

Document Number: MCUXSDKAPIRM
Rev 2.12.0
Jul 2022

MCUXpresso SDK API Reference Manual

NXP Semiconductors



Contents

Chapter 1 Introduction

Chapter 2 Trademarks

Chapter 3 Architectural Overview

Chapter 4 Clock Driver

4.1 Overview	7
4.2 Data Structure Documentation	14
4.2.1 struct scg_sys_clk_config_t	14
4.2.2 struct scg_sosc_config_t	15
4.2.3 struct scg_sirc_config_t	16
4.2.4 struct scg_firc_trim_config_t	16
4.2.5 struct scg_firc_config_t	17
4.2.6 struct scg_lpfll_trim_config_t	17
4.2.7 struct scg_lpfll_config_t	18
4.3 Macro Definition Documentation	18
4.3.1 FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL	18
4.3.2 FSL_CLOCK_DRIVER_VERSION	18
4.3.3 DMAMUX_CLOCKS	18
4.3.4 PORT_CLOCKS	19
4.3.5 LPI2C_CLOCKS	19
4.3.6 FLEXIO_CLOCKS	19
4.3.7 TSI_CLOCKS	19
4.3.8 EDMA_CLOCKS	19
4.3.9 LPUART_CLOCKS	20
4.3.10 LPTMR_CLOCKS	20
4.3.11 ADC12_CLOCKS	20
4.3.12 LPSPI_CLOCKS	20
4.3.13 LPIT_CLOCKS	20
4.3.14 CRC_CLOCKS	21
4.3.15 CMP_CLOCKS	21
4.3.16 FLASH_CLOCKS	21
4.3.17 EWM_CLOCKS	21
4.3.18 FTM_CLOCKS	21

Section No.	Title	Page No.
4.3.19	PWT_CLOCKS	22
4.3.20	FLEXIOTRIG_CLOCKS	22
4.3.21	CLOCK_GetOsc0ErClkFreq	22
4.4	Enumeration Type Documentation	22
4.4.1	clock_name_t	22
4.4.2	clock_ip_src_t	22
4.4.3	clock_ip_name_t	23
4.4.4	anonymous enum	23
4.4.5	scg_sys_clk_t	23
4.4.6	scg_sys_clk_src_t	23
4.4.7	scg_sys_clk_div_t	23
4.4.8	clock_clkout_src_t	24
4.4.9	scg_async_clk_t	24
4.4.10	scg_async_clk_div_t	24
4.4.11	scg_sosc_monitor_mode_t	25
4.4.12	scg_sosc_mode_t	25
4.4.13	anonymous enum	25
4.4.14	scg_sirc_range_t	25
4.4.15	anonymous enum	25
4.4.16	scg_firc_trim_mode_t	26
4.4.17	scg_firc_trim_div_t	26
4.4.18	scg_firc_trim_src_t	26
4.4.19	scg_firc_range_t	26
4.4.20	anonymous enum	26
4.4.21	anonymous enum	27
4.4.22	scg_lpfl_range_t	27
4.4.23	scg_lpfl_trim_mode_t	27
4.4.24	scg_lpfl_trim_src_t	27
4.4.25	scg_lpfl_lock_mode_t	27
4.5	Function Documentation	28
4.5.1	CLOCK_EnableClock	28
4.5.2	CLOCK_DisableClock	29
4.5.3	CLOCK_SetIpSrc	29
4.5.4	CLOCK_SetIpSrcDiv	29
4.5.5	CLOCK_GetFreq	30
4.5.6	CLOCK_GetCoreSysClkFreq	30
4.5.7	CLOCK_GetBusClkFreq	30
4.5.8	CLOCK_GetFlashClkFreq	30
4.5.9	CLOCK_GetErClkFreq	30
4.5.10	CLOCK_GetIpFreq	31
4.5.11	CLOCK_GetSysClkFreq	32
4.5.12	CLOCK_SetVlprModeSysClkConfig	32
4.5.13	CLOCK_SetRunModeSysClkConfig	32

