration.*
- void **LLWU\_ClearPinFilterFlag** (LLWU\_Type \*base, uint32\_t filterIndex)  
*Clears the pin filter configuration.*

## 15.5 Data Structure Documentation

### 15.5.1 struct **llwu\_external\_pin\_filter\_mode\_t**

#### Data Fields

- uint32\_t **pinIndex**  
*A pin number.*
- **llwu\_pin\_filter\_mode\_t** **filterMode**  
*Filter mode.*

## 15.6 Macro Definition Documentation

### 15.6.1 #define **FSL\_LLWU\_DRIVER\_VERSION** (MAKE\_VERSION(2, 0, 1))

## 15.7 Enumeration Type Documentation

### 15.7.1 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.

## 15.7.2 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.

## 15.8 Function Documentation

### 15.8.1 void LLWU\_SetExternalWakeupsPinMode ( *LLWU\_Type \* base*, *uint32\_t pinIndex*, *llwu\_external\_pin\_mode\_t pinMode* )

This function sets the external input pin source mode that is used as a wake up source.

Parameters

|                 |                                                                         |
|-----------------|-------------------------------------------------------------------------|
| <i>base</i>     | LLWU peripheral base address.                                           |
| <i>pinIndex</i> | A pin index to be enabled as an external wakeup source starting from 1. |
| <i>pinMode</i>  | A pin configuration mode defined in the llwu_external_pin_modes_t.      |

### 15.8.2 bool LLWU\_GetExternalWakeupsPinFlag ( *LLWU\_Type \* base*, *uint32\_t pinIndex* )

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

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.

## Function Documentation

**15.8.3 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. |

#### 15.8.4 static void LLWU\_EnableInternalModuleInterruptWakup ( **LLWU\_Type** \* *base*, **uint32\_t** *moduleIndex*, **bool** *enable* ) [inline], [static]

This function enables/disables the internal module source mode that is used as a wake up source.

Parameters

|                    |                                                                            |
|--------------------|----------------------------------------------------------------------------|
| <i>base</i>        | LLWU peripheral base address.                                              |
| <i>moduleIndex</i> | A module index to be enabled as an internal wakeup source starting from 1. |
| <i>enable</i>      | An enable or a disable setting                                             |

#### 15.8.5 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.

#### 15.8.6 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                                                    |

### **15.8.7 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.

### **15.8.8 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 16

## LPTMR: Low-Power Timer

### 16.1 Overview

The KSDK provides a driver for the Low-Power Timer (LPTMR) of Kinetis 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 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 }
   
*List of the LPTMR interrupts.*
- enum `lptmr_status_flags_t` { `kLPTMR_TimerCompareFlag` = LPTMR\_CSR\_TCF\_MASK }
   
*List of the LPTMR status flags.*

## Driver version

- #define `FSL_LPTMR_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 0))

Version 2.0.0.

### 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, uint16\_t ticks)  
*Sets the timer period in units of count.*
- static uint16\_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.

## Enumeration Type Documentation

### 16.5.4 enum lptmr\_prescaler\_glitch\_value\_t

Enumerator

- kLPTMR\_Prescale\_Glitch\_0*** Prescaler divide 2, glitch filter does not support this setting.
- kLPTMR\_Prescale\_Glitch\_1*** Prescaler divide 4, glitch filter 2.
- kLPTMR\_Prescale\_Glitch\_2*** Prescaler divide 8, glitch filter 4.
- kLPTMR\_Prescale\_Glitch\_3*** Prescaler divide 16, glitch filter 8.
- kLPTMR\_Prescale\_Glitch\_4*** Prescaler divide 32, glitch filter 16.
- kLPTMR\_Prescale\_Glitch\_5*** Prescaler divide 64, glitch filter 32.
- kLPTMR\_Prescale\_Glitch\_6*** Prescaler divide 128, glitch filter 64.
- kLPTMR\_Prescale\_Glitch\_7*** Prescaler divide 256, glitch filter 128.
- kLPTMR\_Prescale\_Glitch\_8*** Prescaler divide 512, glitch filter 256.
- kLPTMR\_Prescale\_Glitch\_9*** Prescaler divide 1024, glitch filter 512.
- kLPTMR\_Prescale\_Glitch\_10*** Prescaler divide 2048 glitch filter 1024.
- kLPTMR\_Prescale\_Glitch\_11*** Prescaler divide 4096, glitch filter 2048.
- kLPTMR\_Prescale\_Glitch\_12*** Prescaler divide 8192, glitch filter 4096.
- kLPTMR\_Prescale\_Glitch\_13*** Prescaler divide 16384, glitch filter 8192.
- kLPTMR\_Prescale\_Glitch\_14*** Prescaler divide 32768, glitch filter 16384.
- kLPTMR\_Prescale\_Glitch\_15*** Prescaler divide 65536, glitch filter 32768.

### 16.5.5 enum lptmr\_prescaler\_clock\_select\_t

Note

Clock connections are SoC-specific

Enumerator

- kLPTMR\_PrescalerClock\_0*** Prescaler/glitch filter clock 0 selected.
- kLPTMR\_PrescalerClock\_1*** Prescaler/glitch filter clock 1 selected.
- kLPTMR\_PrescalerClock\_2*** Prescaler/glitch filter clock 2 selected.
- kLPTMR\_PrescalerClock\_3*** Prescaler/glitch filter clock 3 selected.

### 16.5.6 enum lptmr\_interrupt\_enable\_t

Enumerator

- kLPTMR\_TimerInterruptEnable*** 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 <a href="#">lptmr_interrupt_enable_t</a> |

### 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 <a href="#">lptmr_interrupt_enable_t</a> . |

### 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 <a href="#">lptmr_status_flags_t</a> . |

#### 16.6.9 static void LPTMR\_SetTimerPeriod ( LPTMR\_Type \* *base*, uint16\_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 uint16\_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

## LPUART: Low Power UART Driver

### 17.1 Overview

#### Modules

- LPUART DMA Driver
- LPUART Driver
- LPUART FreeRTOS Driver
- LPUART eDMA Driver
- LPUART µCOS/II Driver
- LPUART µCOS/III Driver

### 17.2 LPUART Driver

#### 17.2.1 Overview

The KSDK provides a peripheral driver for the Low Power UART (LPUART) module of Kinetis devices.

#### 17.2.2 Typical use case

##### 17.2.2.1 LPUART Operation

```
uint8_t ch;
LPUART_GetDefaultConfig(&user_config);
user_config.baudRate = 115200U;
config.enableTx = true;
config.enableRx = true;

LPUART_Init(LPUART1, &user_config, 120000000U);

LPUART_WriteBlocking(LPUART1, txbuff, sizeof(txbuff) - 1);

while(1)
{
 LPUART_ReadBlocking(LPUART1, &ch, 1);
 LPUART_WriteBlocking(LPUART1, &ch, 1);
}
```

## Data Structures

- struct [lpuart\\_config\\_t](#)  
*LPUART configuration structure.* [More...](#)
- struct [lpuart\\_transfer\\_t](#)  
*LPUART transfer structure.* [More...](#)
- struct [lpuart\\_handle\\_t](#)  
*LPUART handle structure.* [More...](#)

## Typedefs

- typedef void(\* [lpuart\\_transfer\\_callback\\_t](#) )(LPUART\_Type \*base, lpuart\_handle\_t \*handle, status\_t status, void \*userData)  
*LPUART transfer callback function.*

## Enumerations

- enum `_lpuart_status` {
   
kStatus\_LPUART\_TxBusy = MAKE\_STATUS(kStatusGroup\_LPUART, 0),
   
kStatus\_LPUART\_RxBusy = MAKE\_STATUS(kStatusGroup\_LPUART, 1),
   
kStatus\_LPUART\_TxIdle = MAKE\_STATUS(kStatusGroup\_LPUART, 2),
   
kStatus\_LPUART\_RxIdle = MAKE\_STATUS(kStatusGroup\_LPUART, 3),
   
kStatus\_LPUART\_TxWatermarkTooLarge = MAKE\_STATUS(kStatusGroup\_LPUART, 4),
   
kStatus\_LPUART\_RxWatermarkTooLarge = MAKE\_STATUS(kStatusGroup\_LPUART, 5),
   
kStatus\_LPUART\_FlagCannotClearManually = MAKE\_STATUS(kStatusGroup\_LPUART, 6),
   
kStatus\_LPUART\_Error = MAKE\_STATUS(kStatusGroup\_LPUART, 7),
   
kStatus\_LPUART\_RxRingBufferOverrun,
   
kStatus\_LPUART\_RxHardwareOverrun = MAKE\_STATUS(kStatusGroup\_LPUART, 9),
   
kStatus\_LPUART\_NoiseError = MAKE\_STATUS(kStatusGroup\_LPUART, 10),
   
kStatus\_LPUART\_FramingError = MAKE\_STATUS(kStatusGroup\_LPUART, 11),
   
kStatus\_LPUART\_ParityError = MAKE\_STATUS(kStatusGroup\_LPUART, 12),
   
kStatus\_LPUART\_BaudrateNotSupport }
   
*Error codes for the LPUART driver.*
- enum `lpuart_parity_mode_t` {
   
kLPUART\_ParityDisabled = 0x0U,
   
kLPUART\_ParityEven = 0x2U,
   
kLPUART\_ParityOdd = 0x3U }
   
*LPUART parity mode.*
- enum `lpuart_data_bits_t` { kLPUART\_EightDataBits = 0x0U }
   
*LPUART data bits count.*
- enum `lpuart_stop_bit_count_t` {
   
kLPUART\_OneStopBit = 0U,
   
kLPUART\_TwoStopBit = 1U }
   
*LPUART stop bit count.*
- enum `_lpuart_interrupt_enable` {
   
kLPUART\_LinBreakInterruptEnable = (LPUART\_BAUD\_LBKDIIE\_MASK >> 8),
   
kLPUART\_RxActiveEdgeInterruptEnable = (LPUART\_BAUD\_RXEDGIE\_MASK >> 8),
   
kLPUART\_TxDataRegEmptyInterruptEnable = (LPUART\_CTRL\_TIE\_MASK),
   
kLPUART\_TransmissionCompleteInterruptEnable = (LPUART\_CTRL\_TCIE\_MASK),
   
kLPUART\_RxDataRegFullInterruptEnable = (LPUART\_CTRL\_RIE\_MASK),
   
kLPUART\_IdleLineInterruptEnable = (LPUART\_CTRL\_ILIE\_MASK),
   
kLPUART\_RxOverrunInterruptEnable = (LPUART\_CTRL\_ORIE\_MASK),
   
kLPUART\_NoiseErrorInterruptEnable = (LPUART\_CTRL\_NEIE\_MASK),
   
kLPUART\_FramingErrorInterruptEnable = (LPUART\_CTRL\_FEIE\_MASK),
   
kLPUART\_ParityErrorInterruptEnable = (LPUART\_CTRL\_PEIE\_MASK) }
   
*LPUART interrupt configuration structure, default settings all disabled.*
- enum `_lpuart_flags` {

## LPUART Driver

```
kLPUART_TxDataRegEmptyFlag,
kLPUART_TransmissionCompleteFlag,
kLPUART_RxDataRegFullFlag,
kLPUART_IdleLineFlag = (LPUART_STAT_IDLE_MASK),
kLPUART_RxOverrunFlag = (LPUART_STAT_OR_MASK),
kLPUART_NoiseErrorFlag = (LPUART_STAT_NF_MASK),
kLPUART_FramingErrorFlag,
kLPUART_ParityErrorFlag = (LPUART_STAT_PF_MASK),
kLPUART_LinBreakFlag = (LPUART_STAT_LBKDIF_MASK),
kLPUART_RxActiveEdgeFlag,
kLPUART_RxActiveFlag,
kLPUART_DataMatch1Flag = LPUART_STAT_MA1F_MASK,
kLPUART_DataMatch2Flag = LPUART_STAT_MA2F_MASK,
kLPUART_NoiseErrorInRxDataRegFlag,
kLPUART_ParityErrorInRxDataRegFlag }
```

*LPUART status flags.*

## Driver version

- #define **FSL\_LPUART\_DRIVER\_VERSION** (MAKE\_VERSION(2, 2, 3))  
*LPUART driver version 2.2.1.*

## Initialization and deinitialization

- status\_t **LPUART\_Init** (LPUART\_Type \*base, const lpuart\_config\_t \*config, uint32\_t srcClock\_Hz)  
*Initializes an LPUART instance with the user configuration structure and the peripheral clock.*
- void **LPUART\_Deinit** (LPUART\_Type \*base)  
*Deinitializes a LPUART instance.*
- void **LPUART\_GetDefaultConfig** (lpuart\_config\_t \*config)  
*Gets the default configuration structure.*
- status\_t **LPUART\_SetBaudRate** (LPUART\_Type \*base, uint32\_t baudRate\_Bps, uint32\_t srcClock\_Hz)  
*Sets the LPUART instance baudrate.*

## Status

- uint32\_t **LPUART\_GetStatusFlags** (LPUART\_Type \*base)  
*Gets LPUART status flags.*
- status\_t **LPUART\_ClearStatusFlags** (LPUART\_Type \*base, uint32\_t mask)  
*Clears status flags with a provided mask.*

## Interrupts

- void [LPUART\\_EnableInterrupts](#) (LPUART\_Type \*base, uint32\_t mask)  
*Enables LPUART interrupts according to a provided mask.*
- void [LPUART\\_DisableInterrupts](#) (LPUART\_Type \*base, uint32\_t mask)  
*Disables LPUART interrupts according to a provided mask.*
- uint32\_t [LPUART\\_GetEnabledInterrupts](#) (LPUART\_Type \*base)  
*Gets enabled LPUART interrupts.*
- static uint32\_t [LPUART\\_GetDataRegisterAddress](#) (LPUART\_Type \*base)  
*Gets the LPUART data register address.*
- static void [LPUART\\_EnableTxDMA](#) (LPUART\_Type \*base, bool enable)  
*Enables or disables the LPUART transmitter DMA request.*
- static void [LPUART\\_EnableRxDMA](#) (LPUART\_Type \*base, bool enable)  
*Enables or disables the LPUART receiver DMA.*

## Bus Operations

- static void [LPUART\\_EnableTx](#) (LPUART\_Type \*base, bool enable)  
*Enables or disables the LPUART transmitter.*
- static void [LPUART\\_EnableRx](#) (LPUART\_Type \*base, bool enable)  
*Enables or disables the LPUART receiver.*
- static void [LPUART\\_WriteByte](#) (LPUART\_Type \*base, uint8\_t data)  
*Writes to the transmitter register.*
- static uint8\_t [LPUART\\_ReadByte](#) (LPUART\_Type \*base)  
*Reads the receiver register.*
- void [LPUART\\_WriteBlocking](#) (LPUART\_Type \*base, const uint8\_t \*data, size\_t length)  
*Writes to the transmitter register using a blocking method.*
- status\_t [LPUART\\_ReadBlocking](#) (LPUART\_Type \*base, uint8\_t \*data, size\_t length)  
*Reads the receiver data register using a blocking method.*

## Transactional

- void [LPUART\\_TransferCreateHandle](#) (LPUART\_Type \*base, lpuart\_handle\_t \*handle, [lpuart\\_transfer\\_callback\\_t](#) callback, void \*userData)  
*Initializes the LPUART handle.*
- status\_t [LPUART\\_TransferSendNonBlocking](#) (LPUART\_Type \*base, lpuart\_handle\_t \*handle, [lpuart\\_transfer\\_t](#) \*xfer)  
*Transmits a buffer of data using the interrupt method.*
- void [LPUART\\_TransferStartRingBuffer](#) (LPUART\_Type \*base, lpuart\_handle\_t \*handle, uint8\_t \*ringBuffer, size\_t ringBufferSize)  
*Sets up the RX ring buffer.*
- void [LPUART\\_TransferStopRingBuffer](#) (LPUART\_Type \*base, lpuart\_handle\_t \*handle)  
*Aborts the background transfer and uninstalls the ring buffer.*
- void [LPUART\\_TransferAbortSend](#) (LPUART\_Type \*base, lpuart\_handle\_t \*handle)  
*Aborts the interrupt-driven data transmit.*
- status\_t [LPUART\\_TransferGetSendCount](#) (LPUART\_Type \*base, lpuart\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes that have been written to the LPUART transmitter register.*

## LPUART Driver

- status\_t **LPUART\_TransferReceiveNonBlocking** (LPUART\_Type \*base, lpuart\_handle\_t \*handle, lpuart\_transfer\_t \*xfer, size\_t \*receivedBytes)  
*Receives a buffer of data using the interrupt method.*
- void **LPUART\_TransferAbortReceive** (LPUART\_Type \*base, lpuart\_handle\_t \*handle)  
*Aborts the interrupt-driven data receiving.*
- status\_t **LPUART\_TransferGetReceiveCount** (LPUART\_Type \*base, lpuart\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes that have been received.*
- void **LPUART\_TransferHandleIRQ** (LPUART\_Type \*base, lpuart\_handle\_t \*handle)  
*LPUART IRQ handle function.*
- void **LPUART\_TransferHandleErrorIRQ** (LPUART\_Type \*base, lpuart\_handle\_t \*handle)  
*LPUART Error IRQ handle function.*

### 17.2.3 Data Structure Documentation

#### 17.2.3.1 struct lpuart\_config\_t

##### Data Fields

- uint32\_t **baudRate\_Bps**  
*LPUART baud rate.*
- **lpuart\_parity\_mode\_t parityMode**  
*Parity mode, disabled (default), even, odd.*
- **lpuart\_data\_bits\_t dataBitsCount**  
*Data bits count, eight (default), seven.*
- bool **isMsb**  
*Data bits order, LSB (default), MSB.*
- **lpuart\_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.*

#### 17.2.3.2 struct lpuart\_transfer\_t

##### Data Fields

- uint8\_t \* **data**  
*The buffer of data to be transfer.*
- size\_t **dataSize**  
*The byte count to be transfer.*

### 17.2.3.2.0.9 Field Documentation

**17.2.3.2.0.9.1 uint8\_t\* lpuart\_transfer\_t::data**

**17.2.3.2.0.9.2 size\_t lpuart\_transfer\_t::dataSize**

### 17.2.3.3 struct \_lpuart\_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.*
- **lpuart\_transfer\_callback\_t callback**  
*Callback function.*
- **void \* userData**  
*LPUART callback function parameter.*
- **volatile uint8\_t txState**  
*TX transfer state.*
- **volatile uint8\_t rxState**  
*RX transfer state.*

## LPUART Driver

### 17.2.3.3.0.10 Field Documentation

- 17.2.3.3.0.10.1 `uint8_t* volatile Ipuart_handle_t::txData`
- 17.2.3.3.0.10.2 `volatile size_t Ipuart_handle_t::txDataSize`
- 17.2.3.3.0.10.3 `size_t Ipuart_handle_t::txDataSizeAll`
- 17.2.3.3.0.10.4 `uint8_t* volatile Ipuart_handle_t::rxData`
- 17.2.3.3.0.10.5 `volatile size_t Ipuart_handle_t::rxDataSize`
- 17.2.3.3.0.10.6 `size_t Ipuart_handle_t::rxDataSizeAll`
- 17.2.3.3.0.10.7 `uint8_t* Ipuart_handle_t::rxRingBuffer`
- 17.2.3.3.0.10.8 `size_t Ipuart_handle_t::rxRingBufferSize`
- 17.2.3.3.0.10.9 `volatile uint16_t Ipuart_handle_t::rxRingBufferHead`
- 17.2.3.3.0.10.10 `volatile uint16_t Ipuart_handle_t::rxRingBufferTail`
- 17.2.3.3.0.10.11 `Ipuart_transfer_callback_t Ipuart_handle_t::callback`
- 17.2.3.3.0.10.12 `void* Ipuart_handle_t::userData`
- 17.2.3.3.0.10.13 `volatile uint8_t Ipuart_handle_t::txState`
- 17.2.3.3.0.10.14 `volatile uint8_t Ipuart_handle_t::rxState`

### 17.2.4 Macro Definition Documentation

- 17.2.4.1 `#define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 2, 3))`

### 17.2.5 Typedef Documentation

- 17.2.5.1 `typedef void(* Ipuart_transfer_callback_t)(LPUART_Type *base, Ipuart_handle_t *handle, status_t status, void *userData)`

### 17.2.6 Enumeration Type Documentation

#### 17.2.6.1 enum \_Ipuart\_status

Enumerator

- `kStatus_LPUART_TxBusy` TX busy.
- `kStatus_LPUART_RxBusy` RX busy.
- `kStatus_LPUART_TxIdle` LPUART transmitter is idle.

*kStatus\_LPUART\_RxIdle* LPUART receiver is idle.  
*kStatus\_LPUART\_TxWatermarkTooLarge* TX FIFO watermark too large.  
*kStatus\_LPUART\_RxWatermarkTooLarge* RX FIFO watermark too large.  
*kStatus\_LPUART\_FlagCannotClearManually* Some flag can't manually clear.  
*kStatus\_LPUART\_Error* Error happens on LPUART.  
*kStatus\_LPUART\_RxRingBufferOverrun* LPUART RX software ring buffer overrun.  
*kStatus\_LPUART\_RxHardwareOverrun* LPUART RX receiver overrun.  
*kStatus\_LPUART\_NoiseError* LPUART noise error.  
*kStatus\_LPUART\_FramingError* LPUART framing error.  
*kStatus\_LPUART\_ParityError* LPUART parity error.  
*kStatus\_LPUART\_BaudrateNotSupport* Baudrate is not support in current clock source.

### 17.2.6.2 enum lpuart\_parity\_mode\_t

Enumerator

*kLPUART\_ParityDisabled* Parity disabled.  
*kLPUART\_ParityEven* Parity enabled, type even, bit setting: PE|PT = 10.  
*kLPUART\_ParityOdd* Parity enabled, type odd, bit setting: PE|PT = 11.

### 17.2.6.3 enum lpuart\_data\_bits\_t

Enumerator

*kLPUART\_EightDataBits* Eight data bit.

### 17.2.6.4 enum lpuart\_stop\_bit\_count\_t

Enumerator

*kLPUART\_OneStopBit* One stop bit.  
*kLPUART\_TwoStopBit* Two stop bits.

### 17.2.6.5 enum \_lpuart\_interrupt\_enable

This structure contains the settings for all LPUART interrupt configurations.

Enumerator

*kLPUART\_LinBreakInterruptEnable* LIN break detect.  
*kLPUART\_RxActiveEdgeInterruptEnable* Receive Active Edge.  
*kLPUART\_TxDataRegEmptyInterruptEnable* Transmit data register empty.

***kLPUART\_TransmissionCompleteInterruptEnable*** Transmission complete.  
***kLPUART\_RxDataRegFullInterruptEnable*** Receiver data register full.  
***kLPUART\_IdleLineInterruptEnable*** Idle line.  
***kLPUART\_RxOverrunInterruptEnable*** Receiver Overrun.  
***kLPUART\_NoiseErrorInterruptEnable*** Noise error flag.  
***kLPUART\_FramingErrorInterruptEnable*** Framing error flag.  
***kLPUART\_ParityErrorInterruptEnable*** Parity error flag.

### 17.2.6.6 enum \_lpuart\_flags

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

Enumerator

***kLPUART\_TxDataRegEmptyFlag*** Transmit data register empty flag, sets when transmit buffer is empty.  
***kLPUART\_TransmissionCompleteFlag*** Transmission complete flag, sets when transmission activity complete.  
***kLPUART\_RxDataRegFullFlag*** Receive data register full flag, sets when the receive data buffer is full.  
***kLPUART\_IdleLineFlag*** Idle line detect flag, sets when idle line detected.  
***kLPUART\_RxOverrunFlag*** Receive Overrun, sets when new data is received before data is read from receive register.  
***kLPUART\_NoiseErrorFlag*** Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets  
***kLPUART\_FramingErrorFlag*** Frame error flag, sets if logic 0 was detected where stop bit expected.  
***kLPUART\_ParityErrorFlag*** If parity enabled, sets upon parity error detection.  
***kLPUART\_LinBreakFlag*** LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled.  
***kLPUART\_RxActiveEdgeFlag*** Receive pin active edge interrupt flag, sets when active edge detected.  
***kLPUART\_RxActiveFlag*** Receiver Active Flag (RAF), sets at beginning of valid start bit.  
***kLPUART\_DataMatch1Flag*** The next character to be read from LPUART\_DATA matches MA1.  
***kLPUART\_DataMatch2Flag*** The next character to be read from LPUART\_DATA matches MA2.  
***kLPUART\_NoiseErrorInRxDataRegFlag*** NOISY bit, sets if noise detected in current data word.  
***kLPUART\_ParityErrorInRxDataRegFlag*** PARITYE bit, sets if noise detected in current data word.