Section No.	Title	Page No.
4.5.14	<code>CLOCK_GetCurSysClkConfig</code>	33
4.5.15	<code>CLOCK_SetClkOutSel</code>	33
4.5.16	<code>CLOCK_InitSysOsc</code>	33
4.5.17	<code>CLOCK_DeinitSysOsc</code>	34
4.5.18	<code>CLOCK_SetSysOscAsyncClkDiv</code>	34
4.5.19	<code>CLOCK_GetSysOscFreq</code>	34
4.5.20	<code>CLOCK_GetSysOscAsyncFreq</code>	35
4.5.21	<code>CLOCK_IsSysOscErr</code>	36
4.5.22	<code>CLOCK_SetSysOscMonitorMode</code>	36
4.5.23	<code>CLOCK_IsSysOscValid</code>	36
4.5.24	<code>CLOCK_InitSirc</code>	36
4.5.25	<code>CLOCK_DeinitSirc</code>	37
4.5.26	<code>CLOCK_SetSircAsyncClkDiv</code>	37
4.5.27	<code>CLOCK_GetSircFreq</code>	38
4.5.28	<code>CLOCK_GetSircAsyncFreq</code>	38
4.5.29	<code>CLOCK_IsSircValid</code>	38
4.5.30	<code>CLOCK_InitFirc</code>	38
4.5.31	<code>CLOCK_DeinitFirc</code>	39
4.5.32	<code>CLOCK_SetFircAsyncClkDiv</code>	40
4.5.33	<code>CLOCK_GetFircFreq</code>	40
4.5.34	<code>CLOCK_GetFircAsyncFreq</code>	40
4.5.35	<code>CLOCK_IsFircValid</code>	41
4.5.36	<code>CLOCK_InitLpFll</code>	41
4.5.37	<code>CLOCK_DeinitLpFll</code>	41
4.5.38	<code>CLOCK_SetLpFllAsyncClkDiv</code>	41
4.5.39	<code>CLOCK_GetLpFllFreq</code>	42
4.5.40	<code>CLOCK_GetLpFllAsyncFreq</code>	42
4.5.41	<code>CLOCK_IsLpFllValid</code>	42
4.5.42	<code>CLOCK_SetXtal0Freq</code>	42
4.6	Variable Documentation	43
4.6.1	<code>g_xtal0Freq</code>	43
4.7	System Clock Generator (SCG)	44
4.7.1	Function description	44
4.7.2	Typical use case	46

Chapter 5 ACMP: Analog Comparator Driver

5.1	Overview	48
5.2	Typical use case	48
5.2.1	Normal Configuration	48
5.2.2	Interrupt Configuration	48
5.2.3	Round robin Configuration	48

Section No.	Title	Page No.
5.3	Data Structure Documentation	50
5.3.1	struct acmp_config_t	51
5.3.2	struct acmp_channel_config_t	51
5.3.3	struct acmp_filter_config_t	52
5.3.4	struct acmp_dac_config_t	52
5.3.5	struct acmp_round_robin_config_t	53
5.4	Macro Definition Documentation	53
5.4.1	FSL_ACMP_DRIVER_VERSION	53
5.4.2	CMP_C0_CFx_MASK	53
5.5	Enumeration Type Documentation	53
5.5.1	_acmp_interrupt_enable	53
5.5.2	_acmp_status_flags	54
5.5.3	acmp_offset_mode_t	54
5.5.4	acmp_hysteresis_mode_t	54
5.5.5	acmp_reference_voltage_source_t	54
5.5.6	acmp_port_input_t	55
5.5.7	acmp_fixed_port_t	55
5.6	Function Documentation	55
5.6.1	ACMP_Init	55
5.6.2	ACMP_Deinit	55
5.6.3	ACMP_GetDefaultConfig	55
5.6.4	ACMP_Enable	56
5.6.5	ACMP_SetChannelConfig	56
5.6.6	ACMP_EnableDMA	56
5.6.7	ACMP_EnableWindowMode	57
5.6.8	ACMP_SetFilterConfig	57
5.6.9	ACMP_SetDACConfig	57
5.6.10	ACMP_SetRoundRobinConfig	58
5.6.11	ACMP_SetRoundRobinPreState	58
5.6.12	ACMP_GetRoundRobinStatusFlags	58
5.6.13	ACMP_ClearRoundRobinStatusFlags	59
5.6.14	ACMP_GetRoundRobinResult	59
5.6.15	ACMP_EnableInterrupts	59
5.6.16	ACMP_DisableInterrupts	60
5.6.17	ACMP_GetStatusFlags	60
5.6.18	ACMP_ClearStatusFlags	60
Chapter 6	ADC12: Analog-to-Digital Converter	
6.1	Overview	61
6.2	Function groups	61