## 17.2.7 Function Documentation

### 17.2.7.1 status\_t LPUART\_Init ( LPUART\_Type \* *base*, const lpuart\_config\_t \* *config*, uint32\_t *srcClock\_Hz* )

This function configures the LPUART module with user-defined settings. Call the [LPUART\\_GetDefaultConfig\(\)](#) function to configure the configuration structure and get the default configuration. The example below shows how to use this API to configure the LPUART.

```
* lpuart_config_t lpuartConfig;
* lpuartConfig.baudRate_Bps = 115200U;
* lpuartConfig.parityMode = kLPUART_ParityDisabled;
* lpuartConfig.dataBitsCount = kLPUART_EightDataBits;
* lpuartConfig.isMsb = false;
* lpuartConfig.stopBitCount = kLPUART_OneStopBit;
* lpuartConfig.txFifoWatermark = 0;
* lpuartConfig.rxFifoWatermark = 1;
* LPUART_Init(LPUART1, &lpuartConfig, 20000000U);
*
```

Parameters

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

Return values

|                                          |                                                  |
|------------------------------------------|--------------------------------------------------|
| <i>kStatus_LPUART_BaudrateNotSupport</i> | Baudrate is not support in current clock source. |
| <i>kStatus_Success</i>                   | LPUART initialize succeed                        |

### 17.2.7.2 void LPUART\_Deinit ( LPUART\_Type \* *base* )

This function waits for transmit to complete, disables TX and RX, and disables the LPUART clock.

Parameters

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

### 17.2.7.3 void LPUART\_GetDefaultConfig ( lpuart\_config\_t \* *config* )

This function initializes the LPUART configuration structure to a default value. The default values are:  
: lpuartConfig->baudRate\_Bps = 115200U; lpuartConfig->parityMode = kLPUART\_ParityDisabled;

## LPUART Driver

```
lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;
```

## Parameters

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

**17.2.7.4 status\_t LPUART\_SetBaudRate ( LPUART\_Type \* *base*, uint32\_t *baudRate\_Bps*, uint32\_t *srcClock\_Hz* )**

This function configures the LPUART module baudrate. This function is used to update the LPUART module baudrate after the LPUART module is initialized by the LPUART\_Init.

```
* LPUART_SetBaudRate(LPUART1, 115200U, 20000000U);
*
```

## Parameters

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

## Return values

|                                          |                                                        |
|------------------------------------------|--------------------------------------------------------|
| <i>kStatus_LPUART_BaudrateNotSupport</i> | Baudrate is not supported in the current clock source. |
| <i>kStatus_Success</i>                   | Set baudrate succeeded.                                |

**17.2.7.5 uint32\_t LPUART\_GetStatusFlags ( LPUART\_Type \* *base* )**

This function gets all LPUART status flags. The flags are returned as the logical OR value of the enumerators [\\_lpuart\\_flags](#). To check for a specific status, compare the return value with enumerators in the [\\_lpuart\\_flags](#). For example, to check whether the TX is empty:

```
* if (kLPUART_TxDataRegEmptyFlag &
* LPUART_GetStatusFlags(LPUART1))
* {
* ...
* }
*
```

## LPUART Driver

Parameters

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

Returns

LPUART status flags which are ORed by the enumerators in the \_lpuart\_flags.

### 17.2.7.6 **status\_t LPUART\_ClearStatusFlags ( LPUART\_Type \* *base*, uint32\_t *mask* )**

This function clears LPUART status flags with a provided mask. Automatically cleared flags can't be cleared by this function. Flags that can only be cleared or set by hardware are: kLPUART\_TxDataRegEmptyFlag, kLPUART\_TransmissionCompleteFlag, kLPUART\_RxDataRegFullFlag, kLPUART\_RxActiveFlag, kLPUART\_NoiseErrorInRxDataRegFlag, kLPUART\_ParityErrorInRxDataRegFlag, kLPUART\_TxFifoEmptyFlag, kLPUART\_RxFifoEmptyFlag. Note: This API should be called when the Tx/Rx is idle, otherwise it takes no effects.

Parameters

|             |                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPUART peripheral base address.                                                                                                        |
| <i>mask</i> | the status flags to be cleared. The user can use the enumerators in the _lpuart_status_flag_t to do the OR operation and get the mask. |

Returns

0 succeed, others failed.

Return values

|                                               |                                                                                         |
|-----------------------------------------------|-----------------------------------------------------------------------------------------|
| <i>kStatus_LPUART_FlagCannotClearManually</i> | The flag can't be cleared by this function but it is cleared automatically by hardware. |
| <i>kStatus_Success</i>                        | Status in the mask are cleared.                                                         |

### 17.2.7.7 **void LPUART\_EnableInterrupts ( LPUART\_Type \* *base*, uint32\_t *mask* )**

This function enables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See the [\\_lpuart\\_interrupt\\_enable](#). This examples shows how to enable TX empty interrupt and RX full interrupt:

```
* LPUART_EnableInterrupts(LPUART1,
* kLPUART_TxDataRegEmptyInterruptEnable |
* kLPUART_RxDataRegFullInterruptEnable);
*
```

## Parameters

|             |                                                                                    |
|-------------|------------------------------------------------------------------------------------|
| <i>base</i> | LPUART peripheral base address.                                                    |
| <i>mask</i> | The interrupts to enable. Logical OR of <a href="#">_lpuart_interrupt_enable</a> . |

**17.2.7.8 void LPUART\_DisableInterrupts ( LPUART\_Type \* *base*, uint32\_t *mask* )**

This function disables the LPUART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See [\\_lpuart\\_interrupt\\_enable](#). This example shows how to disable the TX empty interrupt and RX full interrupt:

```
* LPUART_DisableInterrupts(LPUART1,
* kLPUART_TxDataRegEmptyInterruptEnable |
* kLPUART_RxDataRegFullInterruptEnable);
*
```

## Parameters

|             |                                                                                     |
|-------------|-------------------------------------------------------------------------------------|
| <i>base</i> | LPUART peripheral base address.                                                     |
| <i>mask</i> | The interrupts to disable. Logical OR of <a href="#">_lpuart_interrupt_enable</a> . |

**17.2.7.9 uint32\_t LPUART\_GetEnabledInterrupts ( LPUART\_Type \* *base* )**

This function gets the enabled LPUART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators [\\_lpuart\\_interrupt\\_enable](#). To check a specific interrupt enable status, compare the return value with enumerators in [\\_lpuart\\_interrupt\\_enable](#). For example, to check whether the TX empty interrupt is enabled:

```
* uint32_t enabledInterrupts = LPUART_GetEnabledInterrupts(LPUART1);
*
* if (kLPUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)
* {
* ...
* }
*
```

## Parameters

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

## Returns

LPUART interrupt flags which are logical OR of the enumerators in [\\_lpuart\\_interrupt\\_enable](#).

### 17.2.7.10 static uint32\_t LPUART\_GetDataRegisterAddress ( LPUART\_Type \* *base* ) [inline], [static]

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

Parameters

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

Returns

LPUART data register addresses which are used both by the transmitter and receiver.

#### 17.2.7.11 static void LPUART\_EnableTxDMA ( LPUART\_Type \* *base*, bool *enable* ) [inline], [static]

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

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | LPUART peripheral base address.   |
| <i>enable</i> | True to enable, false to disable. |

#### 17.2.7.12 static void LPUART\_EnableRxDMA ( LPUART\_Type \* *base*, bool *enable* ) [inline], [static]

This function enables or disables the receiver data register full flag, STAT[RDRF], to generate DMA requests.

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | LPUART peripheral base address.   |
| <i>enable</i> | True to enable, false to disable. |

#### 17.2.7.13 static void LPUART\_EnableTx ( LPUART\_Type \* *base*, bool *enable* ) [inline], [static]

This function enables or disables the LPUART transmitter.

Parameters

## LPUART Driver

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | LPUART peripheral base address.   |
| <i>enable</i> | True to enable, false to disable. |

### 17.2.7.14 static void LPUART\_EnableRx ( LPUART\_Type \* *base*, bool *enable* ) [inline], [static]

This function enables or disables the LPUART receiver.

Parameters

|               |                                   |
|---------------|-----------------------------------|
| <i>base</i>   | LPUART peripheral base address.   |
| <i>enable</i> | True to enable, false to disable. |

### 17.2.7.15 static void LPUART\_WriteByte ( LPUART\_Type \* *base*, uint8\_t *data* ) [inline], [static]

This function writes data to the transmitter register directly. The upper layer must ensure that the TX register is empty or that the TX FIFO has room before calling this function.

Parameters

|             |                                 |
|-------------|---------------------------------|
| <i>base</i> | LPUART peripheral base address. |
| <i>data</i> | Data write to the TX register.  |

### 17.2.7.16 static uint8\_t LPUART\_ReadByte ( LPUART\_Type \* *base* ) [inline], [static]

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

Parameters

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

Returns

Data read from data register.

### 17.2.7.17 void LPUART\_WriteBlocking ( LPUART\_Type \* *base*, const uint8\_t \* *data*, size\_t *length* )

This function polls the transmitter register, waits for the register to be empty or for TX FIFO to have room, and writes data to the transmitter buffer.

#### Note

This function does not check whether all data has been sent out to the bus. Before disabling the transmitter, check the kLPUART\_TransmissionCompleteFlag to ensure that the transmit is finished.

#### Parameters

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

### 17.2.7.18 status\_t LPUART\_ReadBlocking ( LPUART\_Type \* *base*, uint8\_t \* *data*, size\_t *length* )

This function polls the receiver register, waits for the receiver register full or receiver FIFO has data, and reads data from the TX register.

#### Parameters

|               |                                                         |
|---------------|---------------------------------------------------------|
| <i>base</i>   | LPUART 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_LPUART_Rx-HardwareOverrun</i> | Receiver overrun happened while receiving data. |
| <i>kStatus_LPUART_Noise-Error</i>        | Noise error happened while receiving data.      |
| <i>kStatus_LPUART_FramingError</i>       | Framing error happened while receiving data.    |
| <i>kStatus_LPUART_Parity-Error</i>       | Parity error happened while receiving data.     |
| <i>kStatus_Success</i>                   | Successfully received all data.                 |

## LPUART Driver

### 17.2.7.19 void LPUART\_TransferCreateHandle ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle*, Ipuart\_transfer\_callback\_t *callback*, void \* *userData* )

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

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

Parameters

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

### 17.2.7.20 status\_t LPUART\_TransferSendNonBlocking ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle*, Ipuart\_transfer\_t \* *xfer* )

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the [kStatus\\_LPUART\\_TxIdle](#) as status parameter.

Note

The [kStatus\\_LPUART\\_TxIdle](#) is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the T-X, check the [kLPUART\\_TransmissionCompleteFlag](#) to ensure that the transmit is finished.

Parameters

|               |                                                                    |
|---------------|--------------------------------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.                                    |
| <i>handle</i> | LPUART handle pointer.                                             |
| <i>xfer</i>   | LPUART transfer structure, see <a href="#">Ipuart_transfer_t</a> . |

Return values

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

### 17.2.7.21 void LPUART\_TransferStartRingBuffer ( LPUART\_Type \* *base*, Ipuart\_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 is stored into the ring buffer even when the user doesn't call the [UART\\_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 RX ring buffer, one byte is reserved for internal use. In other words, if *ringBufferSize* is 32, then only 31 bytes are used for saving data.

Parameters

|                       |                                                                                              |
|-----------------------|----------------------------------------------------------------------------------------------|
| <i>base</i>           | LPUART peripheral base address.                                                              |
| <i>handle</i>         | LPUART 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.                                                                     |

### 17.2.7.22 void LPUART\_TransferStopRingBuffer ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle* )

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |

### 17.2.7.23 void LPUART\_TransferAbortSend ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle* )

This function aborts the interrupt driven data sending. The user can get the remainBtyes to find out how many bytes are not sent out.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |

#### 17.2.7.24 status\_t LPUART\_TransferGetSendCount ( **LPUART\_Type \* base,** **Ipuart\_handle\_t \* handle, uint32\_t \* count** )

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

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |
| <i>count</i>  | Send bytes count.               |

Return values

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

#### 17.2.7.25 status\_t LPUART\_TransferReceiveNonBlocking ( **LPUART\_Type \* base,** **Ipuart\_handle\_t \* handle, Ipuart\_transfer\_t \* xfer, size\_t \* receivedBytes** )

This function receives data using an interrupt method. This is a non-blocking function which returns without waiting to ensure that all data are 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 for read, the receive request is saved by the LPUART driver. When the new data arrives, the receive request is serviced first. When all data is received, the LPUART driver notifies the upper layer through a callback function and passes a status parameter *kStatus\_UART\_RxIdle*. For example, the upper layer needs 10 bytes but there are only 5 bytes in ring buffer. The 5 bytes are copied to *xfer->data*, which returns with the parameter *receivedBytes* set to 5. For the remaining 5 bytes, the newly arrived data is saved from *xfer->data[5]*. When 5 bytes are received, the LPUART 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 *xfer->data*. When all data is received, the upper layer is notified.

## LPUART Driver

Parameters

|                      |                                                                  |
|----------------------|------------------------------------------------------------------|
| <i>base</i>          | LPUART peripheral base address.                                  |
| <i>handle</i>        | LPUART handle pointer.                                           |
| <i>xfer</i>          | LPUART transfer structure, see <a href="#">uart_transfer_t</a> . |
| <i>receivedBytes</i> | Bytes received from the ring buffer directly.                    |

Return values

|                                |                                                          |
|--------------------------------|----------------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully queue the transfer into the transmit queue. |
| <i>kStatus_LPUART_Rx-Busy</i>  | Previous receive request is not finished.                |
| <i>kStatus_InvalidArgument</i> | Invalid argument.                                        |

### 17.2.7.26 void LPUART\_TransferAbortReceive ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle* )

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to find out how many bytes not received yet.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |

### 17.2.7.27 status\_t LPUART\_TransferGetReceiveCount ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle*, uint32\_t \* *count* )

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

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART 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 count; |

#### 17.2.7.28 void LPUART\_TransferHandleIRQ ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle* )

This function handles the LPUART transmit and receive IRQ request.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |

#### 17.2.7.29 void LPUART\_TransferHandleErrorIRQ ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle* )

This function handles the LPUART error IRQ request.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |

### 17.3 LPUART DMA Driver

#### 17.3.1 Overview

#### Data Structures

- struct [lpuart\\_dma\\_handle\\_t](#)  
*LPUART DMA handle.* [More...](#)

#### TypeDefs

- typedef void(\* [lpuart\\_dma\\_transfer\\_callback\\_t](#))[\(LPUART\\_Type \\*base, lpuart\\_dma\\_handle\\_t \\*handle, status\\_t status, void \\*userData\)](#)  
*LPUART transfer callback function.*

#### EDMA transactional

- void [LPUART\\_TransferCreateHandleDMA](#) (LPUART\_Type \*base, lpuart\_dma\_handle\_t \*handle, [lpuart\\_dma\\_transfer\\_callback\\_t](#) callback, void \*userData, [dma\\_handle\\_t](#) \*txDmaHandle, [dma\\_handle\\_t](#) \*rxDmaHandle)  
*Initializes the LPUART handle which is used in transactional functions.*
- status\_t [LPUART\\_TransferSendDMA](#) (LPUART\_Type \*base, lpuart\_dma\_handle\_t \*handle, [lpuart\\_transfer\\_t](#) \*xfer)  
*Sends data using DMA.*
- status\_t [LPUART\\_TransferReceiveDMA](#) (LPUART\_Type \*base, lpuart\_dma\_handle\_t \*handle, [lpuart\\_transfer\\_t](#) \*xfer)  
*Receives data using DMA.*
- void [LPUART\\_TransferAbortSendDMA](#) (LPUART\_Type \*base, lpuart\_dma\_handle\_t \*handle)  
*Aborts the sent data using DMA.*
- void [LPUART\\_TransferAbortReceiveDMA](#) (LPUART\_Type \*base, lpuart\_dma\_handle\_t \*handle)  
*Aborts the received data using DMA.*
- status\_t [LPUART\\_TransferGetSendCountDMA](#) (LPUART\_Type \*base, lpuart\_dma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes written to the LPUART TX register.*
- status\_t [LPUART\\_TransferGetReceiveCountDMA](#) (LPUART\_Type \*base, lpuart\_dma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of received bytes.*

#### 17.3.2 Data Structure Documentation

##### 17.3.2.1 struct [\\_lpuart\\_dma\\_handle](#)

###### Data Fields

- [lpuart\\_dma\\_transfer\\_callback\\_t](#) [callback](#)

- *Callback function.*
- `void * userData`  
*LPUART 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.*

### 17.3.2.1.0.11 Field Documentation

17.3.2.1.0.11.1 `lpuart_dma_transfer_callback_t lpuart_dma_handle_t::callback`

17.3.2.1.0.11.2 `void* lpuart_dma_handle_t::userData`

17.3.2.1.0.11.3 `size_t lpuart_dma_handle_t::rxDataSizeAll`

17.3.2.1.0.11.4 `size_t lpuart_dma_handle_t::txDataSizeAll`

17.3.2.1.0.11.5 `dma_handle_t* lpuart_dma_handle_t::txDmaHandle`

17.3.2.1.0.11.6 `dma_handle_t* lpuart_dma_handle_t::rxDmaHandle`

17.3.2.1.0.11.7 `volatile uint8_t lpuart_dma_handle_t::txState`

### 17.3.3 Typedef Documentation

17.3.3.1 `typedef void(* lpuart_dma_transfer_callback_t)(LPUART_Type *base,  
lpuart_dma_handle_t *handle, status_t status, void *userData)`

### 17.3.4 Function Documentation

17.3.4.1 `void LPUART_TransferCreateHandleDMA ( LPUART_Type * base,  
lpuart_dma_handle_t * handle, lpuart_dma_transfer_callback_t callback, void *  
userData, dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle )`

## LPUART DMA Driver

Parameters

|                    |                                                |
|--------------------|------------------------------------------------|
| <i>base</i>        | LPUART peripheral base address.                |
| <i>handle</i>      | Pointer to lpuart_dma_handle_t structure.      |
| <i>callback</i>    | Callback function.                             |
| <i>userData</i>    | User data.                                     |
| <i>txDmaHandle</i> | User-requested DMA handle for TX DMA transfer. |
| <i>rxDmaHandle</i> | User-requested DMA handle for RX DMA transfer. |

### 17.3.4.2 status\_t LPUART\_TransferSendDMA ( **LPUART\_Type** \* *base*,                           **lpuart\_dma\_handle\_t** \* *handle*, **lpuart\_transfer\_t** \* *xfer* )

This function sends data using DMA. 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>   | LPUART peripheral base address.                                        |
| <i>handle</i> | LPUART handle pointer.                                                 |
| <i>xfer</i>   | LPUART DMA transfer structure. See <a href="#">lpuart_transfer_t</a> . |

Return values

|                                |                             |
|--------------------------------|-----------------------------|
| <i>kStatus_Success</i>         | if succeed, others failed.  |
| <i>kStatus_LPUART_TxBusy</i>   | Previous transfer on going. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.           |

### 17.3.4.3 status\_t LPUART\_TransferReceiveDMA ( **LPUART\_Type** \* *base*,                           **lpuart\_dma\_handle\_t** \* *handle*, **lpuart\_transfer\_t** \* *xfer* )

This function receives data using DMA. 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>   | LPUART peripheral base address.                                        |
| <i>handle</i> | Pointer to lpuart_dma_handle_t structure.                              |
| <i>xfer</i>   | LPUART DMA transfer structure. See <a href="#">lpuart_transfer_t</a> . |

Return values

|                                |                             |
|--------------------------------|-----------------------------|
| <i>kStatus_Success</i>         | if succeed, others failed.  |
| <i>kStatus_LPUART_Rx-Busy</i>  | Previous transfer on going. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.           |

#### 17.3.4.4 void LPUART\_TransferAbortSendDMA ( LPUART\_Type \* *base*, lpuart\_dma\_handle\_t \* *handle* )

This function aborts send data using DMA.

Parameters

|               |                                          |
|---------------|------------------------------------------|
| <i>base</i>   | LPUART peripheral base address           |
| <i>handle</i> | Pointer to lpuart_dma_handle_t structure |

#### 17.3.4.5 void LPUART\_TransferAbortReceiveDMA ( LPUART\_Type \* *base*, lpuart\_dma\_handle\_t \* *handle* )

This function aborts the received data using DMA.

Parameters

|               |                                          |
|---------------|------------------------------------------|
| <i>base</i>   | LPUART peripheral base address           |
| <i>handle</i> | Pointer to lpuart_dma_handle_t structure |

#### 17.3.4.6 status\_t LPUART\_TransferGetSendCountDMA ( LPUART\_Type \* *base*, lpuart\_dma\_handle\_t \* *handle*, uint32\_t \* *count* )

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

## LPUART DMA Driver

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART 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> ; |

### 17.3.4.7 **status\_t LPUART\_TransferGetReceiveCountDMA ( LPUART\_Type \* *base*, Ipuart\_dma\_handle\_t \* *handle*, uint32\_t \* *count* )**

This function gets the number of received bytes.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART 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> ; |

## 17.4 LPUART eDMA Driver

### 17.4.1 Overview

#### Data Structures

- struct [lpuart\\_edma\\_handle\\_t](#)  
*LPUART eDMA handle.* [More...](#)

#### TypeDefs

- [typedef void\(\\* lpuart\\_edma\\_transfer\\_callback\\_t \)](#)(LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle, status\_t status, void \*userData)  
*LPUART transfer callback function.*

#### eDMA transactional

- void [LPUART\\_TransferCreateHandleEDMA](#) (LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle, [lpuart\\_edma\\_transfer\\_callback\\_t](#) callback, void \*userData, edma\_handle\_t \*txEdmaHandle, edma\_handle\_t \*rxEdmaHandle)  
*Initializes the LPUART handle which is used in transactional functions.*
- status\_t [LPUART\\_SendEDMA](#) (LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle, [lpuart\\_transfer\\_t](#) \*xfer)  
*Sends data using eDMA.*
- status\_t [LPUART\\_ReceiveEDMA](#) (LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle, [lpuart\\_transfer\\_t](#) \*xfer)  
*Receives data using eDMA.*
- void [LPUART\\_TransferAbortSendEDMA](#) (LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle)  
*Aborts the sent data using eDMA.*
- void [LPUART\\_TransferAbortReceiveEDMA](#) (LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle)  
*Aborts the received data using eDMA.*
- status\_t [LPUART\\_TransferGetSendCountEDMA](#) (LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of bytes written to the LPUART TX register.*
- status\_t [LPUART\\_TransferGetReceiveCountEDMA](#) (LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle, uint32\_t \*count)  
*Gets the number of received bytes.*

### 17.4.2 Data Structure Documentation

#### 17.4.2.1 struct \_lpuart\_edma\_handle

##### Data Fields

- `lpuart_edma_transfer_callback_t callback`  
*Callback function.*
- `void *userData`  
*LPUART 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.*

#### 17.4.2.1.0.12 Field Documentation

17.4.2.1.0.12.1 `lpuart_edma_transfer_callback_t lpuart_edma_handle_t::callback`

17.4.2.1.0.12.2 `void* lpuart_edma_handle_t::userData`

17.4.2.1.0.12.3 `size_t lpuart_edma_handle_t::rxDataSizeAll`

17.4.2.1.0.12.4 `size_t lpuart_edma_handle_t::txDataSizeAll`

17.4.2.1.0.12.5 `edma_handle_t* lpuart_edma_handle_t::txEdmaHandle`

17.4.2.1.0.12.6 `edma_handle_t* lpuart_edma_handle_t::rxEdmaHandle`

17.4.2.1.0.12.7 `uint8_t lpuart_edma_handle_t::nbytes`

17.4.2.1.0.12.8 `volatile uint8_t lpuart_edma_handle_t::txState`

#### 17.4.3 Typedef Documentation

17.4.3.1 `typedef void(* lpuart_edma_transfer_callback_t)(LPUART_Type *base,  
lpuart_edma_handle_t *handle, status_t status, void *userData)`

#### 17.4.4 Function Documentation

17.4.4.1 `void LPUART_TransferCreateHandleEDMA ( LPUART_Type * base,  
lpuart_edma_handle_t * handle, lpuart_edma_transfer_callback_t callback, void  
* userData, edma_handle_t * txEdmaHandle, edma_handle_t * rxEdmaHandle )`

## LPUART eDMA Driver

Parameters

|                     |                                                |
|---------------------|------------------------------------------------|
| <i>base</i>         | LPUART peripheral base address.                |
| <i>handle</i>       | Pointer to lpuart_edma_handle_t structure.     |
| <i>callback</i>     | Callback function.                             |
| <i>userData</i>     | User data.                                     |
| <i>txEdmaHandle</i> | User requested DMA handle for TX DMA transfer. |
| <i>rxEdmaHandle</i> | User requested DMA handle for RX DMA transfer. |

### 17.4.4.2 status\_t LPUART\_SendEDMA ( LPUART\_Type \* *base*, lpuart\_edma\_handle\_t \* *handle*, lpuart\_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>   | LPUART peripheral base address.                                         |
| <i>handle</i> | LPUART handle pointer.                                                  |
| <i>xfer</i>   | LPUART eDMA transfer structure. See <a href="#">lpuart_transfer_t</a> . |

Return values

|                                |                             |
|--------------------------------|-----------------------------|
| <i>kStatus_Success</i>         | if succeed, others failed.  |
| <i>kStatus_LPUART_TxBusy</i>   | Previous transfer on going. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.           |

### 17.4.4.3 status\_t LPUART\_ReceiveEDMA ( LPUART\_Type \* *base*, lpuart\_edma\_handle\_t \* *handle*, lpuart\_transfer\_t \* *xfer* )

This function receives data using eDMA. This is non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

|               |                                                                         |
|---------------|-------------------------------------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.                                         |
| <i>handle</i> | Pointer to lpuart_edma_handle_t structure.                              |
| <i>xfer</i>   | LPUART eDMA transfer structure, see <a href="#">lpuart_transfer_t</a> . |

Return values

|                                |                            |
|--------------------------------|----------------------------|
| <i>kStatus_Success</i>         | if succeed, others fail.   |
| <i>kStatus_LPUART_Rx-Busy</i>  | Previous transfer ongoing. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.          |

#### 17.4.4.4 void LPUART\_TransferAbortSendEDMA ( LPUART\_Type \* *base*, lpuart\_edma\_handle\_t \* *handle* )

This function aborts the sent data using eDMA.