Section No.	Title	Page No.
6.2.1	Initialization and deinitialization	61
6.2.2	Basic Operations	61
6.2.3	Advanced Operations	61
6.3	Typical use case	61
6.3.1	Normal Configuration	61
6.3.2	Interrupt Configuration	62
6.4	Data Structure Documentation	64
6.4.1	struct adc12_config_t	64
6.4.2	struct adc12_hardware_compare_config_t	64
6.4.3	struct adc12_channel_config_t	65
6.5	Macro Definition Documentation	65
6.5.1	FSL_ADC12_DRIVER_VERSION	65
6.6	Enumeration Type Documentation	65
6.6.1	_adc12_channel_status_flags	65
6.6.2	_adc12_status_flags	65
6.6.3	adc12_clock_divider_t	66
6.6.4	adc12_resolution_t	66
6.6.5	adc12_clock_source_t	66
6.6.6	adc12_reference_voltage_source_t	66
6.6.7	adc12_hardware_average_mode_t	66
6.6.8	adc12_hardware_compare_mode_t	67
6.7	Function Documentation	67
6.7.1	ADC12_Init	67
6.7.2	ADC12_Deinit	67
6.7.3	ADC12_GetDefaultConfig	67
6.7.4	ADC12_SetChannelConfig	68
6.7.5	ADC12_GetChannelConversionValue	68
6.7.6	ADC12_GetChannelStatusFlags	69
6.7.7	ADC12_DoAutoCalibration	69
6.7.8	ADC12_SetOffsetValue	69
6.7.9	ADC12_SetGainValue	70
6.7.10	ADC12_EnableDMA	70
6.7.11	ADC12_EnableHardwareTrigger	70
6.7.12	ADC12_SetHardwareCompareConfig	71
6.7.13	ADC12_SetHardwareAverage	71
6.7.14	ADC12_GetStatusFlags	71
Chapter 7	Common Driver	72
7.1	Overview	72

Section No.	Title	Page No.
7.2	Macro Definition Documentation	74
7.2.1	FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ	74
7.2.2	MAKE_STATUS	74
7.2.3	MAKE_VERSION	75
7.2.4	FSL_COMMON_DRIVER_VERSION	75
7.2.5	DEBUG_CONSOLE_DEVICE_TYPE_NONE	75
7.2.6	DEBUG_CONSOLE_DEVICE_TYPE_UART	75
7.2.7	DEBUG_CONSOLE_DEVICE_TYPE_LPUART	75
7.2.8	DEBUG_CONSOLE_DEVICE_TYPE_LPSCI	75
7.2.9	DEBUG_CONSOLE_DEVICE_TYPE_USBCDC	75
7.2.10	DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM	75
7.2.11	DEBUG_CONSOLE_DEVICE_TYPE_IUART	75
7.2.12	DEBUG_CONSOLE_DEVICE_TYPE_VUSART	75
7.2.13	DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART	75
7.2.14	DEBUG_CONSOLE_DEVICE_TYPE_SWO	75
7.2.15	DEBUG_CONSOLE_DEVICE_TYPE_QSCI	75
7.2.16	ARRAY_SIZE	75
7.3	Typedef Documentation	75
7.3.1	status_t	75
7.4	Enumeration Type Documentation	76
7.4.1	_status_groups	76
7.4.2	anonymous enum	78
7.5	Function Documentation	79
7.5.1	SDK_Malloc	79
7.5.2	SDK_Free	79
7.5.3	SDK_DelayAtLeastUs	79
Chapter 8 CRC: Cyclic Redundancy Check Driver		
8.1	Overview	80
8.2	CRC Driver Initialization and Configuration	80
8.3	CRC Write Data	80
8.4	CRC Get Checksum	80
8.5	Comments about API usage in RTOS	81
8.6	Data Structure Documentation	82
8.6.1	struct crc_config_t	82
8.7	Macro Definition Documentation	83

Section No.	Title	Page No.
8.7.1	FSL_CRC_DRIVER_VERSION	83
8.7.2	CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT	83
8.8	Enumeration Type Documentation	83
8.8.1	crc_bits_t	83
8.8.2	crc_result_t	83
8.9	Function Documentation	83
8.9.1	CRC_Init	83
8.9.2	CRC_Deinit	84
8.9.3	CRC_GetDefaultConfig	84
8.9.4	CRC_WriteData	84
8.9.5	CRC_Get32bitResult	85
8.9.6	CRC_Get16bitResult	85

Chapter 9 DMAMUX: Direct Memory Access Multiplexer Driver

9.1	Overview	86
9.2	Typical use case	86
9.2.1	DMAMUX Operation	86
9.3	Macro Definition Documentation	86
9.3.1	FSL_DMAMUX_DRIVER_VERSION	86
9.4	Function Documentation	86
9.4.1	DMAMUX_Init	87
9.4.2	DMAMUX_Deinit	88
9.4.3	DMAMUX_EnableChannel	88
9.4.4	DMAMUX_DisableChannel	88
9.4.5	DMAMUX_SetSource	89
9.4.6	DMAMUX_EnablePeriodTrigger	89
9.4.7	DMAMUX_DisablePeriodTrigger	89

Chapter 10 eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

10.1	Overview	90
10.2	Typical use case	90
10.2.1	eDMA Operation	90
10.3	Data Structure Documentation	95
10.3.1	struct edma_config_t	95
10.3.2	struct edma_transfer_config_t	96
10.3.3	struct edma_channel_Preemption_config_t	97
10.3.4	struct edma_minor_offset_config_t	97

Section No.	Title	Page No.
10.3.5	struct edma_tcd_t	97
10.3.6	struct edma_handle_t	98
10.4	Macro Definition Documentation	99
10.4.1	FSL_EDMA_DRIVER_VERSION	99
10.5	Typedef Documentation	99
10.5.1	edma_callback	99
10.6	Enumeration Type Documentation	100
10.6.1	edma_transfer_size_t	100
10.6.2	edma_modulo_t	100
10.6.3	edma_bandwidth_t	101
10.6.4	edma_channel_link_type_t	101
10.6.5	anonymous enum	101
10.6.6	anonymous enum	102
10.6.7	edma_interrupt_enable_t	102
10.6.8	edma_transfer_type_t	102
10.6.9	anonymous enum	102
10.7	Function Documentation	103
10.7.1	EDMA_Init	103
10.7.2	EDMA_Deinit	104
10.7.3	EDMA_InstallTCD	104
10.7.4	EDMA_GetDefaultConfig	104
10.7.5	EDMA_EnableContinuousChannelLinkMode	105
10.7.6	EDMA_EnableMinorLoopMapping	105
10.7.7	EDMA_ResetChannel	105
10.7.8	EDMA_SetTransferConfig	106
10.7.9	EDMA_SetMinorOffsetConfig	106
10.7.10	EDMA_SetChannelPreemptionConfig	107
10.7.11	EDMA_SetChannelLink	107
10.7.12	EDMA_SetBandWidth	108
10.7.13	EDMA_SetModulo	108
10.7.14	EDMA_EnableAsyncRequest	109
10.7.15	EDMA_EnableAutoStopRequest	109
10.7.16	EDMA_EnableChannelInterrupts	109
10.7.17	EDMA_DisableChannelInterrupts	109
10.7.18	EDMA_SetMajorOffsetConfig	110
10.7.19	EDMA_TcdReset	110
10.7.20	EDMA_TcdSetTransferConfig	110
10.7.21	EDMA_TcdSetMinorOffsetConfig	111
10.7.22	EDMA_TcdSetChannelLink	111
10.7.23	EDMA_TcdSetBandWidth	112
10.7.24	EDMA_TcdSetModulo	112