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.            |
| <i>handle</i> | Pointer to lpuart_edma_handle_t structure. |

#### 17.4.4.5 void LPUART\_TransferAbortReceiveEDMA ( LPUART\_Type \* *base*, lpuart\_edma\_handle\_t \* *handle* )

This function aborts the received data using eDMA.

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.            |
| <i>handle</i> | Pointer to lpuart_edma_handle_t structure. |

#### 17.4.4.6 status\_t LPUART\_TransferGetSendCountEDMA ( LPUART\_Type \* *base*, lpuart\_edma\_handle\_t \* *handle*, uint32\_t \* *count* )

This function gets the number of bytes written to the LPUART TX register by DMA.

## LPUART eDMA Driver

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |
| <i>count</i>  | Send bytes count.               |

Return values

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

### 17.4.4.7 **status\_t LPUART\_TransferGetReceiveCountEDMA ( LPUART\_Type \* *base*, Ipuart\_edma\_handle\_t \* *handle*, uint32\_t \* *count* )**

This function gets the number of received bytes.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |
| <i>count</i>  | Receive bytes count.            |

Return values

|                                      |                                                       |
|--------------------------------------|-------------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | No receive in progress.                               |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

## 17.5 LPUART µCOS/II Driver

### 17.5.1 Overview

#### Data Structures

- struct **lpuart\_rtos\_config\_t**  
*LPUART RTOS configuration structure.* [More...](#)

#### LPUART RTOS Operation

- int **LPUART\_RTOS\_Init** (lpuart\_rtos\_handle\_t \*handle, lpuart\_handle\_t \*t\_handle, const lpuart\_rtos\_config\_t \*cfg)  
*Initializes an LPUART instance for operation in RTOS.*
- int **LPUART\_RTOS\_Deinit** (lpuart\_rtos\_handle\_t \*handle)  
*Deinitializes an LPUART instance for operation.*

#### LPUART transactional Operation

- int **LPUART\_RTOS\_Send** (lpuart\_rtos\_handle\_t \*handle, const uint8\_t \*buffer, uint32\_t length)  
*Sends data in the background.*
- int **LPUART\_RTOS\_Receive** (lpuart\_rtos\_handle\_t \*handle, uint8\_t \*buffer, uint32\_t length, size\_t \*received)  
*Receives data.*

### 17.5.2 Data Structure Documentation

#### 17.5.2.1 struct lpuart\_rtos\_config\_t

##### Data Fields

- **LPUART\_Type \* base**  
*UART base address.*
- **uint32\_t srecclk**  
*UART source clock in Hz.*
- **uint32\_t baudrate**  
*Desired communication speed.*
- **lpuart\_parity\_mode\_t parity**  
*Parity setting.*
- **lpuart\_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.*

### 17.5.3 Function Documentation

17.5.3.1 `int LPUART_RTOS_Init( Ipuart_rtos_handle_t * handle, Ipuart_handle_t * t_handle, const Ipuart_rtos_config_t * cfg )`

Parameters

|                        |                                                                                      |
|------------------------|--------------------------------------------------------------------------------------|
| <i>handle</i>          | The RTOS LPUART handle, the pointer to an allocated space for RTOS context.          |
| <i>lpuart_t_handle</i> | The pointer to an allocated space to store the transactional layer internal state.   |
| <i>cfg</i>             | The pointer to the parameters required to configure the LPUART after initialization. |

Returns

0 succeed, others failed

#### 17.5.3.2 int LPUART\_RTOS\_Deinit ( *lpuart\_rtos\_handle\_t \* handle* )

This function deinitializes the LPUART module, sets all register values to the reset value, and releases the resources.

Parameters

|               |                         |
|---------------|-------------------------|
| <i>handle</i> | The RTOS LPUART handle. |
|---------------|-------------------------|

#### 17.5.3.3 int LPUART\_RTOS\_Send ( *lpuart\_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 LPUART handle.        |
| <i>buffer</i> | The pointer to buffer to send. |
| <i>length</i> | The number of bytes to send.   |

#### 17.5.3.4 int LPUART\_RTOS\_Receive ( *lpuart\_rtos\_handle\_t \* handle, uint8\_t \* buffer, uint32\_t length, size\_t \* received* )

This function receives data from LPUART. It is a synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

## LPUART µCOS/II Driver

### Parameters

|                 |                                                                                  |
|-----------------|----------------------------------------------------------------------------------|
| <i>handle</i>   | The RTOS LPUART 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 a variable of size_t where the number of received data is filled. |

## 17.6 LPUART µCOS/III Driver

### 17.6.1 Overview

#### Data Structures

- struct **lpuart\_rtos\_config\_t**  
*LPUART RTOS configuration structure.* [More...](#)

#### LPUART RTOS Operation

- int **LPUART\_RTOS\_Init** (lpuart\_rtos\_handle\_t \*handle, lpuart\_handle\_t \*t\_handle, const lpuart\_rtos\_config\_t \*cfg)  
*Initializes an LPUART instance for operation in RTOS.*
- int **LPUART\_RTOS\_Deinit** (lpuart\_rtos\_handle\_t \*handle)  
*Deinitializes an LPUART instance for operation.*

#### LPUART transactional Operation

- int **LPUART\_RTOS\_Send** (lpuart\_rtos\_handle\_t \*handle, const uint8\_t \*buffer, uint32\_t length)  
*Sends data in the background.*
- int **LPUART\_RTOS\_Receive** (lpuart\_rtos\_handle\_t \*handle, uint8\_t \*buffer, uint32\_t length, size\_t \*received)  
*Receives data.*

### 17.6.2 Data Structure Documentation

#### 17.6.2.1 struct lpuart\_rtos\_config\_t

##### Data Fields

- **LPUART\_Type \* base**  
*UART base address.*
- **uint32\_t srecclk**  
*UART source clock in Hz.*
- **uint32\_t baudrate**  
*Desired communication speed.*
- **lpuart\_parity\_mode\_t parity**  
*Parity setting.*
- **lpuart\_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.*

### 17.6.3 Function Documentation

17.6.3.1 `int LPUART_RTOS_Init( Ipuart_rtos_handle_t * handle, Ipuart_handle_t * t_handle, const Ipuart_rtos_config_t * cfg )`

Parameters

|                        |                                                                                      |
|------------------------|--------------------------------------------------------------------------------------|
| <i>handle</i>          | The RTOS LPUART handle, the pointer to allocated space for RTOS context.             |
| <i>lpuart_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 LPUART after initialization. |

Returns

0 succeed, others failed

#### 17.6.3.2 int LPUART\_RTOS\_Deinit ( *lpuart\_rtos\_handle\_t \* handle* )

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

Parameters

|               |                         |
|---------------|-------------------------|
| <i>handle</i> | The RTOS LPUART handle. |
|---------------|-------------------------|

#### 17.6.3.3 int LPUART\_RTOS\_Send ( *lpuart\_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 LPUART handle.        |
| <i>buffer</i> | The pointer to buffer to send. |
| <i>length</i> | The number of bytes to send.   |

#### 17.6.3.4 int LPUART\_RTOS\_Receive ( *lpuart\_rtos\_handle\_t \* handle, uint8\_t \* buffer, uint32\_t length, size\_t \* received* )

It is a synchronous API.

This function receives data from LPUART. If any data is immediately available it will be returned imidiately and the number of bytes received.

## LPUART µCOS/III Driver

### Parameters

|                 |                                                                                     |
|-----------------|-------------------------------------------------------------------------------------|
| <i>handle</i>   | The RTOS LPUART 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 will be filled. |

## 17.7 LPUART FreeRTOS Driver

### 17.7.1 Overview

#### Data Structures

- struct **lpuart\_rtos\_config\_t**  
*LPUART RTOS configuration structure.* [More...](#)

#### LPUART RTOS Operation

- int **LPUART\_RTOS\_Init** (lpuart\_rtos\_handle\_t \*handle, lpuart\_handle\_t \*t\_handle, const **lpuart\_rtos\_config\_t** \*cfg)  
*Initializes an LPUART instance for operation in RTOS.*
- int **LPUART\_RTOS\_Deinit** (lpuart\_rtos\_handle\_t \*handle)  
*Deinitializes an LPUART instance for operation.*

#### LPUART transactional Operation

- int **LPUART\_RTOS\_Send** (lpuart\_rtos\_handle\_t \*handle, const uint8\_t \*buffer, uint32\_t length)  
*Sends data in the background.*
- int **LPUART\_RTOS\_Receive** (lpuart\_rtos\_handle\_t \*handle, uint8\_t \*buffer, uint32\_t length, size\_t \*received)  
*Receives data.*

### 17.7.2 Data Structure Documentation

#### 17.7.2.1 struct **lpuart\_rtos\_config\_t**

##### Data Fields

- **LPUART\_Type \* base**  
*UART base address.*
- **uint32\_t srecclk**  
*UART source clock in Hz.*
- **uint32\_t baudrate**  
*Desired communication speed.*
- **lpuart\_parity\_mode\_t parity**  
*Parity setting.*
- **lpuart\_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.*

### 17.7.3 Function Documentation

17.7.3.1 `int LPUART_RTOS_Init( Ipuart_rtos_handle_t * handle, Ipuart_handle_t * t_handle, const Ipuart_rtos_config_t * cfg )`

Parameters

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

Returns

0 succeed, others failed

#### 17.7.3.2 int LPUART\_RTOS\_Deinit ( *Ipuart\_rtos\_handle\_t \* handle* )

This function deinitializes the LPUART module, sets all register value to the reset value, and releases the resources.

Parameters

|               |                         |
|---------------|-------------------------|
| <i>handle</i> | The RTOS LPUART handle. |
|---------------|-------------------------|

#### 17.7.3.3 int LPUART\_RTOS\_Send ( *Ipuart\_rtos\_handle\_t \* handle, const uint8\_t \* buffer, uint32\_t length* )

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

Parameters

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

#### 17.7.3.4 int LPUART\_RTOS\_Receive ( *Ipuart\_rtos\_handle\_t \* handle, uint8\_t \* buffer, uint32\_t length, size\_t \* received* )

This function receives data from LPUART. It is an synchronous API. If any data is immediately available it is returned immediately and the number of bytes received.

## LPUART FreeRTOS Driver

### Parameters

|                 |                                                                                  |
|-----------------|----------------------------------------------------------------------------------|
| <i>handle</i>   | The RTOS LPUART 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 a variable of size_t where the number of received data is filled. |

# Chapter 18

## PIT: Periodic Interrupt Timer

### 18.1 Overview

The KSDK provides a driver for the Periodic Interrupt Timer (PIT) of Kinetis devices.

### 18.2 Function groups

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

#### 18.2.1 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.

#### 18.2.2 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.

#### 18.2.3 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

### 18.2.4 Status

Provides functions to get and clear the PIT status.

### 18.2.5 Interrupt

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

## 18.3 Typical use case

### 18.3.1 PIT tick example

Updates the PIT period and toggles an LED periodically.

```
int main(void)
{
 /* Structure of initialize PIT */
 pit_config_t pitConfig;

 /* Initialize and enable LED */
 LED_INIT();

 /* Board pin, clock, debug console init */
 BOARD_InitHardware();

 PIT_GetDefaultConfig(&pitConfig);

 /* Init pit module */
 PIT_Init(PIT, &pitConfig);

 /* Set timer period for channel 0 */
 PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(1000000U,
 PIT_SOURCE_CLOCK));

 /* Enable timer interrupts for channel 0 */
 PIT_EnableInterrupts(PIT, kPIT_Chnl_0,
 kPIT_TimerInterruptEnable);

 /* Enable at the NVIC */
 EnableIRQ(PIT IRQ_ID);

 /* Start channel 0 */
 PRINTF("\r\nStarting channel No.0 ...");
 PIT_StartTimer(PIT, kPIT_Chnl_0);

 while (true)
 {
 /* Check whether occur interrupt and toggle LED */
 if (true == pitIsrFlag)
 {
 PRINTF("\r\n Channel No.0 interrupt is occurred !");
 LED_TOGGLE();
 pitIsrFlag = false;
 }
 }
}
```

## Data Structures

- struct [pit\\_config\\_t](#)  
*PIT configuration structure.* [More...](#)

## Enumerations

- enum [pit\\_chnl\\_t](#) {
   
kPIT\_Chnl\_0 = 0U,
   
kPIT\_Chnl\_1,
   
kPIT\_Chnl\_2,
   
kPIT\_Chnl\_3
 }
   
*List of PIT channels.*
  - enum [pit\\_interrupt\\_enable\\_t](#) { kPIT\_TimerInterruptEnable = PIT\_TCTRL\_TIE\_MASK }
  - enum [pit\\_status\\_flags\\_t](#) { kPIT\_TimerFlag = PIT\_TFLG\_TIF\_MASK }
- List of PIT interrupts.*
- List of PIT status flags.*

## 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)`  
*Ungates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.*
- `void PIT_Deinit (PIT_Type *base)`  
*Gates the PIT clock and disables the PIT module.*
- `static void PIT_GetDefaultConfig (pit_config_t *config)`  
*Fills in the PIT configuration structure with the default settings.*
- `static void PIT_SetTimerChainMode (PIT_Type *base, pit_chnl_t channel, bool enable)`  
*Enables or disables chaining a timer with the previous timer.*

## Interrupt Interface

- `static void PIT_EnableInterrupts (PIT_Type *base, pit_chnl_t channel, uint32_t mask)`  
*Enables the selected PIT interrupts.*
- `static void PIT_DisableInterrupts (PIT_Type *base, pit_chnl_t channel, uint32_t mask)`  
*Disables the selected PIT interrupts.*
- `static uint32_t PIT_GetEnabledInterrupts (PIT_Type *base, pit_chnl_t channel)`  
*Gets the enabled PIT interrupts.*

## 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.*

## 18.4 Data Structure Documentation

### 18.4.1 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*

## 18.5 Enumeration Type Documentation

### 18.5.1 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.

### 18.5.2 enum pit\_interrupt\_enable\_t

Enumerator

***kPIT\_TimerInterruptEnable*** Timer interrupt enable.

### 18.5.3 enum pit\_status\_flags\_t

Enumerator

***kPIT\_TimerFlag*** Timer flag.

## 18.6 Function Documentation

### 18.6.1 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 |

### 18.6.2 void PIT\_Deinit ( PIT\_Type \* *base* )

Parameters

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

### 18.6.3 static void PIT\_GetDefaultConfig ( pit\_config\_t \* *config* ) [inline], [static]

The default values are as follows.

```
* config->enableRunInDebug = false;
*
```

## Function Documentation

Parameters

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

### 18.6.4 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. |

### 18.6.5 static void PIT\_EnableInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|                |                                                                                                                     |
|----------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                         |
| <i>channel</i> | Timer channel number                                                                                                |
| <i>mask</i>    | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">pit_interrupt_enable_t</a> |

### 18.6.6 static void PIT\_DisableInterrupts ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

Parameters

|                |                                                                                                                      |
|----------------|----------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                          |
| <i>channel</i> | Timer channel number                                                                                                 |
| <i>mask</i>    | The interrupts to disable. This is a logical OR of members of the enumeration <a href="#">pit_interrupt_enable_t</a> |

### 18.6.7 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](#)

### 18.6.8 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](#)

### 18.6.9 static void PIT\_ClearStatusFlags ( **PIT\_Type** \* *base*, **pit\_chnl\_t** *channel*, **uint32\_t** *mask* ) [inline], [static]

## Function Documentation

Parameters

|                |                                                                                                                  |
|----------------|------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | PIT peripheral base address                                                                                      |
| <i>channel</i> | Timer channel number                                                                                             |
| <i>mask</i>    | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">pit_status_flags_t</a> |

### **18.6.10 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 |

### **18.6.11 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

#### 18.6.12 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.       |

#### 18.6.13 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.       |

#### 18.6.14 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 19

## PMC: Power Management Controller

### 19.1 Overview

The KSDK provides a Peripheral driver for the Power Management Controller (PMC) module of Kinetis 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.*

## 19.2 Data Structure Documentation

### 19.2.1 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.*

### 19.2.2 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.*

### 19.2.3 struct pmc\_bandgap\_buffer\_config\_t

#### Data Fields

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

*Enable bandgap buffer in low-power mode.*

#### 19.2.3.0.0.13 Field Documentation

19.2.3.0.0.13.1 `bool pmc_bandgap_buffer_config_t::enable`

19.2.3.0.0.13.2 `bool pmc_bandgap_buffer_config_t::enableInLowPowerMode`

### 19.3 Macro Definition Documentation

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

Version 2.0.0.

### 19.4 Enumeration Type Documentation

19.4.1 `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 )

19.4.2 `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)

### 19.5 Function Documentation

19.5.1 `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. |

### 19.5.2 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

### 19.5.3 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. |
|-------------|------------------------------|

### 19.5.4 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. |

### 19.5.5 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.

### 19.5.6 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. |
|-------------|------------------------------|

### 19.5.7 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 |

### 19.5.8 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.

### 19.5.9 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. |
|-------------|------------------------------|

### 19.5.10 static bool PMC\_IsRegulatorInRunRegulation ( **PMC\_Type** \* *base* ) [**inline**], [**static**]

This function returns the regulator to run a regulation status. It provides the current status of the internal voltage regulator.

## Parameters

|             |                                        |
|-------------|----------------------------------------|
| <i>base</i> | PMC peripheral base address.           |
| <i>base</i> | Base address for current PMC instance. |

## Returns

Regulation status 0 - Regulator is in a stop regulation or in transition to/from the regulation. 1 - Regulator is in a run regulation.

## Function Documentation

# Chapter 20

## PORT: Port Control and Interrupts

### 20.1 Overview

The KSDK provides a driver for the Port Control and Interrupts (PORT) module of Kinetis devices.

### 20.2 Typical configuration use case

#### 20.2.1 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);
```

#### 20.2.2 I2C PORT Configuration

```
/* I2C pin PORTconfiguration */
port_pin_config_t config = {
 kPORT_PullUp,
 kPORT_FastSlewRate,
 kPORT_PassiveFilterDisable,
 kPORT_OpenDrainEnable,
 kPORT_LowDriveStrength,
 kPORT_MuxAlt5,
 kPORT_UnLockRegister,
};

PORT_SetPinConfig(PORTE, 24u, &config);
PORT_SetPinConfig(PORTE, 25u, &config);
```

## Data Structures

- struct `port_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.*

## 20.3 Data Structure Documentation

### 20.3.1 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.*

## 20.4 Macro Definition Documentation

### 20.4.1 #define [FSL\\_PORT\\_DRIVER\\_VERSION](#) (MAKE\_VERSION(2, 0, 2))

## 20.5 Enumeration Type Documentation

### 20.5.1 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

### 20.5.2 enum \_port\_slew\_rate

Enumerator

***kPORT\_FastSlewRate*** Fast slew rate is configured.

***kPORT\_SlowSlewRate*** Slow slew rate is configured.

### 20.5.3 enum \_port\_passive\_filter\_enable

Enumerator

***kPORT\_PassiveFilterDisable*** Passive input filter is disabled.

***kPORT\_PassiveFilterEnable*** Passive input filter is enabled.

### 20.5.4 enum \_port\_drive\_strength

Enumerator

***kPORT\_LowDriveStrength*** Low-drive strength is configured.

***kPORT\_HighDriveStrength*** High-drive strength is configured.

### 20.5.5 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.

## 20.5.6 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.

## 20.6 Function Documentation

### 20.6.1 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. |

### 20.6.2 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. |

### 20.6.3 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"><li>• <a href="#">kPORT_PinDisabledOrAnalog</a>: Pin disabled or work in analog function.</li><li>• <a href="#">kPORT_MuxAsGpio</a> : Set as GPIO.</li><li>• <a href="#">kPORT_MuxAlt2</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt3</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt4</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt5</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt6</a> : chip-specific.</li><li>• <a href="#">kPORT_MuxAlt7</a> : chip-specific. : This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : <a href="#">kPORT_PinDisabledOrAnalog</a>). This function is recommended to use to reset the pin mux</li></ul> |

### 20.6.4 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> | <p>PORT pin interrupt configuration.</p> <ul style="list-style-type: none"> <li>• <a href="#">kPORT_InterruptOrDMADisabled</a>: Interrupt/DMA request disabled.</li> <li>• <a href="#">kPORT_DMARisingEdge</a> : DMA request on rising edge(if the DMA requests exit).</li> <li>• <a href="#">kPORT_DMAPFallingEdge</a>: DMA request on falling edge(if the DMA requests exit).</li> <li>• <a href="#">kPORT_DMAEitherEdge</a> : DMA request on either edge(if the DMA requests exit).</li> <li>• #<a href="#">kPORT_FlagRisingEdge</a> : Flag sets on rising edge(if the Flag states exit).</li> <li>• #<a href="#">kPORT_FlagFallingEdge</a> : Flag sets on falling edge(if the Flag states exit).</li> <li>• #<a href="#">kPORT_FlagEitherEdge</a> : Flag sets on either edge(if the Flag states exit).</li> <li>• <a href="#">kPORT_InterruptLogicZero</a> : Interrupt when logic zero.</li> <li>• <a href="#">kPORT_InterruptRisingEdge</a> : Interrupt on rising edge.</li> <li>• <a href="#">kPORT_InterruptFallingEdge</a>: Interrupt on falling edge.</li> <li>• <a href="#">kPORT_InterruptEitherEdge</a> : Interrupt on either edge.</li> <li>• <a href="#">kPORT_InterruptLogicOne</a> : Interrupt when logic one.</li> <li>• #<a href="#">kPORT_ActiveHighTriggerOutputEnable</a> : Enable active high-trigger output (if the trigger states exit).</li> <li>• #<a href="#">kPORT_ActiveLowTriggerOutputEnable</a> : Enable active low-trigger output (if the trigger states exit).</li> </ul> |

## 20.6.5 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.

## Function Documentation

20.6.6 **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.        |

## Function Documentation

# Chapter 21

## RCM: Reset Control Module Driver

### 21.1 Overview

The KSDK provides a Peripheral driver for the Reset Control Module (RCM) module of Kinetis 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_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_SourceMdmap` = 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.*
- enum `rcm_boot_rom_config_t`{  
  `kRCM_BootFlash` = 0U,  
  `kRCM_BootRomCfg0` = 1U,  
  `kRCM_BootRomFopt` = 2U,  
  `kRCM_BootRomBoth` = 3U }  
    *Boot from ROM configuration.*

### Driver version

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

### Reset Control Module APIs

- static `uint32_t RCM_GetPreviousResetSources` (`RCM_Type *base`)

## Enumeration Type Documentation

- Gets the reset source status which caused a previous reset.
  - static uint32\_t [RCM\\_GetStickyResetSources](#) (RCM\_Type \*base)  
Gets the sticky reset source status.
  - static void [RCM\\_ClearStickyResetSources](#) (RCM\_Type \*base, uint32\_t sourceMasks)  
Clears the sticky reset source status.
- void [RCM\\_ConfigureResetPinFilter](#) (RCM\_Type \*base, const [rcm\\_reset\\_pin\\_filter\\_config\\_t](#) \*config)  
Configures the reset pin filter.
- static [rcm\\_boot\\_rom\\_config\\_t](#) [RCM\\_GetBootRomSource](#) (RCM\_Type \*base)  
Gets the ROM boot source.
- static void [RCM\\_ClearBootRomSource](#) (RCM\_Type \*base)  
Clears the ROM boot source flag.
- void [RCM\\_SetForceBootRomSource](#) (RCM\_Type \*base, [rcm\\_boot\\_rom\\_config\\_t](#) config)  
Forces the boot from ROM.

## 21.2 Data Structure Documentation

### 21.2.1 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.

#### 21.2.1.0.0.14 Field Documentation

##### 21.2.1.0.0.14.1 bool rcm\_reset\_pin\_filter\_config\_t::enableFilterInStop

##### 21.2.1.0.0.14.2 rcm\_run\_wait\_filter\_mode\_t rcm\_reset\_pin\_filter\_config\_t::filterInRunWait

##### 21.2.1.0.0.14.3 uint8\_t rcm\_reset\_pin\_filter\_config\_t::busClockFilterCount

## 21.3 Macro Definition Documentation

### 21.3.1 #define FSL\_RCM\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 1))

## 21.4 Enumeration Type Documentation

### 21.4.1 enum rcm\_reset\_source\_t

Enumerator

*kRCM\_SourceWakeup* Low-leakage wakeup reset.

*kRCM\_SourceLvd* Low-voltage detect 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\_SourceMdmap*** MDM-AP system reset.

***kRCM\_SourceSackerr*** Parameter could get all reset flags.

## 21.4.2 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.

## 21.4.3 enum rcm\_boot\_rom\_config\_t

Enumerator

***kRCM\_BootFlash*** Boot from flash.

***kRCM\_BootRomCfg0*** Boot from boot ROM due to BOOTCFG0.

***kRCM\_BootRomFopt*** Boot from boot ROM due to FOPT[7].

***kRCM\_BootRomBoth*** Boot from boot ROM due to both BOOTCFG0 and FOPT[7].

## 21.5 Function Documentation

### 21.5.1 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);
```

## Function Documentation

Parameters

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

Returns

All reset source status bit map.

### 21.5.2 static uint32\_t RCM\_GetStickyResetSources ( RCM\_Type \* *base* ) [inline], [static]

This function gets the current reset source status that has not been cleared by software for a specific source.  
This is an example.

```
uint32_t resetStatus;

// To get all reset source statuses.
resetStatus = RCM_GetStickyResetSources(RCM) & kRCM_SourceAll;

// To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetStickyResetSources(RCM) &
 kRCM_SourceWdog;

// To test multiple reset sources.
resetStatus = RCM_GetStickyResetSources(RCM) & (
 kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

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

Returns

All reset source status bit map.

### 21.5.3 static void RCM\_ClearStickyResetSources ( RCM\_Type \* *base*, uint32\_t *sourceMasks* ) [inline], [static]

This function clears the sticky system reset flags indicated by source masks.

This is an example.

```
// Clears multiple reset sources.
RCM_ClearStickyResetSources(kRCM_SourceWdog |
 kRCM_SourcePin);
```

Parameters

|                    |                              |
|--------------------|------------------------------|
| <i>base</i>        | RCM peripheral base address. |
| <i>sourceMasks</i> | reset source status bit map  |

#### 21.5.4 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.

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | RCM peripheral base address.            |
| <i>config</i> | Pointer to the configuration structure. |

#### 21.5.5 static rcm\_boot\_rom\_config\_t RCM\_GetBootRomSource ( RCM\_Type \* *base* ) [inline], [static]

This function gets the ROM boot source during the last chip reset.

Parameters

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

Returns

The ROM boot source.

#### 21.5.6 static void RCM\_ClearBootRomSource ( RCM\_Type \* *base* ) [inline], [static]

This function clears the ROM boot source flag.

Parameters

---

## Function Documentation

|             |                              |
|-------------|------------------------------|
| <i>base</i> | Register base address of RCM |
|-------------|------------------------------|

### 21.5.7 void RCM\_SetForceBootRomSource ( RCM\_Type \* *base*, rcm\_boot\_rom\_config\_t *config* )

This function forces booting from ROM during all subsequent system resets.

Parameters

|               |                              |
|---------------|------------------------------|
| <i>base</i>   | RCM peripheral base address. |
| <i>config</i> | Boot configuration.          |

# Chapter 22

## RTC: Real Time Clock

### 22.1 Overview

The KSDK provides a driver for the Real Time Clock (RTC) of Kinetis devices.

### 22.2 Function groups

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

#### 22.2.1 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.

#### 22.2.2 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.

#### 22.2.3 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

### 22.2.4 Start & Stop timer

The function [RTC\\_StartTimer\(\)](#) starts the RTC time counter.

The function [RTC\\_StopTimer\(\)](#) stops the RTC time counter.

### 22.2.5 Status

Provides functions to get and clear the RTC status.

### 22.2.6 Interrupt

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

### 22.2.7 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.

### 22.2.8 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.

## 22.3 Typical use case

### 22.3.1 RTC tick example

Example to set the RTC current time and trigger an alarm.

```
int main(void)
{
 uint32_t sec;
 uint32_t currSeconds;
 rtc_datetime_t date;
 rtc_config_t rtcConfig;

 /* Board pin, clock, debug console init */

 /* Set the RTC current time and trigger an alarm */

 /* ... */

 /* Main loop */
 while(1)
 {
 /* ... */
 }
}
```

```

BOARD_InitHardware();
/* Init RTC */
RTC_GetDefaultConfig(&rtcConfig);
RTC_Init(RTC, &rtcConfig);
/* Select RTC clock source */
BOARD_SetRtcClockSource();

PRINTF("RTC example: set up time to wake up an alarm\r\n");

/* Set a start date time and start RT */
date.year = 2014U;
date.month = 12U;
date.day = 25U;
date.hour = 19U;
date.minute = 0;
date.second = 0;

/* RTC time counter has to be stopped before setting the date & time in the TSR register */
RTC_StopTimer(RTC);

/* Set RTC time to default */
RTC_SetDatetime(RTC, &date);

/* Enable RTC alarm interrupt */
RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);

/* Enable at the NVIC */
EnableIRQ(RTC IRQn);

/* Start the RTC time counter */
RTC_StartTimer(RTC);

/* This loop will set the RTC alarm */
while (1)
{
 busyWait = true;
 /* Get date time */
 RTC_GetDatetime(RTC, &date);

 /* print default time */
 PRINTF("Current datetime: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day, date.hour,
 date.minute, date.second);

 /* Get alarm time from the user */
 sec = 0;
 PRINTF("Input the number of second to wait for alarm \r\n");
 PRINTF("The second must be positive value\r\n");
 while (sec < 1)
 {
 SCANF("%d", &sec);
 }

 /* Read the RTC seconds register to get current time in seconds */
 currSeconds = RTC->TSR;

 /* Add alarm seconds to current time */
 currSeconds += sec;

 /* Set alarm time in seconds */
 RTC->TAR = currSeconds;

 /* Get alarm time */
 RTC_GetAlarm(RTC, &date);

 /* Print alarm time */
 PRINTF("Alarm will occur at: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day,

```

## Typical use case

```
 date.hour, date.minute, date.second);

 /* Wait until alarm occurs */
 while (busyWait)
 {
 }

 PRINTF("\r\n Alarm occurs !!!! ");
}
}
```

## Data Structures

- struct `rtc_datetime_t`  
*Structure is used to hold the date and time.* [More...](#)
- struct `rtc_config_t`  
*RTC config structure.* [More...](#)

## Enumerations

- enum `rtc_interrupt_enable_t` {  
    kRTC\_TimeInvalidInterruptEnable = RTC\_IER\_TIE\_MASK,  
    kRTC\_TimeOverflowInterruptEnable = RTC\_IER\_TOIE\_MASK,  
    kRTC\_AlarmInterruptEnable = RTC\_IER\_TAIE\_MASK,  
    kRTC\_SecondsInterruptEnable = RTC\_IER\_TSIE\_MASK }  
*List of RTC interrupts.*
- enum `rtc_status_flags_t` {  
    kRTC\_TimeInvalidFlag = RTC\_SR\_TIF\_MASK,  
    kRTC\_TimeOverflowFlag = RTC\_SR\_TOF\_MASK,  
    kRTC\_AlarmFlag = RTC\_SR\_TAF\_MASK }  
*List of RTC flags.*
- enum `rtc_osc_cap_load_t` {  
    kRTC\_Capacitor\_2p = RTC\_CR\_SC2P\_MASK,  
    kRTC\_Capacitor\_4p = RTC\_CR\_SC4P\_MASK,  
    kRTC\_Capacitor\_8p = RTC\_CR\_SC8P\_MASK,  
    kRTC\_Capacitor\_16p = RTC\_CR\_SC16P\_MASK }  
*List of RTC Oscillator capacitor load settings.*

## Functions

- static void `RTC_SetOscCapLoad` (RTC\_Type \*base, uint32\_t capLoad)  
*This function sets the specified capacitor configuration for the RTC oscillator.*
- static void `RTC_Reset` (RTC\_Type \*base)  
*Performs a software reset on the RTC module.*

## Driver version

- #define `FSL_RTC_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 0))  
*Version 2.0.0.*

## Initialization and deinitialization

- void [RTC\\_Init](#) (RTC\_Type \*base, const [rtc\\_config\\_t](#) \*config)  
*Ungates the RTC clock and configures the peripheral for basic operation.*
- static void [RTC\\_Deinit](#) (RTC\_Type \*base)  
*Stops the timer and gate the RTC clock.*
- void [RTC\\_GetDefaultConfig](#) ([rtc\\_config\\_t](#) \*config)  
*Fills in the RTC config struct with the default settings.*

## Current Time & Alarm

- status\_t [RTC\\_SetDatetime](#) (RTC\_Type \*base, const [rtc\\_datetime\\_t](#) \*datetime)  
*Sets the RTC date and time according to the given time structure.*
- void [RTC\\_GetDatetime](#) (RTC\_Type \*base, [rtc\\_datetime\\_t](#) \*datetime)  
*Gets the RTC time and stores it in the given time structure.*
- status\_t [RTC\\_SetAlarm](#) (RTC\_Type \*base, const [rtc\\_datetime\\_t](#) \*alarmTime)  
*Sets the RTC alarm time.*
- void [RTC\\_GetAlarm](#) (RTC\_Type \*base, [rtc\\_datetime\\_t](#) \*datetime)  
*Returns the RTC alarm time.*

## Interrupt Interface

- static void [RTC\\_EnableInterrupts](#) (RTC\_Type \*base, uint32\_t mask)  
*Enables the selected RTC interrupts.*
- static void [RTC\\_DisableInterrupts](#) (RTC\_Type \*base, uint32\_t mask)  
*Disables the selected RTC interrupts.*
- static uint32\_t [RTC\\_GetEnabledInterrupts](#) (RTC\_Type \*base)  
*Gets the enabled RTC interrupts.*

## Status Interface

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

## Timer Start and Stop

- static void [RTC\\_StartTimer](#) (RTC\_Type \*base)  
*Starts the RTC time counter.*
- static void [RTC\\_StopTimer](#) (RTC\_Type \*base)  
*Stops the RTC time counter.*

## 22.4 Data Structure Documentation

### 22.4.1 struct [rtc\\_datetime\\_t](#)

#### Data Fields

- uint16\_t year

## Data Structure Documentation

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

### 22.4.1.0.0.15 Field Documentation

#### 22.4.1.0.0.15.1 `uint16_t rtc_datetime_t::year`

#### 22.4.1.0.0.15.2 `uint8_t rtc_datetime_t::month`

#### 22.4.1.0.0.15.3 `uint8_t rtc_datetime_t::day`

#### 22.4.1.0.0.15.4 `uint8_t rtc_datetime_t::hour`

#### 22.4.1.0.0.15.5 `uint8_t rtc_datetime_t::minute`

#### 22.4.1.0.0.15.6 `uint8_t rtc_datetime_t::second`

## 22.4.2 `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.*

## 22.5 Enumeration Type Documentation

### 22.5.1 enum rtc\_interrupt\_enable\_t

Enumerator

- kRTC\_TimeInvalidInterruptEnable* Time invalid interrupt.
- kRTC\_TimeOverflowInterruptEnable* Time overflow interrupt.
- kRTC\_AlarmInterruptEnable* Alarm interrupt.
- kRTC\_SecondsInterruptEnable* Seconds interrupt.

### 22.5.2 enum rtc\_status\_flags\_t

Enumerator

- kRTC\_TimeInvalidFlag* Time invalid flag.
- kRTC\_TimeOverflowFlag* Time overflow flag.
- kRTC\_AlarmFlag* Alarm flag.

### 22.5.3 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

## 22.6 Function Documentation

### 22.6.1 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. |

### 22.6.2 static void RTC\_Deinit ( RTC\_Type \* *base* ) [inline], [static]

Parameters

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

### 22.6.3 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. |
|---------------|----------------------------------------------------|

### 22.6.4 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

## 22.6.5 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. |

## 22.6.6 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

## 22.6.7 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. |

**22.6.8 static void RTC\_EnableInterrupts ( RTC\_Type \* *base*, uint32\_t *mask* )  
[inline], [static]**

Parameters

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | RTC peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">rtc_interrupt_enable_t</a> |

**22.6.9 static void RTC\_DisableInterrupts ( RTC\_Type \* *base*, uint32\_t *mask* )  
[inline], [static]**

Parameters

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | RTC peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">rtc_interrupt_enable_t</a> |

**22.6.10 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](#)

**22.6.11 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](#)

### 22.6.12 void RTC\_ClearStatusFlags ( RTC\_Type \* *base*, uint32\_t *mask* )

Parameters

|             |                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | RTC peripheral base address                                                                                      |
| <i>mask</i> | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">rtc_status_flags_t</a> |

### 22.6.13 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 |
|-------------|-----------------------------|

### 22.6.14 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 |
|-------------|-----------------------------|

### 22.6.15 static void RTC\_SetOscCapLoad ( RTC\_Type \* *base*, uint32\_t *capLoad* ) [inline], [static]

## Function Documentation

Parameters

|                |                                                                                                                    |
|----------------|--------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | RTC peripheral base address                                                                                        |
| <i>capLoad</i> | Oscillator loads to enable. This is a logical OR of members of the enumeration <a href="#">rtc_-osc_cap_load_t</a> |

### 22.6.16 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 23

## SAI: Serial Audio Interface

### 23.1 Overview

The KSDK provides a peripheral driver for the Serial Audio Interface (SAI) module of Kinetis devices.

SAI driver includes functional APIs and transactional APIs.

Functional APIs target low-level APIs. Functional APIs can be used for SAI initialization/configuration/operation for optimization/customization purpose. 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.

### 23.2 Typical use case

#### 23.2.1 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)
{
 //...
 SAI_TxGetDefaultConfig(&user_config);
```

## Typical use case

```
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)
{
}

// ...
}
```

### 23.2.2 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) }
- `SAI return status.`

## Typical use case

- enum `sai_protocol_t` {  
    `kSAI_BusLeftJustified` = 0x0U,  
    `kSAI_BusRightJustified`,  
    `kSAI_BusI2S`,  
    `kSAI_BusPCMA`,  
    `kSAI_BusPCMB` }  
    *Define the SAI bus type.*
- enum `sai_master_slave_t` {  
    `kSAI_Master` = 0x0U,  
    `kSAI_Slave` = 0x1U }  
    *Master or slave mode.*
- enum `sai_mono_stereo_t` {  
    `kSAI_Stereo` = 0x0U,  
    `kSAI_MonoLeft`,  
    `kSAI_MonoRight` }  
    *Mono or stereo audio format.*
- enum `sai_sync_mode_t` {  
    `kSAI_ModeAsync` = 0x0U,  
    `kSAI_ModeSync`,  
    `kSAI_ModeSyncWithOtherTx`,  
    `kSAI_ModeSyncWithOtherRx` }  
    *Synchronous or asynchronous mode.*
- enum `sai_mclk_source_t` {  
    `kSAI_MclkSourceSysclk` = 0x0U,  
    `kSAI_MclkSourceSelect1`,  
    `kSAI_MclkSourceSelect2`,  
    `kSAI_MclkSourceSelect3` }  
    *Mater clock source.*
- enum `sai_bclk_source_t` {  
    `kSAI_BclkSourceBusclk` = 0x0U,  
    `kSAI_BclkSourceMclkDiv`,  
    `kSAI_BclkSourceOtherSai0`,  
    `kSAI_BclkSourceOtherSai1` }  
    *Bit clock source.*
- enum `_sai_interrupt_enable_t` {  
    `kSAI_WordStartInterruptEnable`,  
    `kSAI_SyncErrorInterruptEnable` = I2S\_TCSR\_SEIE\_MASK,  
    `kSAI_FIFOWarningInterruptEnable` = I2S\_TCSR\_FWIE\_MASK,  
    `kSAI_FIFOErrorInterruptEnable` = I2S\_TCSR\_FEIE\_MASK }  
    *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 DMA request sources.*

- 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_fifo_packing_t` {
   
    `kSAI_FifoPackingDisabled` = 0x0U,
   
    `kSAI_FifoPacking8bit` = 0x2U,
   
    `kSAI_FifoPacking16bit` = 0x3U }
- The SAI packing mode The mode includes 8 bit and 16 bit packing.
- 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, 1))
- Version 2.1.1.

## Initialization and deinitialization

- void `SAI_TxInit` (I2S\_Type \*base, const `sai_config_t` \*config)
   
    Initializes the SAI Tx peripheral.
- void `SAI_RxInit` (I2S\_Type \*base, const `sai_config_t` \*config)
   
    Initializes the the SAI Rx peripheral.
- void `SAI_TxGetDefaultConfig` (`sai_config_t` \*config)
   
    Sets the SAI Tx configuration structure to default values.
- void `SAI_RxGetDefaultConfig` (`sai_config_t` \*config)
   
    Sets the SAI Rx configuration structure to default values.
- void `SAI_Deinit` (I2S\_Type \*base)
   
    De-initializes the SAI peripheral.
- void `SAI_TxReset` (I2S\_Type \*base)

## Typical use case

- Resets the SAI Tx.  
void **SAI\_RxReset** (I2S\_Type \*base)  
*Resets the SAI Rx.*
- void **SAI\_TxEnable** (I2S\_Type \*base, bool enable)  
*Enables/disables the SAI Tx.*
- void **SAI\_RxEnable** (I2S\_Type \*base, bool enable)  
*Enables/disables the SAI Rx.*

## Status

- static uint32\_t **SAI\_TxGetStatusFlag** (I2S\_Type \*base)  
*Gets the SAI Tx status flag state.*
- static void **SAI\_TxClearStatusFlags** (I2S\_Type \*base, uint32\_t mask)  
*Clears the SAI Tx status flag state.*
- static uint32\_t **SAI\_RxGetStatusFlag** (I2S\_Type \*base)  
*Gets the SAI Rx status flag state.*
- static void **SAI\_RxClearStatusFlags** (I2S\_Type \*base, uint32\_t mask)  
*Clears the SAI Rx status flag state.*

## Interrupts

- static void **SAI\_TxEnableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
*Enables the SAI Tx interrupt requests.*
- static void **SAI\_RxEnableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
*Enables the SAI Rx interrupt requests.*
- static void **SAI\_TxDisableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
*Disables the SAI Tx interrupt requests.*
- static void **SAI\_RxDisableInterrupts** (I2S\_Type \*base, uint32\_t mask)  
*Disables the SAI Rx interrupt requests.*

## DMA Control

- static void **SAI\_TxEnableDMA** (I2S\_Type \*base, uint32\_t mask, bool enable)  
*Enables/disables the SAI Tx DMA requests.*
- static void **SAI\_RxEnableDMA** (I2S\_Type \*base, uint32\_t mask, bool enable)  
*Enables/disables the SAI Rx DMA requests.*
- static uint32\_t **SAI\_TxGetDataRegisterAddress** (I2S\_Type \*base, uint32\_t channel)  
*Gets the SAI Tx data register address.*
- static uint32\_t **SAI\_RxGetDataRegisterAddress** (I2S\_Type \*base, uint32\_t channel)  
*Gets the SAI Rx data register address.*

## Bus Operations

- void **SAI\_TxSetFormat** (I2S\_Type \*base, sai\_transfer\_format\_t \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
*Configures the SAI Tx audio format.*
- void **SAI\_RxSetFormat** (I2S\_Type \*base, sai\_transfer\_format\_t \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)  
*Configures the SAI Rx audio format.*
- void **SAI\_WriteBlocking** (I2S\_Type \*base, uint32\_t channel, uint32\_t bitWidth, uint8\_t \*buffer, uint32\_t size)

- static void [SAI\\_WriteData](#) (I2S\_Type \*base, uint32\_t channel, uint32\_t data)
 

*Sends data using a blocking method.*

*Writes data into SAI FIFO.*
- void [SAI\\_ReadBlocking](#) (I2S\_Type \*base, uint32\_t channel, uint32\_t bitWidth, uint8\_t \*buffer, uint32\_t size)
 

*Receives data using a blocking method.*

*Reads data from the SAI FIFO.*

## Transactional

- void [SAI\\_TransferTxCreateHandle](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_callback\\_t](#) callback, void \*userData)
 

*Initializes the SAI Tx handle.*
- void [SAI\\_TransferRxCreateHandle](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_callback\\_t](#) callback, void \*userData)
 

*Initializes the SAI Rx handle.*
- status\_t [SAI\\_TransferTxSetFormat](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_format\\_t](#) \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)
 

*Configures the SAI Tx audio format.*
- status\_t [SAI\\_TransferRxSetFormat](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_format\\_t](#) \*format, uint32\_t mclkSourceClockHz, uint32\_t bclkSourceClockHz)
 

*Configures the SAI Rx audio format.*
- status\_t [SAI\\_TransferSendNonBlocking](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_t](#) \*xfer)
 

*Performs an interrupt non-blocking send transfer on SAI.*
- status\_t [SAI\\_TransferReceiveNonBlocking](#) (I2S\_Type \*base, sai\_handle\_t \*handle, [sai\\_transfer\\_t](#) \*xfer)
 

*Performs an interrupt non-blocking receive transfer on SAI.*
- status\_t [SAI\\_TransferGetSendCount](#) (I2S\_Type \*base, sai\_handle\_t \*handle, size\_t \*count)
 

*Gets a set byte count.*
- status\_t [SAI\\_TransferGetReceiveCount](#) (I2S\_Type \*base, sai\_handle\_t \*handle, size\_t \*count)
 

*Gets a received byte count.*
- void [SAI\\_TransferAbortSend](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Aborts the current send.*
- void [SAI\\_TransferAbortReceive](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Aborts the the current IRQ receive.*
- void [SAI\\_TransferTxHandleIRQ](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Tx interrupt handler.*
- void [SAI\\_TransferRxHandleIRQ](#) (I2S\_Type \*base, sai\_handle\_t \*handle)
 

*Tx interrupt handler.*

## 23.3 Data Structure Documentation

### 23.3.1 struct sai\_config\_t

#### Data Fields

- [sai\\_protocol\\_t](#) protocol

## Data Structure Documentation

- *Audio bus protocol in SAI.*
  - **sai\_sync\_mode\_t syncMode**  
*SAI sync mode, control Tx/Rx clock sync.*
  - **bool mclkOutputEnable**  
*Master clock output enable, true means master clock divider enabled.*
- **sai\_mclk\_source\_t mclkSource**  
*Master Clock source.*
- **sai\_bclk\_source\_t bclkSource**  
*Bit Clock source.*
- **sai\_master\_slave\_t masterSlave**  
*Master or slave.*

### 23.3.2 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.*

#### 23.3.2.0.0.16 Field Documentation

##### 23.3.2.0.0.16.1 uint8\_t sai\_transfer\_format\_t::channel

### 23.3.3 struct sai\_transfer\_t

#### Data Fields

- **uint8\_t \* data**  
*Data start address to transfer.*
- **size\_t dataSize**  
*Transfer size.*

### 23.3.3.0.0.17 Field Documentation

23.3.3.0.0.17.1 `uint8_t* sai_transfer_t::data`

23.3.3.0.0.17.2 `size_t sai_transfer_t::dataSize`

### 23.3.4 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.*

### 23.4 Macro Definition Documentation

#### 23.4.1 #define SAI\_XFER\_QUEUE\_SIZE (4)

### 23.5 Enumeration Type Documentation

#### 23.5.1 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

### 23.5.2 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.

### 23.5.3 enum sai\_master\_slave\_t

Enumerator

- kSAI\_Master*** Master mode.
- kSAI\_Slave*** Slave mode.

### 23.5.4 enum sai\_mono\_stereo\_t

Enumerator

- kSAI\_Stereo*** Stereo sound.
- kSAI\_MonoLeft*** Only left channel have sound.
- kSAI\_MonoRight*** Only Right channel have sound.