Section No.	Title	Page No.
10.7.25	EDMA_TcdEnableAutoStopRequest	113
10.7.26	EDMA_TcdEnableInterrupts	114
10.7.27	EDMA_TcdDisableInterrupts	114
10.7.28	EDMA_TcdSetMajorOffsetConfig	114
10.7.29	EDMA_EnableChannelRequest	114
10.7.30	EDMA_DisableChannelRequest	115
10.7.31	EDMA_TriggerChannelStart	115
10.7.32	EDMA_GetRemainingMajorLoopCount	115
10.7.33	EDMA_GetErrorStatusFlags	116
10.7.34	EDMA_GetChannelStatusFlags	116
10.7.35	EDMA_ClearChannelStatusFlags	116
10.7.36	EDMA_CreateHandle	117
10.7.37	EDMA_InstallTCDMemory	117
10.7.38	EDMA_SetCallback	117
10.7.39	EDMA_PreparesTransferConfig	118
10.7.40	EDMA_PreparesTransfer	118
10.7.41	EDMA_SubmitTransfer	119
10.7.42	EDMA_StartTransfer	120
10.7.43	EDMA_StopTransfer	121
10.7.44	EDMA_AbortTransfer	121
10.7.45	EDMA_GetUnusedTCDNumber	121
10.7.46	EDMA_GetNextTCDAddress	121
10.7.47	EDMA_HandleIRQ	122

Chapter 11 EWM: External Watchdog Monitor Driver

11.1	Overview	123
11.2	Typical use case	123
11.3	Data Structure Documentation	124
11.3.1	struct ewm_config_t	124
11.4	Macro Definition Documentation	124
11.4.1	FSL_EWM_DRIVER_VERSION	124
11.5	Enumeration Type Documentation	124
11.5.1	_ewm_interrupt_enable_t	124
11.5.2	_ewm_status_flags_t	124
11.6	Function Documentation	125
11.6.1	EWM_Init	125
11.6.2	EWM_Deinit	125
11.6.3	EWM_GetDefaultConfig	125
11.6.4	EWM_EnableInterrupts	126

Section No.	Title	Page No.
11.6.5	EWM_DisableInterrupts	126
11.6.6	EWM_GetStatusFlags	126
11.6.7	EWM_Refresh	127

Chapter 12 C90TFS Flash Driver

12.1	Overview	128
12.2	Ftftx FLASH Driver	129
12.2.1	Overview	129
12.2.2	Data Structure Documentation	131
12.2.3	Macro Definition Documentation	132
12.2.4	Enumeration Type Documentation	132
12.2.5	Function Documentation	133
12.3	Ftftx CACHE Driver	148
12.3.1	Overview	148
12.3.2	Data Structure Documentation	148
12.3.3	Enumeration Type Documentation	149
12.3.4	Function Documentation	149
12.4	Ftftx FLEXNVM Driver	152
12.4.1	Overview	152
12.4.2	Data Structure Documentation	154
12.4.3	Enumeration Type Documentation	154
12.4.4	Function Documentation	154
12.5	ftfx feature	168
12.5.1	Overview	168
12.5.2	Macro Definition Documentation	168
12.5.3	ftfx adapter	169
12.6	ftfx controller	170
12.6.1	Overview	170
12.6.2	Data Structure Documentation	173
12.6.3	Macro Definition Documentation	175
12.6.4	Enumeration Type Documentation	175
12.6.5	Function Documentation	177
12.6.6	ftfx utilities	189

Chapter 13 FlexIO: FlexIO Driver

13.1	Overview	190
13.2	FlexIO Driver	191
13.2.1	Overview	191

Section No.	Title	Page No.
13.2.2	Data Structure Documentation	195
13.2.3	Macro Definition Documentation	198
13.2.4	Typedef Documentation	198
13.2.5	Enumeration Type Documentation	198
13.2.6	Function Documentation	202
13.2.7	Variable Documentation	212
13.3	FlexIO I2C Master Driver	213
13.3.1	Overview	213
13.3.2	Typical use case	213
13.3.3	Data Structure Documentation	217
13.3.4	Macro Definition Documentation	219
13.3.5	Typedef Documentation	219
13.3.6	Enumeration Type Documentation	220
13.3.7	Function Documentation	220
13.4	FlexIO SPI Driver	230
13.4.1	Overview	230
13.4.2	Typical use case	230
13.4.3	Data Structure Documentation	237
13.4.4	Macro Definition Documentation	240
13.4.5	Typedef Documentation	241
13.4.6	Enumeration Type Documentation	241
13.4.7	Function Documentation	242
13.4.8	FlexIO eDMA SPI Driver	257
13.5	FlexIO UART Driver	263
13.5.1	Overview	263
13.5.2	Typical use case	263
13.5.3	Data Structure Documentation	272
13.5.4	Macro Definition Documentation	274
13.5.5	Typedef Documentation	274
13.5.6	Enumeration Type Documentation	275
13.5.7	Function Documentation	276
13.5.8	FlexIO eDMA UART Driver	287