### 23.5.5 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.

### 23.5.6 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.

### 23.5.7 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.

### 23.5.8 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.

### 23.5.9 enum \_sai\_dma\_enable\_t

Enumerator

- kSAI\_FIFOWarningDMAEnable* FIFO warning caused by the DMA request.

### 23.5.10 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.

### 23.5.11 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

### 23.5.12 enum sai\_fifo\_packing\_t

Enumerator

*kSAI\_FifoPackingDisabled* Packing disabled.

*kSAI\_FifoPacking8bit* 8 bit packing enabled

*kSAI\_FifoPacking16bit* 16bit packing enabled

### 23.5.13 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.

### 23.5.14 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.

## 23.6 Function Documentation

### 23.6.1 void SAI\_TxInit ( I2S\_Type \* *base*, const sai\_config\_t \* *config* )

Ungates the SAI clock, resets the module, and configures SAI Tx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI\\_TxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAIM module can cause a hard fault because the clock is not enabled.

Parameters

|               |                              |
|---------------|------------------------------|
| <i>base</i>   | SAI base pointer             |
| <i>config</i> | SAI configuration structure. |

### 23.6.2 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. |

### 23.6.3 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 |
|---------------|-------------------------------------------|

### 23.6.4 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 |
|---------------|-------------------------------------------|

### 23.6.5 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 |
|-------------|------------------|

### 23.6.6 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 |
|-------------|------------------|

### 23.6.7 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 |
|-------------|------------------|

### 23.6.8 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. |
|---------------|------------------------------------------------|

**23.6.9 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. |

**23.6.10 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.

**23.6.11 static void SAI\_TxClearStatusFlags ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

|             |                                                                                                                                                                                                        |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | SAI base pointer                                                                                                                                                                                       |
| <i>mask</i> | State mask. It can be a combination of the following source if defined: <ul style="list-style-type: none"><li>• kSAI_WordStartFlag</li><li>• kSAI_SyncErrorFlag</li><li>• kSAI_FIFOErrorFlag</li></ul> |

**23.6.12 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.

### 23.6.13 static void SAI\_RxClearStatusFlags ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | SAI base pointer                                                                                                                                                                                        |
| <i>mask</i> | State mask. It can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_WordStartFlag</li><li>• kSAI_SyncErrorFlag</li><li>• kSAI_FIFOErrorFlag</li></ul> |

### 23.6.14 static void SAI\_TxEnableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                                                                                                                                                                                                                                              |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | SAI base pointer                                                                                                                                                                                                                                                                                                                             |
| <i>mask</i> | interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_WordStartInterruptEnable</li><li>• kSAI_SyncErrorInterruptEnable</li><li>• kSAI_FIFOWarningInterruptEnable</li><li>• kSAI_FIFORequestInterruptEnable</li><li>• kSAI_FIFOErrorInterruptEnable</li></ul> |

### 23.6.15 static void SAI\_RxEnableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                                                                                                                                                                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | SAI base pointer                                                                                                                                                                                                                                                                                                                                          |
| <i>mask</i> | <p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> <li>• kSAI_WordStartInterruptEnable</li> <li>• kSAI_SyncErrorInterruptEnable</li> <li>• kSAI_FIFOWarningInterruptEnable</li> <li>• kSAI_FIFORequestInterruptEnable</li> <li>• kSAI_FIFOErrorInterruptEnable</li> </ul> |

### 23.6.16 static void SAI\_TxDisableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                                                                                                                                                                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | SAI base pointer                                                                                                                                                                                                                                                                                                                                          |
| <i>mask</i> | <p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> <li>• kSAI_WordStartInterruptEnable</li> <li>• kSAI_SyncErrorInterruptEnable</li> <li>• kSAI_FIFOWarningInterruptEnable</li> <li>• kSAI_FIFORequestInterruptEnable</li> <li>• kSAI_FIFOErrorInterruptEnable</li> </ul> |

## Function Documentation

**23.6.17 static void SAI\_RxDisableInterrupts ( I2S\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

|             |                                                                                                                                                                                                                                                                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | SAI base pointer                                                                                                                                                                                                                                                                                                                              |
| <i>mask</i> | interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_WordStartInterruptEnable</li><li>• kSAI_SyncErrorInterruptEnable</li><li>• kSAI_FIFOWarningInterruptEnable</li><li>• kSAI_FIFOResponseInterruptEnable</li><li>• kSAI_FIFOErrorInterruptEnable</li></ul> |

**23.6.18 static void SAI\_TxEnableDMA ( I2S\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]**

Parameters

|               |                                                                                                                                                                                                   |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | SAI base pointer                                                                                                                                                                                  |
| <i>mask</i>   | DMA source The parameter can be combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_FIFOWarningDMAEnable</li><li>• kSAI_FIFOResponseDMAEnable</li></ul> |
| <i>enable</i> | True means enable DMA, false means disable DMA.                                                                                                                                                   |

**23.6.19 static void SAI\_RxEnableDMA ( I2S\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]**

Parameters

|               |                                                                                                                                                                                                     |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>   | SAI base pointer                                                                                                                                                                                    |
| <i>mask</i>   | DMA source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"><li>• kSAI_FIFOWarningDMAEnable</li><li>• kSAI_FIFOResponseDMAEnable</li></ul> |
| <i>enable</i> | True means enable DMA, false means disable DMA.                                                                                                                                                     |

**23.6.20 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.

### **23.6.21 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.

### **23.6.22 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. |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------|

### 23.6.23 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. |

### 23.6.24 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.                                     |

### 23.6.25 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. |

**23.6.26 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.                                        |

**23.6.27 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.

**23.6.28 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 |

### 23.6.29 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. |

### 23.6.30 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.

### 23.6.31 **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.

### 23.6.32 **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.        |

**23.6.33 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.        |

**23.6.34 status\_t SAI\_TransferGetSendCount ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, size\_t \* *count* )**

## Function Documentation

Parameters

|               |                                                                        |
|---------------|------------------------------------------------------------------------|
| <i>base</i>   | SAI base pointer.                                                      |
| <i>handle</i> | Pointer to the sai_handle_t structure which stores the transfer state. |
| <i>count</i>  | Bytes count sent.                                                      |

Return values

|                                      |                                                                |
|--------------------------------------|----------------------------------------------------------------|
| <i>kStatus_Success</i>               | Succeed get the transfer count.                                |
| <i>kStatus_NoTransferIn-Progress</i> | There is not a non-blocking transaction currently in progress. |

### 23.6.35 **status\_t SAI\_TransferGetReceiveCount ( I2S\_Type \* *base*, sai\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

|               |                                                                        |
|---------------|------------------------------------------------------------------------|
| <i>base</i>   | SAI base pointer.                                                      |
| <i>handle</i> | Pointer to the sai_handle_t structure which stores the transfer state. |
| <i>count</i>  | Bytes count received.                                                  |

Return values

|                                      |                                                                |
|--------------------------------------|----------------------------------------------------------------|
| <i>kStatus_Success</i>               | Succeed get the transfer count.                                |
| <i>kStatus_NoTransferIn-Progress</i> | There is not a non-blocking transaction currently in progress. |

### 23.6.36 **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. |

### 23.6.37 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. |

### 23.6.38 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. |

### 23.6.39 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. |

### 23.7 SAI DMA Driver

#### 23.7.1 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.*

## 23.7.2 Data Structure Documentation

### 23.7.2.1 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.*

#### 23.7.2.1.0.18 Field Documentation

##### 23.7.2.1.0.18.1 `sai_transfer_t sai_dma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

##### 23.7.2.1.0.18.2 `volatile uint8_t sai_dma_handle_t::queueUser`

## 23.7.3 Function Documentation

### 23.7.3.1 `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. |

**23.7.3.2 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. |

**23.7.3.3 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. |

**23.7.3.4 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. |

**23.7.3.5 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.      |

### 23.7.3.6 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.      |

### 23.7.3.7 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. |

### 23.7.3.8 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. |

### 23.7.3.9 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. |

### 23.7.3.10 status\_t SAI\_TransferGetReceiveCountDMA ( I2S\_Type \* *base*, sai\_dma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

|             |                   |
|-------------|-------------------|
| <i>base</i> | SAI base pointer. |
|-------------|-------------------|

## SAI DMA Driver

|               |                              |
|---------------|------------------------------|
| <i>handle</i> | SAI DMA handle pointer.      |
| <i>count</i>  | Bytes count received by SAI. |

Return values

|                                      |                                                                |
|--------------------------------------|----------------------------------------------------------------|
| <i>kStatus_Success</i>               | Succeed get the transfer count.                                |
| <i>kStatus_NoTransferIn-Progress</i> | There is not a non-blocking transaction currently in progress. |

## 23.8 SAI eDMA Driver

### 23.8.1 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.*

### 23.8.2 Data Structure Documentation

#### 23.8.2.1 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.*

### 23.8.2.1.0.19 Field Documentation

23.8.2.1.0.19.1 `uint8_t sai_edma_handle_t::nbytes`

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

23.8.2.1.0.19.3 `sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`

23.8.2.1.0.19.4 `volatile uint8_t sai_edma_handle_t::queueUser`

### 23.8.3 Function Documentation

23.8.3.1 `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. |

**23.8.3.2 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. |

**23.8.3.3 void SAI\_TransferTxSetFormatEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )**

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

|             |                   |
|-------------|-------------------|
| <i>base</i> | SAI base pointer. |
|-------------|-------------------|

|                          |                                                                                                                                 |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <i>handle</i>            | SAI eDMA handle pointer.                                                                                                        |
| <i>format</i>            | Pointer to SAI audio data format structure.                                                                                     |
| <i>mclkSourceClockHz</i> | SAI master clock source frequency in Hz.                                                                                        |
| <i>bclkSourceClockHz</i> | SAI bit clock source frequency in Hz. If bit clock source is master clock, this value should equals to masterClockHz in format. |

Return values

|                                |                                |
|--------------------------------|--------------------------------|
| <i>kStatus_Success</i>         | Audio format set successfully. |
| <i>kStatus_InvalidArgument</i> | The input argument is invalid. |

**23.8.3.4 void SAI\_TransferRxSetFormatEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_transfer\_format\_t \* *format*, uint32\_t *mclkSourceClockHz*, uint32\_t *bclkSourceClockHz* )**

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

|                          |                                                                                                                                      |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>              | SAI base pointer.                                                                                                                    |
| <i>handle</i>            | SAI eDMA handle pointer.                                                                                                             |
| <i>format</i>            | Pointer to SAI audio data format structure.                                                                                          |
| <i>mclkSourceClockHz</i> | SAI master clock source frequency in Hz.                                                                                             |
| <i>bclkSourceClockHz</i> | SAI bit clock source frequency in Hz. If a bit clock source is the master clock, this value should equal to masterClockHz in format. |

Return values

|                                |                                |
|--------------------------------|--------------------------------|
| <i>kStatus_Success</i>         | Audio format set successfully. |
| <i>kStatus_InvalidArgument</i> | The input argument is invalid. |

**23.8.3.5 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.           |

### 23.8.3.6 **status\_t SAI\_TransferReceiveEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, sai\_transfer\_t \* *xfer* )**

### Note

This interface returns immediately after the transfer initiates. Call the SAI\_GetReceiveRemainingBytes to poll the transfer status and check whether the SAI transfer is finished.

### Parameters

|               |                                    |
|---------------|------------------------------------|
| <i>base</i>   | SAI base pointer                   |
| <i>handle</i> | SAI eDMA handle pointer.           |
| <i>xfer</i>   | Pointer to DMA transfer structure. |

### Return values

|                                |                                        |
|--------------------------------|----------------------------------------|
| <i>kStatus_Success</i>         | Start a SAI eDMA receive successfully. |
| <i>kStatus_InvalidArgument</i> | The input argument is invalid.         |
| <i>kStatus_RxBusy</i>          | SAI is busy receiving data.            |

### 23.8.3.7 **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. |

### 23.8.3.8 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. |

### 23.8.3.9 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. |

### 23.8.3.10 status\_t SAI\_TransferGetReceiveCountEDMA ( I2S\_Type \* *base*, sai\_edma\_handle\_t \* *handle*, size\_t \* *count* )

Parameters

|             |                  |
|-------------|------------------|
| <i>base</i> | SAI base pointer |
|-------------|------------------|

## SAI eDMA Driver

|               |                              |
|---------------|------------------------------|
| <i>handle</i> | SAI eDMA handle pointer.     |
| <i>count</i>  | Bytes count received by SAI. |

Return values

|                                      |                                                   |
|--------------------------------------|---------------------------------------------------|
| <i>kStatus_Success</i>               | Succeed get the transfer count.                   |
| <i>kStatus_NoTransferIn-Progress</i> | There is no non-blocking transaction in progress. |

# Chapter 24

## SIM: System Integration Module Driver

### 24.1 Overview

The KSDK provides a peripheral driver for the System Integration Module (SIM) of Kinetis 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.*

### 24.2 Data Structure Documentation

#### 24.2.1 struct `sim_uid_t`

### Data Fields

- uint32\_t `MH`

## Function Documentation

- *UIDMH.*
- `uint32_t ML`  
*UIDML.*
- `uint32_t L`  
*UIDL.*

### 24.2.1.0.0.20 Field Documentation

24.2.1.0.0.20.1 `uint32_t sim_uid_t::MH`

24.2.1.0.0.20.2 `uint32_t sim_uid_t::ML`

24.2.1.0.0.20.3 `uint32_t sim_uid_t::L`

## 24.3 Enumeration Type Documentation

### 24.3.1 `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.

### 24.3.2 `enum _sim_flash_mode`

Enumerator

*kSIM\_FlashDisableInWait* Disable flash in wait mode.

*kSIM\_FlashDisable* Disable flash in normal mode.

## 24.4 Function Documentation

### 24.4.1 `void SIM_SetUsbVoltRegulatorEnableMode ( uint32_t mask )`

This function configures whether the USB voltage regulator is enabled in normal RUN mode, STOP-/VLPS/LLS/VLLS modes, and VLPR/VLPW modes. The configurations are passed in as mask value of `_sim_usb_volt_reg_enable_mode`. For example, to enable USB voltage regulator in RUN/VLPR/VLPW modes and disable in STOP/VLPS/LLS/VLLS mode, use:

```
SIM_SetUsbVoltRegulatorEnableMode(kSIM_UsbVoltRegEnable | kSIM_UsbVoltRegEnableInLowPower);
```

Parameters

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

#### 24.4.2 void SIM\_GetUniqueId ( sim\_uid\_t \* *uid* )

Parameters

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

#### 24.4.3 static void SIM\_SetFlashMode ( uint8\_t *mode* ) [inline], [static]

Parameters

|             |                                                                        |
|-------------|------------------------------------------------------------------------|
| <i>mode</i> | The mode to set; see <a href="#">_sim_flash_mode</a> for mode details. |
|-------------|------------------------------------------------------------------------|

## Function Documentation

# Chapter 25

## SLCD: Segment LCD Driver

### 25.1 Overview

The KSDK provides a peripheral driver for the Segment LCD (SLCD) module of Kinetis 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\(\)](#).

### 25.2 Typical use case

#### 25.2.1 SLCD Initialization operation

```
slcd_config_t configure = 0;
slcd_clock_config_t clkConfig =
{
 kSLCD_AlternateClk1,
 kSLCD_AltClkDivFactor1,
 kSLCD_ClkPrescaler00
#if 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\_InternalVll3UseChargePump,  
 kSLCD\_ExternalVll3UseResistorBiasNetwork,  
 kSLCD\_ExteranVll3UseChargePump,  
 kSLCD\_InternalVll1UseChargePump }  
*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_BlinkMode }
    SLCD display mode.
• enum slcd_blink_mode_t {
    kSLCD_BlinkDisplayBlink = 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 pinIdx, **slcd_phase_type_t** phase)
 Sets the SLCD back plane pin phase.
- static void **SLCD_SetFrontPlaneSegments** (LCD_Type *base, uint32_t pinIdx, uint8_t operation)
 Sets the SLCD front plane segment operation for a front plane pin.
- static void **SLCD_SetFrontPlaneOnePhase** (LCD_Type *base, uint32_t pinIdx, **slcd_phase_index_t** phaseIdx, 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.

Data Structure Documentation

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.

25.3 Data Structure Documentation

25.3.1 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.

25.3.1.0.0.21 Field Documentation

25.3.1.0.0.21.1 bool slcd_fault_detect_config_t::faultDetectIntEnable

25.3.1.0.0.21.2 bool slcd_fault_detect_config_t::faultDetectBackPlaneEnable

25.3.1.0.0.21.3 uint8_t slcd_fault_detect_config_t::faultDetectPinIndex

25.3.1.0.0.21.4 [slcd_fault_detect_clock_prescaler_t](#) slcd_fault_detect_config_t::faultPrescaler

25.3.1.0.0.21.5 [slcd_fault_detect_sample_window_width_t](#) slcd_fault_detect_config_t::width

25.3.2 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.

25.3.2.0.0.22 Field Documentation

25.3.2.0.0.22.1 **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.

25.3.2.0.0.22.2 **slcd_alt_clock_div_t slcd_clock_config_t::altClkDivider**

25.3.2.0.0.22.3 **slcd_clock_prescaler_t slcd_clock_config_t::clkPrescaler**

25.3.2.0.0.22.4 **bool slcd_clock_config_t::fastFrameRateEnable**

25.3.3 **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

25.3.3.0.0.23 Field Documentation

25.3.3.0.0.23.1 `slcd_power_supply_option_t slcd_config_t::powerSupply`

25.3.3.0.0.23.2 `slcd_regulated_voltage_trim_t slcd_config_t::voltageTrim`

25.3.3.0.0.23.3 `slcd_clock_config_t* slcd_config_t::clkConfig`

25.3.3.0.0.23.4 `slcd_display_mode_t slcd_config_t::displayMode`

25.3.3.0.0.23.5 `slcd_load_adjust_t slcd_config_t::loadAdjust`

25.3.3.0.0.23.6 `slcd_duty_cycle_t slcd_config_t::dutyCycle`

25.3.3.0.0.23.7 `slcd_lowpower_behavior slcd_config_t::lowPowerBehavior`

25.3.3.0.0.23.8 `uint32_t slcd_config_t::slcdLowPinEnabled`

Setting bit n to 1 means enable pin n.

25.3.3.0.0.23.9 `uint32_t slcd_config_t::slcdHighPinEnabled`

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

25.3.3.0.0.23.10 `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.

25.3.3.0.0.23.11 `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.

25.3.3.0.0.23.12 `slcd_fault_detect_config_t* slcd_config_t::faultConfig`

If not requirement, set to NULL.

25.4 Macro Definition Documentation

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

25.5 Enumeration Type Documentation

25.5.1 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.

25.5.2 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.

25.5.3 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,V-LL2,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.

25.5.4 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.

25.5.5 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.

25.5.6 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.

25.5.7 enum slcd_duty_cycle_t

Enumerator

<i>kSLCD_IDiv1DutyCycle</i>	LCD use 1 BP 1/1 duty cycle.
<i>kSLCD_IDiv2DutyCycle</i>	LCD use 2 BP 1/2 duty cycle.
<i>kSLCD_IDiv3DutyCycle</i>	LCD use 3 BP 1/3 duty cycle.
<i>kSLCD_IDiv4DutyCycle</i>	LCD use 4 BP 1/4 duty cycle.
<i>kSLCD_IDiv5DutyCycle</i>	LCD use 5 BP 1/5 duty cycle.
<i>kSLCD_IDiv6DutyCycle</i>	LCD use 6 BP 1/6 duty cycle.
<i>kSLCD_IDiv7DutyCycle</i>	LCD use 7 BP 1/7 duty cycle.
<i>kSLCD_IDiv8DutyCycle</i>	LCD use 8 BP 1/8 duty cycle.

25.5.8 enum slcd_phase_type_t

Enumerator

<i>kSLCD_NoPhaseActivate</i>	LCD waveform no phase activates.
<i>kSLCD_PhaseAActivate</i>	LCD waveform phase A activates.
<i>kSLCD_PhaseBActivate</i>	LCD waveform phase B activates.
<i>kSLCD_PhaseCActivate</i>	LCD waveform phase C activates.
<i>kSLCD_PhaseDActivate</i>	LCD waveform phase D activates.
<i>kSLCD_PhaseEActivate</i>	LCD waveform phase E activates.
<i>kSLCD_PhaseFActivate</i>	LCD waveform phase F activates.
<i>kSLCD_PhaseGActivate</i>	LCD waveform phase G activates.
<i>kSLCD_PhaseHActivate</i>	LCD waveform phase H activates.

25.5.9 enum slcd_phase_index_t

Enumerator

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

Enumeration Type Documentation

25.5.10 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_BankMode LCD Blank display mode.

25.5.11 enum slcd_blink_mode_t

Enumerator

kSLCD_BankDisplayBlink Display blank during the blink period.

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

25.5.12 enum slcd_blink_rate_t

Enumerator

kSLCD_BlinkRate00 SLCD blink rate is LCD clock/((2¹²)).

kSLCD_BlinkRate01 SLCD blink rate is LCD clock/((2¹³)).

kSLCD_BlinkRate02 SLCD blink rate is LCD clock/((2¹⁴)).

kSLCD_BlinkRate03 SLCD blink rate is LCD clock/((2¹⁵)).

kSLCD_BlinkRate04 SLCD blink rate is LCD clock/((2¹⁶)).

kSLCD_BlinkRate05 SLCD blink rate is LCD clock/((2¹⁷)).

kSLCD_BlinkRate06 SLCD blink rate is LCD clock/((2¹⁸)).

kSLCD_BlinkRate07 SLCD blink rate is LCD clock/((2¹⁹)).

25.5.13 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.

25.5.14 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.

25.5.15 enum slcd_interrupt_enable_t

Enumerator

- kSLCD_FaultDetectCompleteInterrupt*** SLCD fault detection complete interrupt source.

25.5.16 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.

25.6 Function Documentation**25.6.1 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 SLCD_Getdefaultconfig() is provided to get them. Use it verified for their applications. The others have no default settings, such as "clkConfig", and must be provided by the application before calling the SLCD_Init() API.

25.6.2 void SLCD_Deinit (LCD_Type * *base*)

Parameters

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

25.6.3 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_InternalVll3UseChargePump; // Use charge
    pump internal VLL3
config.voltageTrim     = kSLCD_RegulatedVoltageTrim00;
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.
------------------	---

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

Parameters

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

25.6.5 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.
-------------	-------------------------------

25.6.6 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.

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

Parameters

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

25.6.8 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>pinIdx</i>	SLCD back plane pin index. Range from 0 to 63.
<i>phase</i>	The phase activates for the back plane pin.

25.6.9 static void SLCD_SetFrontPlaneSegments (LCD_Type * *base*, uint32_t *pinIdx*, 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>pinIdx</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.

25.6.10 static void SLCD_SetFrontPlaneOnePhase (LCD_Type * *base*, uint32_t *pinIdx*, slcd_phase_index_t *phaseIdx*, 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.

25.6.11 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.

25.6.12 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.

25.6.13 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.

25.6.14 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.

25.6.15 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.

25.6.16 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 :: slcd_interrupt_enable_t.

Function Documentation

Chapter 26

SMC: System Mode Controller Driver

26.1 Overview

The KSDK provides a peripheral driver for the System Mode Controller (SMC) module of Kinetis 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.

26.2 Typical use case

26.2.1 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 [S-MC_SetPowerModeStop](#). As a result, the MCU enters the STOP mode and never is woken up by the interrupt. In this use case, the application first disables the interrupt through PRIMASK, sets the wakeup interrupt, and enters the STOP mode. After wakeup, enable the interrupt through PRIMASK. The MCU can still be woken up by disabling the interrupt through PRIMASK. The pre and post functions handle the PRIMASK.
2. Disable/enable the flash speculation. When entering stop modes, the flash speculation might be interrupted. As a result, pre functions disable the flash speculation and post functions enable it.