Chapter 14 FTM: FlexTimer Driver

14.1	Overview	293
14.2	Function groups	293
14.2.1	Initialization and deinitialization	293
14.2.2	PWM Operations	293
14.2.3	Input capture operations	293
14.2.4	Output compare operations	294

Section No.	Title	Page No.
14.2.5	Quad decode	294
14.2.6	Fault operation	294
14.3	Register Update	294
14.4	Typical use case	294
14.4.1	PWM output	295
14.5	Data Structure Documentation	301
14.5.1	struct ftm_chnl_pwm_signal_param_t	301
14.5.2	struct ftm_chnl_pwm_config_param_t	302
14.5.3	struct ftm_dual_edge_capture_param_t	303
14.5.4	struct ftm_phase_params_t	303
14.5.5	struct ftm_fault_param_t	303
14.5.6	struct ftm_config_t	304
14.6	Macro Definition Documentation	305
14.6.1	FSL_FTM_DRIVER_VERSION	305
14.7	Enumeration Type Documentation	305
14.7.1	ftm_chnl_t	305
14.7.2	ftm_fault_input_t	305
14.7.3	ftm_pwm_mode_t	305
14.7.4	ftm_pwm_level_select_t	306
14.7.5	ftm_output_compare_mode_t	306
14.7.6	ftm_input_capture_edge_t	306
14.7.7	ftm_dual_edge_capture_mode_t	306
14.7.8	ftm_quad_decode_mode_t	306
14.7.9	ftm_phase_polarity_t	307
14.7.10	ftm_deadtime_prescale_t	307
14.7.11	ftm_clock_source_t	307
14.7.12	ftm_clock_prescale_t	307
14.7.13	ftm_bdm_mode_t	307
14.7.14	ftm_fault_mode_t	308
14.7.15	ftm_external_trigger_t	308
14.7.16	ftm_pwm_sync_method_t	308
14.7.17	ftm_reload_point_t	309
14.7.18	ftm_interrupt_enable_t	309
14.7.19	ftm_status_flags_t	310
14.8	Function Documentation	310
14.8.1	FTM_Init	310
14.8.2	FTM_Deinit	310
14.8.3	FTM_GetDefaultConfig	311
14.8.4	FTM_CalculateCounterClkDiv	311
14.8.5	FTM_SetupPwm	311

Section No.	Title	Page No.
14.8.6	FTM_UpdatePwmDutyCycle	312
14.8.7	FTM_UpdateChnlEdgeLevelSelect	312
14.8.8	FTM_SetupPwmMode	313
14.8.9	FTM_SetupInputCapture	313
14.8.10	FTM_SetupOutputCompare	314
14.8.11	FTM_SetupDualEdgeCapture	314
14.8.12	FTM_SetupFaultInput	315
14.8.13	FTM_EnableInterrupts	315
14.8.14	FTM_DisableInterrupts	315
14.8.15	FTM_GetEnabledInterrupts	315
14.8.16	FTM_GetStatusFlags	316
14.8.17	FTM_ClearStatusFlags	316
14.8.18	FTM_SetTimerPeriod	316
14.8.19	FTM_GetCurrentTimerCount	317
14.8.20	FTM_GetInputCaptureValue	317
14.8.21	FTM_StartTimer	318
14.8.22	FTM_StopTimer	318
14.8.23	FTM_SetSoftwareCtrlEnable	318
14.8.24	FTM_SetSoftwareCtrlVal	318
14.8.25	FTM_SetGlobalTimeBaseOutputEnable	319
14.8.26	FTM_SetOutputMask	319
14.8.27	FTM_SetPwmOutputEnable	319
14.8.28	FTM_SetFaultControlEnable	320
14.8.29	FTM_SetDeadTimeEnable	320
14.8.30	FTM_SetComplementaryEnable	320
14.8.31	FTM_SetInvertEnable	321
14.8.32	FTM_SetupQuadDecode	321
14.8.33	FTM_SetQuadDecoderModuloValue	321
14.8.34	FTM_GetQuadDecoderCounterValue	322
14.8.35	FTM_ClearQuadDecoderCounterValue	322
14.8.36	FTM_SetSoftwareTrigger	322
14.8.37	FTM_SetWriteProtection	322
14.8.38	FTM_EnableDmaTransfer	323