```
SMC_PreEnterStopModes();  
/* Enable the wakeup interrupt here. */  
SMC_SetPowerModeStop(SMC, kSMC_PartialStop);  
SMC_PostExitStopModes();
```

Data Structures

- struct [smc_power_mode_vlls_config_t](#)
SMC Very Low-Leakage Stop power mode configuration. [More...](#)

Typical use case

Enumerations

- enum `smc_power_mode_protection_t` {
 `kSMC_AllowPowerModeVlls` = SMC_PMPROT_AVLLS_MASK,
 `kSMC_AllowPowerModeLls` = SMC_PMPROT_ALLS_MASK,
 `kSMC_AllowPowerModeVlp` = SMC_PMPROT_AVLP_MASK,
 `kSMC_AllowPowerModeAll` }
 Power Modes Protection.
- enum `smc_power_state_t` {
 `kSMC_PowerStateRun` = 0x01U << 0U,
 `kSMC_PowerStateStop` = 0x01U << 1U,
 `kSMC_PowerStateVlpr` = 0x01U << 2U,
 `kSMC_PowerStateVlpw` = 0x01U << 3U,
 `kSMC_PowerStateVlps` = 0x01U << 4U,
 `kSMC_PowerStateLls` = 0x01U << 5U,
 `kSMC_PowerStateVlls` = 0x01U << 6U }
 Power Modes in PMSTAT.
- enum `smc_run_mode_t` {
 `kSMC_RunNormal` = 0U,
 `kSMC_RunVlpr` = 2U }
 Run mode definition.
- enum `smc_stop_mode_t` {
 `kSMC_StopNormal` = 0U,
 `kSMC_StopVlps` = 2U,
 `kSMC_StopLls` = 3U,
 `kSMC_StopVlls` = 4U }
 Stop mode definition.
- enum `smc_stop_submode_t` {
 `kSMC_StopSub0` = 0U,
 `kSMC_StopSub1` = 1U,
 `kSMC_StopSub2` = 2U,
 `kSMC_StopSub3` = 3U }
 VLLS/LLS stop sub mode definition.
- enum `smc_partial_stop_option_t` {
 `kSMC_PartialStop` = 0U,
 `kSMC_PartialStop1` = 1U,
 `kSMC_PartialStop2` = 2U }
 Partial STOP option.
- enum `_smc_status` { `kStatus_SMC_StopAbort` = MAKE_STATUS(kStatusGroup_POWER, 0) }
 SMC configuration status.

Driver version

- #define `FSL_SMC_DRIVER_VERSION` (MAKE_VERSION(2, 0, 3))
 SMC driver version 2.0.3.

System mode controller APIs

- static void [SMC_SetPowerModeProtection](#) (SMC_Type *base, uint8_t allowedModes)
Configures all power mode protection settings.
- static smc_power_state_t [SMC_GetPowerModeState](#) (SMC_Type *base)
Gets the current power mode status.
- void [SMC_PreEnterStopModes](#) (void)
Prepares to enter stop modes.
- void [SMC_PostExitStopModes](#) (void)
Recoveries after wake up from stop modes.
- static void [SMC_PreEnterWaitModes](#) (void)
Prepares to enter wait modes.
- static void [SMC_PostExitWaitModes](#) (void)
Recoveries after wake up from stop modes.
- status_t [SMC_SetPowerModeRun](#) (SMC_Type *base)
Configures the system to RUN power mode.
- status_t [SMC_SetPowerModeWait](#) (SMC_Type *base)
Configures the system to WAIT power mode.
- status_t [SMC_SetPowerModeStop](#) (SMC_Type *base, [smc_partial_stop_option_t](#) option)
Configures the system to Stop power mode.
- status_t [SMC_SetPowerModeVlpr](#) (SMC_Type *base)
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.

26.3 Data Structure Documentation

26.3.1 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.

26.4 Macro Definition Documentation

26.4.1 #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

Enumeration Type Documentation

26.5 Enumeration Type Documentation

26.5.1 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.

26.5.2 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

26.5.3 enum smc_run_mode_t

Enumerator

kSMC_RunNormal Normal RUN mode.

kSMC_RunVlpr Very-low-power RUN mode.

26.5.4 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.

26.5.5 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.

26.5.6 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.

26.5.7 enum _smc_status

Enumerator

- kStatus_SMC_StopAbort* Entering Stop mode is abort.

26.6 Function Documentation

26.6.1 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.

26.6.2 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.

26.6.3 void SMC_PreEnterStopModes (void)

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

26.6.4 void SMC_PostExitStopModes (void)

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

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

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

26.6.6 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](#).

26.6.7 **status_t SMC_SetPowerModeRun (SMC_Type * *base*)**

Function Documentation

Parameters

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

Returns

SMC configuration error code.

26.6.8 status_t SMC_SetPowerModeWait (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.6.9 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.

26.6.10 status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.6.11 status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.6.12 status_t SMC_SetPowerModeVlps (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.6.13 status_t SMC_SetPowerModeLls (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.6.14 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 27

SPI: Serial Peripheral Interface Driver

27.1 Overview

Modules

- SPI DMA Driver
- SPI Driver
- SPI FreeRTOS driver
- SPI µCOS/II driver
- SPI µCOS/III driver

27.2 SPI Driver

27.2.1 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.

27.2.2 Typical use case

27.2.2.1 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)
{
}

// ...
}

```

27.2.2.2 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_w1c_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, 2))  
*SPI driver version 2.0.2.*

## 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 INCTRL.*

## 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.*

### 27.2.3 Data Structure Documentation

#### 27.2.3.1 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\\_rx\\_fifo\\_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.*

#### 27.2.3.2 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.*

### 27.2.3.3 `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.*

#### 27.2.3.3.0.24 Field Documentation

##### 27.2.3.3.0.24.1 `uint32_t spi_transfer_t::flags`

### 27.2.3.4 `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.*

### 27.2.4 Macro Definition Documentation

27.2.4.1 `#define FSL_SPI_DRIVER_VERSION (MAKE_VERSION(2, 0, 2))`

27.2.4.2 `#define SPI_DUMMYDATA (0xFFU)`

### 27.2.5 Enumeration Type Documentation

#### 27.2.5.1 `enum _spi_status`

Enumerator

*kStatus\_SPI\_Busy* SPI bus is busy.

*kStatus\_SPI\_Idle* SPI is idle.

*kStatus\_SPI\_Error* SPI error.

#### 27.2.5.2 `enum spi_clock_polarity_t`

Enumerator

*kSPI\_ClockPolarityActiveHigh* Active-high SPI clock (idles low).

*kSPI\_ClockPolarityActiveLow* Active-low SPI clock (idles high).

#### 27.2.5.3 `enum spi_clock_phase_t`

Enumerator

*kSPI\_ClockPhaseFirstEdge* First edge on SPSCK occurs at the middle of the first cycle of a data transfer.

*kSPI\_ClockPhaseSecondEdge* First edge on SPSCK occurs at the start of the first cycle of a data transfer.

#### 27.2.5.4 `enum spi_shift_direction_t`

Enumerator

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

*kSPI\_LsbFirst* Data transfers start with least significant bit.

### 27.2.5.5 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.

### 27.2.5.6 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.

### 27.2.5.7 enum spi\_data\_bitcount\_mode\_t

Enumerator

*kSPI\_8BitMode* 8-bit data transmission mode

*kSPI\_16BitMode* 16-bit data transmission mode

### 27.2.5.8 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.

### 27.2.5.9 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.

### 27.2.5.10 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.

### 27.2.5.11 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.

### 27.2.5.12 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.

### 27.2.5.13 enum \_spi\_dma\_enable\_t

Enumerator

*kSPI\_TxDmaEnable* Tx DMA request source.

*kSPI\_RxDmaEnable* Rx DMA request source.

*kSPI\_DmaAllEnable* All DMA request source.

## 27.2.6 Function Documentation

### 27.2.6.1 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 |
|---------------|------------------------------------|

### 27.2.6.2 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.                   |

### 27.2.6.3 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 |
|---------------|------------------------------------------|

### 27.2.6.4 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\\_SlaveGetDefaultConfig\(\)](#). 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 |

### 27.2.6.5 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 |
|-------------|------------------|

### 27.2.6.6 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 |

### 27.2.6.7 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.

#### 27.2.6.8 static void SPI\_ClearInterrupt ( SPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

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

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

Parameters

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

#### 27.2.6.10 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"><li>• kSPI_RxFullAndModfInterruptEnable</li><li>• kSPI_TxEmptyInterruptEnable</li><li>• kSPI_MatchInterruptEnable</li><li>• kSPI_RxFifoNearFullInterruptEnable</li><li>• kSPI_TxFifoNearEmptyInterruptEnable</li></ul> |

**27.2.6.11 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 |

**27.2.6.12 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

**27.2.6.13 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. |

#### **27.2.6.14 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.      |

#### **27.2.6.15 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. |

#### **27.2.6.16 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 |

### 27.2.6.17 void SPI\_WriteData ( SPI\_Type \* *base*, uint16\_t *data* )

Parameters

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

### 27.2.6.18 uint16\_t SPI\_ReadData ( SPI\_Type \* *base* )

Parameters

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

Returns

Data in the register.

### 27.2.6.19 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.                   |

### 27.2.6.20 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.     |

#### 27.2.6.21 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. |

#### 27.2.6.22 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. |

### 27.2.6.23 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. |

### 27.2.6.24 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. |

### 27.2.6.25 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.                   |

#### 27.2.6.26 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. |

#### 27.2.6.27 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_NoTransferIn-Progress</i> | There is not a non-blocking transaction currently in progress. |

**27.2.6.28 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. |

**27.2.6.29 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 |

## 27.3 SPI DMA Driver

### 27.3.1 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.*

### 27.3.2 Data Structure Documentation

#### 27.3.2.1 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.*

### 27.3.3 Typedef Documentation

#### 27.3.3.1 `typedef void(* spi_dma_callback_t)(SPI_Type *base, spi_dma_handle_t *handle, status_t status, void *userData)`

### 27.3.4 Function Documentation

#### 27.3.4.1 `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. |

#### 27.3.4.2 **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. |

#### 27.3.4.3 **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. |
|---------------|-------------------------|

**27.3.4.4 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. |

**27.3.4.5 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. |

**27.3.4.6 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. |

**27.3.4.7 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.      |

**27.3.4.8 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. |

## 27.4 SPI FreeRTOS driver

### 27.4.1 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.*

### 27.4.2 Function Documentation

#### 27.4.2.1 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.

#### 27.4.2.2 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. |
|---------------|----------------------|

### 27.4.2.3 **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.

## 27.5 SPI µCOS/II driver

### 27.5.1 Overview

This section describes the programming interface of the SPI µCOS/II 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.*

### 27.5.2 Function Documentation

#### 27.5.2.1 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.

#### 27.5.2.2 status\_t SPI\_RTOS\_Deinit ( `spi_rtos_handle_t * handle` )

This function deinitializes the SPI module and related RTOS context.

## SPI µCOS/II driver

Parameters

|               |                      |
|---------------|----------------------|
| <i>handle</i> | The RTOS SPI handle. |
|---------------|----------------------|

### 27.5.2.3 **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.

## 27.6 SPI µCOS/III driver

### 27.6.1 Overview

This section describes the programming interface of the SPI µCOS/III 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.*

### 27.6.2 Function Documentation

#### 27.6.2.1 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.

#### 27.6.2.2 status\_t SPI\_RTOS\_Deinit ( *spi\_rtos\_handle\_t \* handle* )

This function deinitializes the SPI module and related RTOS context.

## SPI µCOS/III driver

Parameters

|               |                      |
|---------------|----------------------|
| <i>handle</i> | The RTOS SPI handle. |
|---------------|----------------------|

### 27.6.2.3 **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 28

## TPM: Timer PWM Module

### 28.1 Overview

The KSDK provides a driver for the Timer PWM Module (TPM) of Kinetis devices.

The KSDK 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 input signal.

## Typical use case

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.

## 28.2 Typical use case

### 28.2.1 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\_trigger\_source\_t** {
 kTPM\_TriggerSource\_External = 0U,
 kTPM\_TriggerSource\_Internal }
   
*Trigger source 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\_SHIFT)),
 kTPM\_SetOnMatch = ((1U << TPM\_CnSC\_MSA\_SHIFT) | (3U << TPM\_CnSC\_ELSA\_SHIF-

## Typical use case

- T)),  
kTPM\_HighPulseOutput = ((3U << TPM\_CnSC\_MSA\_SHIFT) | (1U << TPM\_CnSC\_ELSA\_SHIFT)),  
kTPM\_LowPulseOutput = ((3U << TPM\_CnSC\_MSA\_SHIFT) | (2U << TPM\_CnSC\_ELSA\_SHIFT)) }
- 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.*

## Data Structure Documentation

### 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.*

## 28.3 Data Structure Documentation

### 28.3.1 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)...*

#### 28.3.1.0.0.25 Field Documentation

##### 28.3.1.0.0.25.1 [tpm\\_chnl\\_t tpm\\_chnl\\_pwm\\_signal\\_param\\_t::chnlNumber](#)

In combined mode (available in some SoC's, this represents the channel pair number

##### 28.3.1.0.0.25.2 [uint8\\_t tpm\\_chnl\\_pwm\\_signal\\_param\\_t::dutyCyclePercent](#)

100=always active signal (100% duty cycle)

### 28.3.2 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.*
- [tpm\\_trigger\\_source\\_t triggerSource](#)

*Decides if we use external or internal trigger.*

- 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*
- bool `enablePauseOnTrigger`  
*true: TPM counter will pause while trigger remains asserted; false: TPM counter continues running*

### 28.3.2.0.0.26 Field Documentation

#### 28.3.2.0.0.26.1 tpm\_trigger\_source\_t tpm\_config\_t::triggerSource

## 28.4 Enumeration Type Documentation

### 28.4.1 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.

### 28.4.2 enum tpm\_pwm\_mode\_t

Enumerator

- `kTPM_EdgeAlignedPwm` Edge aligned PWM.
- `kTPM_CenterAlignedPwm` Center aligned PWM.

## Enumeration Type Documentation

### 28.4.3 enum tpm\_pwm\_level\_select\_t

Enumerator

*kTPM\_NoPwmSignal* No PWM output on pin.

*kTPM\_LowTrue* Low true pulses.

*kTPM\_HighTrue* High true pulses.

### 28.4.4 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.

### 28.4.5 enum tpm\_trigger\_source\_t

Note

This selection is available only on some SoC's. For SoC's without this selection, the only trigger source available is internal trigger.

Enumerator

*kTPM\_TriggerSource\_External* Use external trigger input.

*kTPM\_TriggerSource\_Internal* Use internal trigger.

### 28.4.6 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.

**28.4.7 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.**28.4.8 enum tpm\_clock\_source\_t**

Enumerator

*kTPM\_SystemClock* System clock.*kTPM\_ExternalClock* External clock.**28.4.9 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.**28.4.10 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.

## Function Documentation

### 28.4.11 enum tpm\_status\_flags\_t

Enumerator

|                              |                     |
|------------------------------|---------------------|
| <i>kTPM_Chnl0Flag</i>        | Channel 0 flag.     |
| <i>kTPM_Chnl1Flag</i>        | Channel 1 flag.     |
| <i>kTPM_Chnl2Flag</i>        | Channel 2 flag.     |
| <i>kTPM_Chnl3Flag</i>        | Channel 3 flag.     |
| <i>kTPM_Chnl4Flag</i>        | Channel 4 flag.     |
| <i>kTPM_Chnl5Flag</i>        | Channel 5 flag.     |
| <i>kTPM_Chnl6Flag</i>        | Channel 6 flag.     |
| <i>kTPM_Chnl7Flag</i>        | Channel 7 flag.     |
| <i>kTPM_TimeOverflowFlag</i> | Time overflow flag. |

## 28.5 Function Documentation

### 28.5.1 void TPM\_Init ( TPM\_Type \* *base*, const tpm\_config\_t \* *config* )

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. |

### 28.5.2 void TPM\_Deinit ( TPM\_Type \* *base* )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

### 28.5.3 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. |
|---------------|-----------------------------------------|

#### 28.5.4 **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 <a href="#">tpm_pwm_mode_t</a> |
| <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

#### 28.5.5 **void TPM\_UpdatePwmDutycycle ( TPM\_Type \* *base*, tpm\_chnl\_t *chnlNumber*, tpm\_pwm\_mode\_t *currentPwmMode*, uint8\_t *dutyCyclePercent* )**

## Function Documentation

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) |

### 28.5.6 void TPM\_UpdateChnlEdgeLevelSelect ( **TPM\_Type** \* *base*, **tpm\_chnl\_t** *chnlNumber*, **uint8\_t** *level* )

Parameters

|                   |                                                                                                                                                       |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | TPM peripheral base address                                                                                                                           |
| <i>chnlNumber</i> | The channel number                                                                                                                                    |
| <i>level</i>      | The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field. |

### 28.5.7 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 |

### 28.5.8 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.                            |

### 28.5.9 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 <a href="#">tpm_interrupt_enable_t</a> |

### 28.5.10 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 <a href="#">tpm_interrupt_enable_t</a> |

### 28.5.11 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](#)

### 28.5.12 static uint32\_t TPM\_GetStatusFlags ( TPM\_Type \* *base* ) [inline], [static]

## Function Documentation

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](#)

### 28.5.13 static void TPM\_ClearStatusFlags ( **TPM\_Type \* base**, **uint32\_t mask** ) [inline], [static]

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 <a href="#">tpm_status_flags_t</a> |

### 28.5.14 static void TPM\_StartTimer ( **TPM\_Type \* base**, **tpm\_clock\_source\_t clockSource** ) [inline], [static]

Parameters

|                    |                                                                           |
|--------------------|---------------------------------------------------------------------------|
| <i>base</i>        | TPM peripheral base address                                               |
| <i>clockSource</i> | TPM clock source; once clock source is set the counter will start running |

### 28.5.15 static void TPM\_StopTimer ( **TPM\_Type \* base** ) [inline], [static]

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | TPM peripheral base address |
|-------------|-----------------------------|

## Chapter 29

# UART: Universal Asynchronous Receiver/Transmitter Driver

### 29.1 Overview

#### Modules

- [UART DMA Driver](#)
- [UART Driver](#)
- [UART FreeRTOS Driver](#)
- [UART eDMA Driver](#)
- [UART μCOS/II Driver](#)
- [UART μCOS/III Driver](#)

### 29.2 UART Driver

#### 29.2.1 Overview

The KSDK provides a peripheral driver for the Universal Asynchronous Receiver/Transmitter (UART) module of Kinetis devices.

The UART driver includes functional APIs and transactional APIs.

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

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the `uart_handle_t` as the second parameter. Initialize the handle by calling the [UART\\_TransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer, which means that the functions [UART\\_TransferSendNonBlocking\(\)](#) and [UART\\_TransferReceiveNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_UART_TxIdle` and `kStatus_UART_RxIdle`.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the [UART\\_TransferCreateHandle\(\)](#). If passing NULL, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The [UART\\_TransferReceiveNonBlocking\(\)](#) function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the `kStatus_UART_RxIdle`.

If the receive ring buffer is full, the upper layer is informed through a callback with the `kStatus_UART_RxRingBufferOverrun`. In the callback function, the upper layer reads data out from the ring buffer. If not, existing data is overwritten by the new data.

The ring buffer size is specified when creating the handle. Note that one byte is reserved for the ring buffer maintenance. When creating handle using the following code.

```
UART_TransferCreateHandle(UART0, &handle, UART_UserCallback, NULL);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

#### 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_uartTxHandle, DMA0, UART_TX_DMA_CHANNEL);
DMA_CreateHandle(&g_uartRxHandle, DMA0, UART_RX_DMA_CHANNEL);

UART_TransferCreateHandleDMA(UART1, &g_uartHandle, UART_UserCallback, NULL,
 &g_uartTxHandle, &g_uartRxHandle);

// Prepare to send.
sendXfer.data = sendData;
sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
txFinished = false;

// Send out.
UART_TransferSendDMA(UART1, &g_uartHandle, &sendXfer);

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

// Prepare to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
rxFinished = false;

// Receive.
UART_TransferReceiveDMA(UART1, &g_uartHandle, &receiveXfer);

// Wait receive finished.
while (!rxFinished)
{
}

// ...
}
```

## Data Structures

- struct [uart\\_config\\_t](#)  
*UART configuration structure.* [More...](#)
- struct [uart\\_transfer\\_t](#)  
*UART transfer structure.* [More...](#)
- struct [uart\\_handle\\_t](#)  
*UART handle structure.* [More...](#)

## TypeDefs

- [typedef void\(\\* uart\\_transfer\\_callback\\_t \)](#)(UART\_Type \*base, [uart\\_handle\\_t](#) \*handle, [status\\_t](#) status, void \*userData)  
*UART transfer callback function.*

## Enumerations

- enum `_uart_status` {
   
    `kStatus_UART_TxBusy` = MAKE\_STATUS(kStatusGroup\_UART, 0),
   
    `kStatus_UART_RxBusy` = MAKE\_STATUS(kStatusGroup\_UART, 1),
   
    `kStatus_UART_TxIdle` = MAKE\_STATUS(kStatusGroup\_UART, 2),
   
    `kStatus_UART_RxIdle` = MAKE\_STATUS(kStatusGroup\_UART, 3),
   
    `kStatus_UART_TxWatermarkTooLarge` = MAKE\_STATUS(kStatusGroup\_UART, 4),
   
    `kStatus_UART_RxWatermarkTooLarge` = MAKE\_STATUS(kStatusGroup\_UART, 5),
   
    `kStatus_UART_FlagCannotClearManually`,
   
    `kStatus_UART_Error` = MAKE\_STATUS(kStatusGroup\_UART, 7),
   
    `kStatus_UART_RxRingBufferOverrun` = MAKE\_STATUS(kStatusGroup\_UART, 8),
   
    `kStatus_UART_RxHardwareOverrun` = MAKE\_STATUS(kStatusGroup\_UART, 9),
   
    `kStatus_UART_NoiseError` = MAKE\_STATUS(kStatusGroup\_UART, 10),
   
    `kStatus_UART_FramingError` = MAKE\_STATUS(kStatusGroup\_UART, 11),
   
    `kStatus_UART_ParityError` = MAKE\_STATUS(kStatusGroup\_UART, 12),
   
    `kStatus_UART_BaudrateNotSupport` }
   
    *Error codes for the UART driver.*
- enum `uart_parity_mode_t` {
   
    `kUART_ParityDisabled` = 0x0U,
   
    `kUART_ParityEven` = 0x2U,
   
    `kUART_ParityOdd` = 0x3U }
   
    *UART parity mode.*
- enum `uart_stop_bit_count_t` {
   
    `kUART_OneStopBit` = 0U,
   
    `kUART_TwoStopBit` = 1U }
   
    *UART stop bit count.*
- enum `_uart_interrupt_enable` {
   
    `kUART_RxActiveEdgeInterruptEnable` = (UART\_BDH\_RXEDGIE\_MASK),
   
    `kUART_TxDataRegEmptyInterruptEnable` = (UART\_C2\_TIE\_MASK << 8),
   
    `kUART_TransmissionCompleteInterruptEnable` = (UART\_C2\_TCIE\_MASK << 8),
   
    `kUART_RxDataRegFullInterruptEnable` = (UART\_C2\_RIE\_MASK << 8),
   
    `kUART_IdleLineInterruptEnable` = (UART\_C2\_ILIE\_MASK << 8),
   
    `kUART_RxOverrunInterruptEnable` = (UART\_C3\_ORIE\_MASK << 16),
   
    `kUART_NoiseErrorInterruptEnable` = (UART\_C3\_NEIE\_MASK << 16),
   
    `kUART_FramingErrorInterruptEnable` = (UART\_C3\_FEIE\_MASK << 16),
   
    `kUART_ParityErrorInterruptEnable` = (UART\_C3\_PEIE\_MASK << 16) }
   
    *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_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.*
- 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.*

#### 29.2.3.2.0.27 Field Documentation

##### 29.2.3.2.0.27.1 uint8\_t\* [uart\\_transfer\\_t::data](#)

##### 29.2.3.2.0.27.2 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

### 29.2.3.3.0.28 Field Documentation

- 29.2.3.3.0.28.1 `uint8_t* volatile uart_handle_t::txData`
- 29.2.3.3.0.28.2 `volatile size_t uart_handle_t::txDataSize`
- 29.2.3.3.0.28.3 `size_t uart_handle_t::txDataSizeAll`
- 29.2.3.3.0.28.4 `uint8_t* volatile uart_handle_t::rxData`
- 29.2.3.3.0.28.5 `volatile size_t uart_handle_t::rxDataSize`
- 29.2.3.3.0.28.6 `size_t uart_handle_t::rxDataSizeAll`
- 29.2.3.3.0.28.7 `uint8_t* uart_handle_t::rxRingBuffer`
- 29.2.3.3.0.28.8 `size_t uart_handle_t::rxRingBufferSize`
- 29.2.3.3.0.28.9 `volatile uint16_t uart_handle_t::rxRingBufferHead`
- 29.2.3.3.0.28.10 `volatile uint16_t uart_handle_t::rxRingBufferTail`
- 29.2.3.3.0.28.11 `uart_transfer_callback_t uart_handle_t::callback`
- 29.2.3.3.0.28.12 `void* uart_handle_t::userData`
- 29.2.3.3.0.28.13 `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\_RxRingBufferOverrun*** UART RX software ring buffer overrun.  
***kStatus\_UART\_RxHardwareOverrun*** UART RX receiver overrun.  
***kStatus\_UART\_NoiseError*** UART noise error.  
***kStatus\_UART\_FramingError*** UART framing error.  
***kStatus\_UART\_ParityError*** UART parity error.  
***kStatus\_UART\_BaudrateNotSupport*** Baudrate is not support in current clock source.

### 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\_RxActiveEdgeInterruptEnable*** RX active edge interrupt.  
***kUART\_TxDataRegEmptyInterruptEnable*** Transmit data register empty interrupt.  
***kUART\_TransmissionCompleteInterruptEnable*** Transmission complete interrupt.  
***kUART\_RxDataRegFullInterruptEnable*** Receiver data register full interrupt.  
***kUART\_IdleLineInterruptEnable*** Idle line interrupt.  
***kUART\_RxOverrunInterruptEnable*** Receiver overrun interrupt.  
***kUART\_NoiseErrorInterruptEnable*** Noise error flag interrupt.  
***kUART\_FramingErrorInterruptEnable*** Framing error flag interrupt.  
***kUART\_ParityErrorInterruptEnable*** Parity error flag interrupt.

### 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\_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, 20000000U);
*
```

## UART Driver

Parameters

|                     |                                    |
|---------------------|------------------------------------|
| <i>base</i>         | UART peripheral base address.      |
| <i>baudRate_Bps</i> | UART baudrate to be set.           |
| <i>srcClock_Hz</i>  | UART clock source frequency in Hz. |

Return values

|                                         |                                                      |
|-----------------------------------------|------------------------------------------------------|
| <i>kStatus_UART_Baudrate-NotSupport</i> | Baudrate is not support in the current clock source. |
| <i>kStatus_Success</i>                  | Set baudrate succeeded.                              |

### 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 <a href="#">_uart_flags</a> . |

## Return values

|                                             |                                                                                         |
|---------------------------------------------|-----------------------------------------------------------------------------------------|
| <i>kStatus_UART_FlagCannotClearManually</i> | The flag can't be cleared by this function but it is cleared automatically by hardware. |
| <i>kStatus_Success</i>                      | Status in the mask is cleared.                                                          |

**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 <a href="#">_uart_interrupt_enable</a> . |

**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 <a href="#">_uart_interrupt_enable</a> . |

### 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_TxTransmissionCompleteFlag` to ensure that the TX is finished.

Parameters

|               |                                                                |
|---------------|----------------------------------------------------------------|
| <i>base</i>   | UART peripheral base address.                                  |
| <i>handle</i> | UART handle pointer.                                           |
| <i>xfer</i>   | UART transfer structure. See <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                                                                    |
|--------------------------------|------------------------------------------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully start the data transmission.                                          |
| <i>kStatus_UART_TxBusy</i>     | Previous transmission still not finished; data not all written to TX register yet. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.                                                                  |

### 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 <code>count</code> ; |

### 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 [k-Status\\_UART\\_RxIdle](#). For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the `xfer->data` and this function returns with the parameter `receivedBytes` set to 5. For the left 5 bytes, newly arrived data is saved from the `xfer->data[5]`. When 5 bytes are received, the UART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the `xfer->data`. When all data is received, the upper layer is notified.

#### Parameters

|                      |                                                                |
|----------------------|----------------------------------------------------------------|
| <i>base</i>          | UART peripheral base address.                                  |
| <i>handle</i>        | UART handle pointer.                                           |
| <i>xfer</i>          | UART transfer structure, see <a href="#">uart_transfer_t</a> . |
| <i>receivedBytes</i> | Bytes received from the ring buffer directly.                  |

#### Return values

|                                |                                                      |
|--------------------------------|------------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully queue the transfer into transmit queue. |
| <i>kStatus_UART_RxBusy</i>     | Previous receive request is not finished.            |
| <i>kStatus_InvalidArgument</i> | Invalid argument.                                    |

### 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.*

### 29.3.2.1.0.29 Field Documentation

29.3.2.1.0.29.1 `UART_Type* uart_dma_handle_t::base`

29.3.2.1.0.29.2 `uart_dma_transfer_callback_t uart_dma_handle_t::callback`

29.3.2.1.0.29.3 `void* uart_dma_handle_t::userData`

29.3.2.1.0.29.4 `size_t uart_dma_handle_t::rxDataSizeAll`

29.3.2.1.0.29.5 `size_t uart_dma_handle_t::txDataSizeAll`

29.3.2.1.0.29.6 `dma_handle_t* uart_dma_handle_t::txDmaHandle`

29.3.2.1.0.29.7 `dma_handle_t* uart_dma_handle_t::rxDmaHandle`

29.3.2.1.0.29.8 `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 <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_TxBusy</i>     | Previous transfer ongoing.      |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

#### 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 <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_RxBusy</i>     | Previous transfer on going.     |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

### 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_NoTransferIn-Progress</i> | No send in progress.                                  |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

#### 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_NoTransferIn-Progress</i> | No receive in progress.                               |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

### 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.*

#### 29.4.2.1.0.30 Field Documentation

29.4.2.1.0.30.1 `uart_edma_transfer_callback_t uart_edma_handle_t::callback`

29.4.2.1.0.30.2 `void* uart_edma_handle_t::userData`

29.4.2.1.0.30.3 `size_t uart_edma_handle_t::rxDataSizeAll`

29.4.2.1.0.30.4 `size_t uart_edma_handle_t::txDataSizeAll`

29.4.2.1.0.30.5 `edma_handle_t* uart_edma_handle_t::txEdmaHandle`

29.4.2.1.0.30.6 `edma_handle_t* uart_edma_handle_t::rxEdmaHandle`

29.4.2.1.0.30.7 `uint8_t uart_edma_handle_t::nbytes`

29.4.2.1.0.30.8 `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 <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_TxBusy</i>     | Previous transfer ongoing.      |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

### 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 <a href="#">uart_transfer_t</a> . |

Return values

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kStatus_Success</i>         | if succeeded; otherwise failed. |
| <i>kStatus_UART_RxBusy</i>     | Previous transfer ongoing.      |
| <i>kStatus_InvalidArgument</i> | Invalid argument.               |

#### **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_NoTransferIn-Progress</i> | No send in progress.                                  |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

### 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_NoTransferIn-Progress</i> | No receive in progress.                               |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

## 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`

UART RTOS configuration structure.

#### 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.*

## UART FreeRTOS Driver

- `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

|                       |                                                                                     |
|-----------------------|-------------------------------------------------------------------------------------|
| <code>handle</code>   | The RTOS UART handle, the pointer to an allocated space for RTOS context.           |
| <code>t_handle</code> | The pointer to the allocated space to store the transactional layer internal state. |
| <code>cfg</code>      | 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

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | 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

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | 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.

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. |

### 29.6 UART µCOS/II Driver

#### 29.6.1 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.*

#### 29.6.2 Data Structure Documentation

##### 29.6.2.1 struct `uart_rtos_config_t`

UART RTOS configuration structure.

#### 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.*

## 29.6.3 Function Documentation

### 29.6.3.1 int UART\_RRTOS\_Init ( `uart_rtos_handle_t * handle`, `uart_handle_t * t_handle`, `const uart_rtos_config_t * cfg` )

Parameters

|                            |                                                                                           |
|----------------------------|-------------------------------------------------------------------------------------------|
| <code>handle</code>        | The RTOS UART handle; the pointer to an allocated space for RTOS context.                 |
| <code>uart_t_handle</code> | The pointer to the allocated space where to store the transactional layer internal state. |
| <code>cfg</code>           | The pointer to the parameters required to configure the UART after initialization.        |

Returns

0 Succeed; otherwise fail.

### 29.6.3.2 int UART\_RRTOS\_Deinit ( `uart_rtos_handle_t * handle` )

This function deinitializes the UART module, sets all register values to reset value, and frees the resources.

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

### 29.6.3.3 int UART\_RRTOS\_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

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

## UART µCOS/II Driver

|               |                                    |
|---------------|------------------------------------|
| <i>buffer</i> | The pointer to the buffer to send. |
| <i>length</i> | The number of bytes to send.       |

### 29.6.3.4 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 any data is immediately available it is returned immediately and the number of bytes received.

Parameters

|                 |                                                                                  |
|-----------------|----------------------------------------------------------------------------------|
| <i>handle</i>   | The RTOS UART handle.                                                            |
| <i>buffer</i>   | The pointer to the buffer where 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. |

## 29.7 UART µCOS/III Driver

### 29.7.1 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.*

### 29.7.2 Data Structure Documentation

#### 29.7.2.1 struct `uart_rtos_config_t`

UART RTOS configuration structure.

#### Data Fields

- `UART_Type * base`  
*UART base address.*
- `uint32_t srclk`  
*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.*

## UART µCOS/III Driver

- `uint32_t buffer_size`  
*Size of buffer for background reception.*

### 29.7.3 Function Documentation

#### 29.7.3.1 int `UART_RRTOS_Init` ( `uart_rtos_handle_t * handle`, `uart_handle_t * t_handle`, `const uart_rtos_config_t * cfg` )

Parameters

|                            |                                                                                    |
|----------------------------|------------------------------------------------------------------------------------|
| <code>handle</code>        | The RTOS UART handle; the pointer to allocated space for RTOS context.             |
| <code>uart_t_handle</code> | The pointer to an allocated space to store transactional layer internal state.     |
| <code>cfg</code>           | The pointer to the parameters required to configure the UART after initialization. |

Returns

0 Succeed; otherwise fail.

#### 29.7.3.2 int `UART_RRTOS_Deinit` ( `uart_rtos_handle_t * handle` )

This function deinitializes the UART module, sets all register values to reset value, and releases the resources.

Parameters

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

#### 29.7.3.3 int `UART_RRTOS_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

|                     |                       |
|---------------------|-----------------------|
| <code>handle</code> | The RTOS UART handle. |
|---------------------|-----------------------|

|               |                                    |
|---------------|------------------------------------|
| <i>buffer</i> | The pointer to the buffer to send. |
| <i>length</i> | The number of bytes to send.       |

#### 29.7.3.4 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 any data is immediately available, it is returned immediately and the number of bytes received.

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 variable of a size_t where the number of received data is filled. |



# Chapter 30

## VREF: Voltage Reference Driver

### 30.1 Overview

The KSDK provides a peripheral driver for the Crossbar Voltage Reference (VREF) block of Kinetis devices.

The Voltage Reference(VREF) supplies an accurate 1.2 V voltage output that can be trimmed in 0.5 mV steps. VREF can be used in applications to provide a reference voltage to external devices and to internal analog peripherals, such as the ADC, DAC, or CMP. The voltage reference has operating modes that provide different levels of supply rejection and power consumption.

To configure the VREF driver, configure `vref_config_t` structure in one of two ways.

1. Use the `VREF_GetDefaultConfig()` function.
2. Set the parameter in the `vref_config_t` structure.

To initialize the VREF driver, call the `VREF_Init()` function and pass a pointer to the `vref_config_t` structure.

To de-initialize the VREF driver, call the `VREF_Deinit()` function.

### 30.2 Typical use case and example

This example shows how to generate a reference voltage by using the VREF module.

```
vref_config_t vrefUserConfig;
VREF_GetDefaultConfig(&vrefUserConfig); /* Gets a default configuration. */
VREF_Init(VREF, &vrefUserConfig); /* Initializes and configures the VREF module */

/* Do something */

VREF_Deinit(VREF); /* De-initializes the VREF module */
```

## Data Structures

- struct `vref_config_t`  
*The description structure for the VREF module. [More...](#)*

## Enumerations

- enum `vref_buffer_mode_t` {  
    `kVREF_ModeBandgapOnly` = 0U,  
    `kVREF_ModeHighPowerBuffer` = 1U,  
    `kVREF_ModeLowPowerBuffer` = 2U }  
*VREF modes.*

## Function Documentation

### Driver version

- #define **FSL\_VREF\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 0))  
*Version 2.1.0.*

### VREF functional operation

- void **VREF\_Init** (VREF\_Type \*base, const vref\_config\_t \*config)  
*Enables the clock gate and configures the VREF module according to the configuration structure.*
- void **VREF\_Deinit** (VREF\_Type \*base)  
*Stops and disables the clock for the VREF module.*
- void **VREF\_GetDefaultConfig** (vref\_config\_t \*config)  
*Initializes the VREF configuration structure.*
- void **VREF\_SetTrimVal** (VREF\_Type \*base, uint8\_t trimValue)  
*Sets a TRIM value for the reference voltage.*
- static uint8\_t **VREF\_GetTrimVal** (VREF\_Type \*base)  
*Reads the value of the TRIM meaning output voltage.*

### 30.3 Data Structure Documentation

#### 30.3.1 struct vref\_config\_t

##### Data Fields

- vref\_buffer\_mode\_t bufferMode  
*Buffer mode selection.*

### 30.4 Macro Definition Documentation

#### 30.4.1 #define FSL\_VREF\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 0))

### 30.5 Enumeration Type Documentation

#### 30.5.1 enum vref\_buffer\_mode\_t

Enumerator

**kVREF\_ModeBandgapOnly** Bandgap on only, for stabilization and startup.

**kVREF\_ModeHighPowerBuffer** High-power buffer mode enabled.

**kVREF\_ModeLowPowerBuffer** Low-power buffer mode enabled.

### 30.6 Function Documentation

#### 30.6.1 void VREF\_Init ( VREF\_Type \* *base*, const vref\_config\_t \* *config* )

This function must be called before calling all other VREF driver functions, read/write registers, and configurations with user-defined settings. The example below shows how to set up **vref\_config\_t** parameters and how to call the VREF\_Init function by passing in these parameters. This is an example.

```

* vref_config_t vrefConfig;
* vrefConfig.bufferMode = kVREF_ModeHighPowerBuffer;
* vrefConfig.enableExternalVoltRef = false;
* vrefConfig.enableLowRef = false;
* VREF_Init(VREF, &vrefConfig);
*

```

## Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | VREF peripheral address.                |
| <i>config</i> | Pointer to the configuration structure. |

**30.6.2 void VREF\_Deinit ( VREF\_Type \* *base* )**

This function should be called to shut down the module. This is an example.

```

* vref_config_t vrefUserConfig;
* VREF_Init(VREF);
* VREF_GetDefaultConfig(&vrefUserConfig);
* ...
* VREF_Deinit(VREF);
*

```

## Parameters

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

**30.6.3 void VREF\_GetDefaultConfig ( vref\_config\_t \* *config* )**

This function initializes the VREF configuration structure to default values. This is an example.

```

* vrefConfig->bufferMode = kVREF_ModeHighPowerBuffer;
* vrefConfig->enableExternalVoltRef = false;
* vrefConfig->enableLowRef = false;
*

```

## Parameters

|               |                                          |
|---------------|------------------------------------------|
| <i>config</i> | Pointer to the initialization structure. |
|---------------|------------------------------------------|

**30.6.4 void VREF\_SetTrimVal ( VREF\_Type \* *base*, uint8\_t *trimValue* )**

This function sets a TRIM value for the reference voltage. Note that the TRIM value maximum is 0x3F.

## Function Documentation

Parameters

|                  |                                                                                        |
|------------------|----------------------------------------------------------------------------------------|
| <i>base</i>      | VREF peripheral address.                                                               |
| <i>trimValue</i> | Value of the trim register to set the output reference voltage (maximum 0x3F (6-bit)). |

### 30.6.5 static uint8\_t VREF\_GetTrimVal ( VREF\_Type \* *base* ) [inline], [static]

This function gets the TRIM value from the TRM register.

Parameters

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

Returns

Six-bit value of trim setting.

# Chapter 31

## Clock Driver

### 31.1 Overview

The KSDK provides APIs for Kinetis devices clock operation.

### 31.2 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.

### 31.3 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 Lite (MCGLITE)

## Files

- file `fsl_clock.h`

## Data Structures

- struct `sim_clock_config_t`  
*SIM configuration structure for clock setting.* [More...](#)
- struct `oscer_config_t`  
*The OSC configuration for OSCERCLK.* [More...](#)
- struct `osc_config_t`  
*OSC Initialization Configuration Structure.* [More...](#)
- struct `mcglite_config_t`  
*MCG\_Lite configure structure for mode change.* [More...](#)

## External clock frequency

### Macros

- #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 **LPUART\_CLOCKS**  
*Clock ip name array for LPUART.*
- #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 **FLEXIO\_CLOCKS**  
*Clock ip name array for FLEXIO.*
- #define **VREF\_CLOCKS**  
*Clock ip name array for VREF.*
- #define **DMA\_CLOCKS**  
*Clock ip name array for DMA.*
- #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\_FlexBusClk,
   
kCLOCK\_FlashClk,
   
kCLOCK\_FastPeriphClk,
   
kCLOCK\_PllFllSelClk,
   
kCLOCK\_Er32kClk,
   
kCLOCK\_Osc0ErClk,
   
kCLOCK\_Osc1ErClk,
   
kCLOCK\_Osc0ErClkUndiv,
   
kCLOCK\_McgFixedFreqClk,
   
kCLOCK\_McgInternalRefClk,
   
kCLOCK\_McgFllClk,
   
kCLOCK\_McgPll0Clk,
   
kCLOCK\_McgPll1Clk,
   
kCLOCK\_McgExtPllClk,
   
kCLOCK\_McgPeriphClk,
   
kCLOCK\_McgIrc48MClk,
   
kCLOCK\_LpoClk }

*Clock name used to get clock frequency.*

- enum `clock_usb_src_t` {
   
kCLOCK\_UsbSrcIrc48M = SIM\_SOPT2\_USBSRC(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_cap_load` {
   
kOSC\_Cap2P = OSC\_CR\_SC2P\_MASK,
   
kOSC\_Cap4P = OSC\_CR\_SC4P\_MASK,
   
kOSC\_Cap8P = OSC\_CR\_SC8P\_MASK,
   
kOSC\_Cap16P = OSC\_CR\_SC16P\_MASK }

*Oscillator capacitor load setting.*

- enum `_oscer_enable_mode` {
   
kOSC\_ErClkEnable = OSC\_CR\_ERCLKEN\_MASK,
   
kOSC\_ErClkEnableInStop = OSC\_CR\_EREFSTEN\_MASK }

*OSCERCLK enable mode.*

- enum `osc_mode_t` {
   
kOSC\_ModeExt = 0U,
   
kOSC\_ModeOscLowPower = MCG\_C2\_EREFS0\_MASK,
   
kOSC\_ModeOscHighGain = MCG\_C2\_EREFS0\_MASK | MCG\_C2\_HGO0\_MASK }

*The OSC work mode.*

- enum `mcglite_clkout_src_t` {

## External clock frequency

- ```
kMCGLITE_ClkSrcHirc,  
kMCGLITE_ClkSrcLirc,  
kMCGLITE_ClkSrcExt }
```
- MCG_Lite clock source selection.*
- enum `mcglite_lirc_mode_t` {
 kMCGLITE_Lirc2M,
 kMCGLITE_Lirc8M }
- MCG_Lite LIRC select.*
- enum `mcglite_lirc_div_t` {
 kMCGLITE_LircDivBy1 = 0U,
 kMCGLITE_LircDivBy2,
 kMCGLITE_LircDivBy4,
 kMCGLITE_LircDivBy8,
 kMCGLITE_LircDivBy16,
 kMCGLITE_LircDivBy32,
 kMCGLITE_LircDivBy64,
 kMCGLITE_LircDivBy128 }
- MCG_Lite divider factor selection for clock source.*
- enum `mcglite_mode_t` {
 kMCGLITE_ModeHirc48M,
 kMCGLITE_ModeLirc8M,
 kMCGLITE_ModeLirc2M,
 kMCGLITE_ModeExt,
 kMCGLITE_ModeError }
- MCG_Lite clock mode definitions.*
- enum `_mcglite_irclk_enable_mode` {
 kMCGLITE_IrclkEnable = MCG_C1_IRCLKEN_MASK,
 kMCGLITE_IrclkEnableInStop = MCG_C1_IREFSTEN_MASK }
- MCG internal reference clock (MCGIRCLK) enable mode definition.*

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_SetLpuart0Clock` (`uint32_t` src)
Set LPUART0 clock source.
- static void `CLOCK_SetLpuart1Clock` (`uint32_t` src)
Set LPUART1 clock source.
- static void `CLOCK_SetTpmClock` (`uint32_t` src)
Set TPM clock source.
- static void `CLOCK_SetFlexio0Clock` (`uint32_t` src)
Set FLEXIO clock 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_SetClkOutClock** (uint32_t src)
Set CLKOUT source.
- static void **CLOCK_SetRtcClkOutClock** (uint32_t src)
Set RTC_CLKOUT source.
- 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.
- 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_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**
The external XTAL32/EXTAL32/RTC_CLKIN clock frequency.

Driver version

- #define **FSL_CLOCK_DRIVER_VERSION** (**MAKE_VERSION**(2, 1, 1))
CLOCK driver version 2.1.1.

MCG_Lite clock frequency

- uint32_t **CLOCK_GetOutClkFreq** (void)
Gets the MCG_Lite output clock (MCGOUTCLK) frequency.
- uint32_t **CLOCK_GetInternalRefClkFreq** (void)
Gets the MCG internal reference clock (MCGIRCLK) frequency.
- uint32_t **CLOCK_GetPeriphClkFreq** (void)
Gets the current MCGPCLK frequency.

MCG_Lite mode.

- **mcglite_mode_t CLOCK_GetMode** (void)

Data Structure Documentation

- Gets the current MCG_Lite mode.
- status_t [CLOCK_SetMcgliteConfig](#) (mcglite_config_t const *targetConfig)
Sets the MCG_Lite configuration.

OSC configuration

- static void [OSC_SetExtRefClkConfig](#) (OSC_Type *base, oscer_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.

31.4 Data Structure Documentation

31.4.1 struct sim_clock_config_t

Data Fields

- uint8_t [er32kSrc](#)
ERCLK32K source selection.
- uint32_t [clkdiv1](#)
SIM_CLKDIV1.

31.4.1.0.0.31 Field Documentation

31.4.1.0.0.31.1 uint8_t sim_clock_config_t::er32kSrc

31.4.1.0.0.31.2 uint32_t sim_clock_config_t::clkdiv1

31.4.2 struct oscer_config_t

Data Fields

- uint8_t [enableMode](#)
OSCERCLK enable mode.

31.4.2.0.0.32 Field Documentation

31.4.2.0.0.32.1 uint8_t oscer_config_t::enableMode

OR'ed value of `_oscer_enable_mode`.

31.4.3 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 settings:

1. freq: The external frequency.
2. workMode: The OSC module mode.

Data Fields

- `uint32_t freq`
External clock frequency.
- `uint8_t capLoad`
Capacitor load setting.
- `osc_mode_t workMode`
OSC work mode setting.
- `oscer_config_t oscerConfig`
Configuration for OSCERCLK.

31.4.3.0.0.33 Field Documentation

31.4.3.0.0.33.1 uint32_t osc_config_t::freq

31.4.3.0.0.33.2 uint8_t osc_config_t::capLoad

31.4.3.0.0.33.3 osc_mode_t osc_config_t::workMode

31.4.3.0.0.33.4 oscer_config_t osc_config_t::oscerConfig

31.4.4 struct mcglite_config_t

Data Fields

- `mcglite_clkout_src_t outSrc`
MCGOUT clock select.
- `uint8_t irclkEnableMode`
MCGIRCLK enable mode, OR'ed value of `_mcglite_irclk_enable_mode`.
- `mcglite_lirc_mode_t ircs`
MCG_C2[IRCS].
- `mcglite_lirc_div_t fcrdiv`
MCG_SC[FCRDIV].

Macro Definition Documentation

- `mcglite_lirc_div_t lircDiv2`
MCG_MC[LIRC_DIV2].
- `bool hircEnableInNotHircMode`
HIRC enable when not in HIRC mode.

31.4.4.0.0.34 Field Documentation

31.4.4.0.0.34.1 `mcglite_clkout_src_t mcglite_config_t::outSrc`

31.4.4.0.0.34.2 `uint8_t mcglite_config_t::irclkEnableMode`

31.4.4.0.0.34.3 `mcglite_lirc_mode_t mcglite_config_t::ircs`

31.4.4.0.0.34.4 `mcglite_lirc_div_t mcglite_config_t::fcrdiv`

31.4.4.0.0.34.5 `mcglite_lirc_div_t mcglite_config_t::lircDiv2`

31.4.4.0.0.34.6 `bool mcglite_config_t::hircEnableInNotHircMode`

31.5 Macro Definition Documentation

31.5.1 `#define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0`

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note

All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

31.5.2 `#define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))`

31.5.3 `#define DMAMUX_CLOCKS`

Value:

```
{           \
    kCLOCK_Dmamux0 \
}
```

31.5.4 `#define RTC_CLOCKS`

Value:

```
{           \
}   kCLOCK_Rtc0 \
```

31.5.5 #define SAI_CLOCKS

Value:

```
{           \
}   kCLOCK_Sai0 \
```

31.5.6 #define SPI_CLOCKS

Value:

```
{           \
}   kCLOCK_Spi0, kCLOCK_Spi1 \
```

31.5.7 #define SLCD_CLOCKS

Value:

```
{           \
}   kCLOCK_Slcd0 \
```

31.5.8 #define PIT_CLOCKS

Value:

```
{           \
}   kCLOCK_Pit0 \
```

31.5.9 #define PORT_CLOCKS

Value:

```
{           \
}   kCLOCK_PortA, kCLOCK_PortB, kCLOCK_PortC, kCLOCK_PortD, kCLOCK_PortE \
```

Macro Definition Documentation

31.5.10 #define LPUART_CLOCKS

Value:

```
{           \
    kCLOCK_Lpuart0, kCLOCK_Lpuart1 \
}
```

31.5.11 #define DAC_CLOCKS

Value:

```
{           \
    kCLOCK_Dac0 \
}
```

31.5.12 #define LPTMR_CLOCKS

Value:

```
{           \
    kCLOCK_Lptmr0 \
}
```

31.5.13 #define ADC16_CLOCKS

Value:

```
{           \
    kCLOCK_Adc0 \
}
```

31.5.14 #define FLEXIO_CLOCKS

Value:

```
{           \
    kCLOCK_Flexio0 \
}
```

31.5.15 #define VREF_CLOCKS

Value:

```
{           \
    kCLOCK_Vref0 \
}
```

31.5.16 #define DMA_CLOCKS

Value:

```
{           \
    kCLOCK_Dma0 \
}
```

31.5.17 #define UART_CLOCKS

Value:

```
{           \
    kCLOCK_IpInvalid, kCLOCK_IpInvalid, kCLOCK_Uart2 \
}
```

31.5.18 #define TPM_CLOCKS

Value:

```
{           \
    kCLOCK_Tpm0, kCLOCK_Tpm1, kCLOCK_Tpm2 \
}
```

31.5.19 #define I2C_CLOCKS

Value:

```
{           \
    kCLOCK_I2c0, kCLOCK_I2c1 \
}
```

Enumeration Type Documentation

31.5.20 #define FTF_CLOCKS

Value:

```
{\n    kCLOCK_Ftf0 \n}
```

31.5.21 #define CMP_CLOCKS

Value:

```
{\n    kCLOCK_Cmp0 \n}
```

31.5.22 #define SYS_CLK kCLOCK_CoreSysClk

31.6 Enumeration Type Documentation

31.6.1 enum clock_name_t

Enumerator

kCLOCK_CoreSysClk Core/system clock.
kCLOCK_PlatClk Platform clock.
kCLOCK_BusClk Bus clock.
kCLOCK_FlexBusClk FlexBus clock.
kCLOCK_FlashClk Flash clock.
kCLOCK_FastPeriphClk Fast peripheral clock.
kCLOCK_PllFllSelClk The clock after SIM[PLLFLSEL].
kCLOCK_Er32kClk External reference 32K clock (ERCLK32K)
kCLOCK_Osc0ErClk OSC0 external reference clock (OSC0ERCLK)
kCLOCK_Osc1ErClk OSC1 external reference clock (OSC1ERCLK)
kCLOCK_Osc0ErClkUndiv OSC0 external reference undivided clock(OSC0ERCLK_UNDIV).
kCLOCK_McgFixedFreqClk MCG fixed frequency clock (MCGFFCLK)
kCLOCK_McgInternalRefClk MCG internal reference clock (MCGIRCLK)
kCLOCK_McgFllClk MCGFLLCLK.
kCLOCK_McgPll0Clk MCGPLL0CLK.
kCLOCK_McgPll1Clk MCGPLL1CLK.
kCLOCK_McgExtPllClk EXT_PLLCLK.
kCLOCK_McgPeriphClk MCG peripheral clock (MCGPCLK)
kCLOCK_McgIrc48MClk MCG IRC48M clock.
kCLOCK_LpoClk LPO clock.

31.6.2 enum clock_usb_src_t

Enumerator

kCLOCK_UsbSrcIrc48M Use IRC48M.
kCLOCK_UsbSrcExt Use USB_CLKIN.

31.6.3 enum clock_ip_name_t

31.6.4 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

31.6.5 enum _oscer_enable_mode

Enumerator

kOSC_ErClkEnable Enable.
kOSC_ErClkEnableInStop Enable in stop mode.

31.6.6 enum osc_mode_t

Enumerator

kOSC_ModeExt Use external clock.
kOSC_ModeOscLowPower Oscillator low power.
kOSC_ModeOscHighGain Oscillator high gain.

31.6.7 enum mcglite_clkout_src_t

Enumerator

kMCGLITE_ClkSrcHirc MCGOUTCLK source is HIRC.
kMCGLITE_ClkSrcLirc MCGOUTCLK source is LIRC.
kMCGLITE_ClkSrcExt MCGOUTCLK source is external clock source.

Function Documentation

31.6.8 enum mcglite_lirc_mode_t

Enumerator

kMCGLITE_Lirc2M Slow internal reference(LIRC) 2 MHz clock selected.

kMCGLITE_Lirc8M Slow internal reference(LIRC) 8 MHz clock selected.

31.6.9 enum mcglite_lirc_div_t

Enumerator

kMCGLITE_LircDivBy1 Divider is 1.

kMCGLITE_LircDivBy2 Divider is 2.

kMCGLITE_LircDivBy4 Divider is 4.

kMCGLITE_LircDivBy8 Divider is 8.

kMCGLITE_LircDivBy16 Divider is 16.

kMCGLITE_LircDivBy32 Divider is 32.

kMCGLITE_LircDivBy64 Divider is 64.

kMCGLITE_LircDivBy128 Divider is 128.

31.6.10 enum mcglite_mode_t

Enumerator

kMCGLITE_ModeHirc48M Clock mode is HIRC 48 M.

kMCGLITE_ModeLirc8M Clock mode is LIRC 8 M.

kMCGLITE_ModeLirc2M Clock mode is LIRC 2 M.

kMCGLITE_ModeExt Clock mode is EXT.

kMCGLITE_ModeError Unknown mode.

31.6.11 enum _mcglite_irclk_enable_mode

Enumerator

kMCGLITE_IrclkEnable MCGIRCLK enable.

kMCGLITE_IrclkEnableInStop MCGIRCLK enable in stop mode.

31.7 Function Documentation

31.7.1 static void CLOCK_EnableClock (*clock_ip_name_t name*) [inline], [static]

Parameters

<i>name</i>	Which clock to enable, see clock_ip_name_t .
-------------	--

31.7.2 static void CLOCK_DisableClock (*clock_ip_name_t name*) [inline], [static]

Parameters

<i>name</i>	Which clock to disable, see clock_ip_name_t .
-------------	---

31.7.3 static void CLOCK_SetEr32kClock (*uint32_t src*) [inline], [static]

Parameters

<i>src</i>	The value to set ERCLK32K clock source.
------------	---

31.7.4 static void CLOCK_SetLpuart0Clock (*uint32_t src*) [inline], [static]

Parameters

<i>src</i>	The value to set LPUART0 clock source.
------------	--

31.7.5 static void CLOCK_SetLpuart1Clock (*uint32_t src*) [inline], [static]

Parameters

<i>src</i>	The value to set LPUART1 clock source.
------------	--

31.7.6 static void CLOCK_SetTpmClock (*uint32_t src*) [inline], [static]

Function Documentation

Parameters

<i>src</i>	The value to set TPM clock source.
------------	------------------------------------

31.7.7 static void CLOCK_SetFlexio0Clock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set FLEXIO clock source.
------------	---------------------------------------

31.7.8 bool CLOCK_EnableUsbfs0Clock (clock_usb_src_t *src*, uint32_t *freq*)

Parameters

<i>src</i>	USB FS clock source.
<i>freq</i>	The frequency specified by src.

Return values

<i>true</i>	The clock is set successfully.
<i>false</i>	The clock source is invalid to get proper USB FS clock.

31.7.9 static void CLOCK_DisableUsbfs0Clock (void) [inline], [static]

Disable USB FS clock.

31.7.10 static void CLOCK_SetClkOutClock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set CLKOUT source.
------------	---------------------------------

31.7.11 static void CLOCK_SetRtcClkOutClock (uint32_t *src*) [inline], [static]

Parameters

<i>src</i>	The value to set RTC_CLKOUT source.
------------	-------------------------------------

31.7.12 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.

31.7.13 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 <i>clock_name_t</i>
------------------	--

Returns

Clock frequency value in Hertz

31.7.14 uint32_t CLOCK_GetCoreSysClkFreq (*void*)

Returns

Clock frequency in Hz.

31.7.15 uint32_t CLOCK_GetPlatClkFreq (*void*)

Returns

Clock frequency in Hz.

Function Documentation

31.7.16 `uint32_t CLOCK_GetBusClkFreq(void)`

Returns

Clock frequency in Hz.

31.7.17 `uint32_t CLOCK_GetFlashClkFreq(void)`

Returns

Clock frequency in Hz.

31.7.18 `uint32_t CLOCK_GetEr32kClkFreq(void)`

Returns

Clock frequency in Hz.

31.7.19 `uint32_t CLOCK_GetOsc0ErClkFreq(void)`

Returns

Clock frequency in Hz.

31.7.20 `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.
---------------	-------------------------------------

31.7.21 `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.
---------------	-------------------------------------

31.7.22 **uint32_t CLOCK_GetOutClkFreq(void)**

This function gets the MCG_Lite output clock frequency in Hz based on the current MCG_Lite register value.

Returns

The frequency of MCGOUTCLK.

31.7.23 **uint32_t CLOCK_GetInternalRefClkFreq(void)**

This function gets the MCG_Lite internal reference clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGIRCLK.

31.7.24 **uint32_t CLOCK_GetPeriphClkFreq(void)**

This function gets the MCGPCLK frequency in Hz based on the current MCG_Lite register settings.

Returns

The frequency of MCGPCLK.

31.7.25 **mcglite_mode_t CLOCK_GetMode(void)**

This function checks the MCG_Lite registers and determines the current MCG_Lite mode.

Returns

The current MCG_Lite mode or error code.

Function Documentation

31.7.26 status_t CLOCK_SetMcgliteConfig (mcglite_config_t const * *targetConfig*)

This function configures the MCG_Lite, includes the output clock source, MCGIRCLK settings, HIRC settings, and so on. See [mcglite_config_t](#) for details.

Parameters

<i>targetConfig</i>	Pointer to the target MCG_Lite mode configuration structure.
---------------------	--

Returns

Error code.

31.7.27 static void OSC_SetExtRefClkConfig (OSC_Type * *base*, oscer_config_t const * *config*) [inline], [static]

This function configures the OSC external reference clock (OSCERCLK). This is an example to enable the OSCERCLK in normal mode and stop mode, and 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.

31.7.28 static void OSC_SetCapLoad (OSC_Type * *base*, uint8_t *capLoad*) [inline], [static]

This function sets the specified capacitor 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.
-------------	-------------------------

Variable Documentation

<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);
```

31.7.29 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.
---------------	--

31.7.30 void CLOCK_DeinitOsc0 (void)

This function deinitializes the OSC0.

31.7.31 static void CLOCK_SetXtal0Freq (uint32_t freq) [inline], [static]

Parameters

<i>freq</i>	The XTAL0/EXTAL0 input clock frequency in Hz.
-------------	---

31.7.32 static void CLOCK_SetXtal32Freq (uint32_t freq) [inline], [static]

Parameters

<i>freq</i>	The XTAL32/EXTAL32/RTC_CLKIN input clock frequency in Hz.
-------------	---

31.8 Variable Documentation

31.8.1 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_InitOsc(...); // Set up the OSC0  
* CLOCK_SetXtal0Freq(80000000); // Set the XTAL0 value to clock driver.  
*
```

This is important for the multicore platforms where 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.

31.8.2 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 one core needs to set up the clock. All other cores need to call the CLOCK_SetXtal32Freq to get a valid clock frequency.

Multipurpose Clock Generator Lite (MCGLITE)

31.9 Multipurpose Clock Generator Lite (MCGLITE)

The KSDK provides a peripheral driver for the MCG_Lite module of Kinetis devices.

31.9.1 Function description

The MCG_Lite driver provides three kinds of APIs:

1. APIs to get the MCG_Lite frequency.
2. APIs for MCG_Lite mode.
3. APIs for OSC setup.

31.9.1.1 MCG_Lite clock frequency

The [CLOCK_GetOutClkFreq\(\)](#), [CLOCK_GetInternalRefClkFreq\(\)](#) and [CLOCK_GetPeriphClkFreq\(\)](#) functions are used to get the frequency of MCGOUTCLK, MCGIRCLK, and MCGPCLK based on the current hardware setting.

31.9.1.2 MCG_Lite mode

The function [CLOCK_GetMode\(\)](#) gets the current MCG_Lite mode.

The function [CLOCK_SetMcgliteConfig\(\)](#) sets the MCG_Lite to a desired configuration. The MCG_Lite can't switch between the LIRC2M and LIRC8M. Instead, the function switches to the HIRC mode first and then switches to the target mode.

31.9.1.3 OSC configuration

To enable the OSC clock, the MCG_Lite is needed together with the OSC module. The function [CLOCK_InitOsc0\(\)](#) uses the MCG_Lite and the OSC to initialize the OSC. The OSC should be configured based on the board design.

Chapter 32

Debug Console

32.1 Overview

This part 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.

32.2 Function groups

32.2.1 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);
```

32.2.2 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 KSDK 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 */
```

32.3 Typical use case

Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

Typical use case

Some examples use the PRINTF function

Statement prints the string format.

```
PRINTF("%s %s\r\n", "Hello", "world!");
```

Statement prints the hexadecimal format/

```
PRINTF("0x%02X hexadecimal number equivalents 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\r\nTime: %u ticks %2.5f milliseconds\r\nDONE\r", "1 day", 86400, 86.4);
```

Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

Print out failure messages using KSDK __assert_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \"% %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file,
           line, func);
    for (;;)
    {}
}
```

Note:

To use 'printf' and 'scanf' for GNUC Base, add file '**fsl_sbrk.c**' in path: ..\{package}\devices\{subset}\utilities\fsl-sbrk.c to your project.

Modules

- Semihosting

32.4 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.

32.4.1 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.

32.4.2 Guide Semihosting for Keil µVision

NOTE: Keil supports Semihosting only for Cortex-M3/Cortex-M4 cores.

Step 1: Prepare code

Remove function `fputc` and `fgetc` is used to support KEIL in "fsl_debug_console.c" and add the following code to project.

```
#pragma import(__use_no_semihosting_swi)

volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY; /* used for Debug Input */
```

```
struct __FILE
{
    int handle;
};

FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f)
{
    return (ITM_SendChar(ch));
}

int fgetc(FILE *f)
{ /* blocking */
    while (ITM_CheckChar() != 1)
        ;
    return (ITM_ReceiveChar());
}

int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}

void _ttywrch(int ch)
{
    ITM_SendChar(ch);
}

void _sys_exit(int return_code)
{
label:
    goto label; /* endless loop */
}
```

Step 2: Setting up the environment

1. In menu bar, choose Project>Options for target or using Alt+F7 or click.
2. Select "Target" tab and not select "Use MicroLIB".
3. Select "Debug" tab, select "J-Link/J-Trace Cortex" and click "Setting button".
4. Select "Debug" tab and choose Port:SW, then select "Trace" tab, choose "Enable" and click OK.

Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

Step 4: Building the project

1. Choose "Debug" on menu bar or Ctrl F5.
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer".
3. Run line by line to see result in Console Window.

32.4.3 Guide Semihosting for KDS

NOTE: After the setting use "printf" for debugging.

Step 1: Setting up the environment

1. In menu bar, choose Project>Properties>C/C++ Build>Settings>Tool Settings.
2. Select “Libraries” on “Cross ARM C Linker” and delete “nosys”.
3. Select “Miscellaneous” on “Cross ARM C Linker”, add “-specs=rdimon.specs” to “Other link flags” and tick “Use newlib-nano”, and click OK.

Step 2: Building the project

1. In menu bar, choose Project>Build Project.

Step 3: Starting semihosting

1. In Debug configurations, choose "Startup" tab, tick “Enable semihosting and Telnet”. Press “Apply” and “Debug”.
2. After clicking Debug, the Window is displayed same as below. Run line by line to see the result in the Console Window.

32.4.4 Guide Semihosting for ATL

NOTE: J-Link has to be used to enable semihosting.

Step 1: Prepare code

Add the following code to the project.

```
int _write(int file, char *ptr, int len)
{
    /* Implement your write code here. This is used by puts and printf. */
    int i=0;
    for(i=0 ; i<len ; i++)
        ITM_SendChar((*ptr++));
    return len;
}
```

Step 2: Setting up the environment

1. In menu bar, choose Debug Configurations. In tab "Embedded C/C++ Application" choose "- Semihosting_ATL_xxx debug J-Link".
2. In tab "Debugger" set up as follows.
 - JTAG mode must be selected

Semihosting

- SWV tracing must be enabled
 - Enter the Core Clock frequency, which is hardware board-specific.
 - Enter the desired SWO Clock frequency. The latter depends on the JTAG Probe and must be a multiple of the Core Clock value.
3. Click "Apply" and "Debug".

Step 3: Starting semihosting

1. In the Views menu, expand the submenu SWV and open the docking view "SWV Console". 2. Open the SWV settings panel by clicking the "Configure Serial Wire Viewer" button in the SWV Console view toolbar. 3. Configure the data ports to be traced by enabling the ITM channel 0 check-box in the ITM stimulus ports group: Choose "EXETRC: Trace Exceptions" and In tab "ITM Stimulus Ports" choose "Enable Port" 0. Then click "OK".
2. It is recommended not to enable other SWV trace functionalities at the same time because this may over use the SWO pin causing packet loss due to a limited bandwidth (certain other SWV tracing capabilities can send a lot of data at very high-speed). Save the SWV configuration by clicking the OK button. The configuration is saved with other debug configurations and remains effective until changed.
3. Press the red Start/Stop Trace button to send the SWV configuration to the target board to enable SWV trace recording. The board does not send any SWV packages until it is properly configured. The SWV Configuration must be present, if the configuration registers on the target board are reset. Also, tracing does not start until the target starts to execute.
4. Start the target execution again by pressing the green Resume Debug button.
5. The SWV console now shows the printf() output.

32.4.5 Guide Semihosting for ARMGCC

Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
 - "Host Name (or IP address)" : localhost
 - "Port" :2333
 - "Connection type" : Telnet.
 - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

Add to "CMakeLists.txt"

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")  
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")  
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --
```

```
defsym=__heap_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}
--defsym=__heap_size__=0x2000")
```

Step 2: Building the project

1. Change "CMakeLists.txt":

Change "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "\${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=nano.specs")"
to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "\${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=rdimon.specs")"

Replace paragraph

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-common")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffunction-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fdata-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffreestanding")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-builtin")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mthumb")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mapcs")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
--gc-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-static")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-z")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
muldefs")
```

To

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
```

Semihosting

```
G} --specs=rdimon.specs ")  
Remove  
target_link_libraries(semihosting_ARMGCC.elf debug nosys)  
2. Run "build_debug.bat" to build project
```

Step 3: Starting semihosting

- (a) Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug  
d:  
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe  
target remote localhost:2331  
monitor reset  
monitor semihosting enable  
monitor semihosting thumbSWI 0xAB  
monitor semihosting IOClient 1  
monitor flash device = MK64FN1M0xxx12  
load semihosting_ARMGCC.elf  
monitor reg pc = (0x00000004)  
monitor reg sp = (0x00000000)  
continue
```

- (b) After the setting, press "enter". The PuTTY window now shows the printf() output.

Chapter 33

Notification Framework

33.1 Overview

This section describes the programming interface of the Notifier driver.

33.2 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

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.

33.3 Data Structure Documentation

33.3.1 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

33.3.1.0.0.35 Field Documentation

33.3.1.0.0.35.1 `notifier_user_config_t* notifier_notification_block_t::targetConfig`

33.3.1.0.0.35.2 `notifier_policy_t notifier_notification_block_t::policy`

33.3.1.0.0.35.3 `notifier_notification_type_t notifier_notification_block_t::notifyType`

33.3.2 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.

33.3.2.0.0.36 Field Documentation

33.3.2.0.0.36.1 `notifier_callback_t notifier_callback_config_t::callback`

33.3.2.0.0.36.2 `notifier_callback_type_t notifier_callback_config_t::callbackType`

33.3.2.0.0.36.3 `void* notifier_callback_config_t::callbackData`

33.3.3 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.

33.3.3.0.0.37 Field Documentation

33.3.3.0.0.37.1 `notifier_user_config_t notifier_handle_t::configsTable`**

33.3.3.0.0.37.2 `uint8_t notifier_handle_t::configsNumber`

33.3.3.0.0.37.3 `notifier_callback_config_t* notifier_handle_t::callbacksTable`

33.3.3.0.0.37.4 `uint8_t notifier_handle_t::callbacksNumber`

33.3.3.0.0.37.5 `uint8_t notifier_handle_t::errorCallbackIndex`

33.3.3.0.0.37.6 `uint8_t notifier_handle_t::currentConfigIndex`

33.3.3.0.0.37.7 `notifier_user_function_t notifier_handle_t::userFunction`

33.3.3.0.0.37.8 `void* notifier_handle_t::userData`

33.4 Typedef Documentation

33.4.1 **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.

33.4.2 **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.

33.4.3 **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.

33.5 Enumeration Type Documentation

33.5.1 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.

33.5.2 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.

33.5.3 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.

33.5.4 enum notifier_callback_type_t

Used in the callback configuration structure (`notifier_callback_config_t`) to specify when the registered callback is called during configuration switch initiated by the `NOTIFIER_SwitchConfig()`. Callback can be invoked in following situations.

- Before the configuration switch (Callback return value can affect `NOTIFIER_SwitchConfig()` execution. See the `NOTIFIER_SwitchConfig()` and `notifier_policy_t` documentation).
- After an unsuccessful attempt to switch configuration
- After a successful configuration switch

Enumerator

kNOTIFIER_CallbackBefore Callback handles BEFORE notification.

kNOTIFIER_CallbackAfter Callback handles AFTER notification.

kNOTIFIER_CallbackBeforeAfter Callback handles BEFORE and AFTER notification.

Function Documentation

33.6 Function Documentation

33.6.1 `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.

33.6.2 **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.

33.6.3 **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 34

Shell

34.1 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.

34.2 Function groups

34.2.1 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>> ");
```

34.2.2 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.

34.2.3 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](#) (6U)
Macro to set maximum count of commands.

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.

34.3 Data Structure Documentation

34.3.1 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**
Total history command in hist buff.
- **char hist_buf [SHELL_HIST_MAX][SHELL_BUFFER_SIZE]**

Data Structure Documentation

- *History buffer.*
• bool [exit](#)
Exit Flag.

34.3.2 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.

34.3.2.0.0.38 Field Documentation

34.3.2.0.0.38.1 const char* shell_command_context_t::pcCommand

For example "help". It must be all lower case.

34.3.2.0.0.38.2 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".

34.3.2.0.0.38.3 const cmd_function_t shell_command_context_t::pFuncCallBack

34.3.2.0.0.38.4 uint8_t shell_command_context_t::cExpectedNumberOfParameters

34.3.3 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.

34.4 Macro Definition Documentation

34.4.1 `#define SHELL_USE_HISTORY (0U)`

34.4.2 `#define SHELL_SEARCH_IN_HIST (1U)`

34.4.3 `#define SHELL_USE_FILE_STREAM (0U)`

34.4.4 `#define SHELL_AUTO_COMPLETE (1U)`

34.4.5 `#define SHELL_BUFFER_SIZE (64U)`

34.4.6 `#define SHELL_MAX_ARGS (8U)`

34.4.7 `#define SHELL_HIST_MAX (3U)`

34.4.8 `#define SHELL_MAX_CMD (6U)`

34.5 Typedef Documentation

34.5.1 `typedef void(* send_data_cb_t)(uint8_t *buf, uint32_t len)`

34.5.2 `typedef void(* recv_data_cb_t)(uint8_t *buf, uint32_t len)`

34.5.3 `typedef int(* printf_data_t)(const char *format,...)`

34.5.4 `typedef int32_t(* cmd_function_t)(p_shell_context_t context, int32_t argc, char **argv)`

34.6 Enumeration Type Documentation

34.6.1 `enum fun_key_status_t`

Enumerator

kSHELL_Normal Normal key.

kSHELL_Special Special key.

kSHELL_Function Function key.

Function Documentation

34.7 Function Documentation

34.7.1 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

34.7.2 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

34.7.3 int32_t SHELL_Main (*p_shell_context_t context*)

Main loop for Shell; After this function is called, Shell begins to initialize the basic variables and starts to work.

Parameters

<i>context</i>	The pointer to the Shell environment and runtime states.
----------------	--

Returns

This function does not return until Shell command exit was called.

Function Documentation

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 Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address:
freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, Kinetis, Processor Expert are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Tower is a trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM, ARM Powered logo, and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2016 Freescale Semiconductor, Inc.