Chapter 15 GPIO: General-Purpose Input/Output Driver

15.1	Overview	324
15.2	Data Structure Documentation	324
15.2.1	struct gpio_pin_config_t	324
15.3	Macro Definition Documentation	325
15.3.1	FSL_GPIO_DRIVER_VERSION	325
15.4	Enumeration Type Documentation	325

Section No.	Title	Page No.
15.4.1	gpio_pin_direction_t	325
15.5	GPIO Driver	326
15.5.1	Overview	326
15.5.2	Typical use case	326
15.5.3	Function Documentation.....	327
15.6	FGPIO Driver	330
15.6.1	Overview	330
15.6.2	Typical use case	330
15.6.3	Function Documentation.....	331
Chapter 16 LPI2C: Low Power Inter-Integrated Circuit Driver		
16.1	Overview	334
16.2	Macro Definition Documentation	334
16.2.1	FSL_LPI2C_DRIVER_VERSION.....	334
16.2.2	I2C_RETRY_TIMES	335
16.3	Enumeration Type Documentation	335
16.3.1	anonymous enum	335
16.4	LPI2C Master Driver	336
16.4.1	Overview	336
16.4.2	Data Structure Documentation	339
16.4.3	Typedef Documentation	343
16.4.4	Enumeration Type Documentation	344
16.4.5	Function Documentation.....	346
16.5	LPI2C Slave Driver	360
16.5.1	Overview	360
16.5.2	Data Structure Documentation	362
16.5.3	Typedef Documentation	365
16.5.4	Enumeration Type Documentation	367
16.5.5	Function Documentation.....	368
16.6	LPI2C Master DMA Driver	377
16.6.1	Overview	377
16.6.2	Data Structure Documentation	377
16.6.3	Typedef Documentation	378
16.6.4	Function Documentation.....	380
16.7	LPI2C FreeRTOS Driver	383
16.7.1	Overview	383
16.7.2	Macro Definition Documentation	383

Section No.	Title	Page No.
16.7.3	Function Documentation	383
16.8	LPI2C CMSIS Driver	386
16.8.1	LPI2C CMSIS Driver	386
Chapter 17 LPIT: Low-Power Interrupt Timer		
17.1	Overview	388
17.2	Function groups	388
17.2.1	Initialization and deinitialization	388
17.2.2	Timer period Operations	388
17.2.3	Start and Stop timer operations	388
17.2.4	Status	389
17.2.5	Interrupt	389
17.3	Typical use case	389
17.3.1	LPIT tick example	389
17.4	Data Structure Documentation	391
17.4.1	struct lpit_chnl_params_t	391
17.4.2	struct lpit_config_t	392
17.5	Enumeration Type Documentation	392
17.5.1	lpit_chnl_t	392
17.5.2	lpit_timer_modes_t	392
17.5.3	lpit_trigger_select_t	393
17.5.4	lpit_trigger_source_t	393
17.5.5	lpit_interrupt_enable_t	393
17.5.6	lpit_status_flags_t	394
17.6	Function Documentation	394
17.6.1	LPIT_Init	394
17.6.2	LPIT_Deinit	394
17.6.3	LPIT_GetDefaultConfig	394
17.6.4	LPIT_SetupChannel	395
17.6.5	LPIT_EnableInterrupts	395
17.6.6	LPIT_DisableInterrupts	395
17.6.7	LPIT_GetEnabledInterrupts	396
17.6.8	LPIT_GetStatusFlags	397
17.6.9	LPIT_ClearStatusFlags	397
17.6.10	LPIT_SetTimerPeriod	397
17.6.11	LPIT_GetCurrentTimerCount	398
17.6.12	LPIT_StartTimer	398
17.6.13	LPIT_StopTimer	398
17.6.14	LPIT_Reset	399

Section No.	Title	Page No.
Chapter 18 LPSPI: Low Power Serial Peripheral Interface		
18.1	Overview	400
18.2	LPSPI Peripheral driver	401
18.2.1	Overview	401
18.2.2	Function groups	401
18.2.3	Typical use case	401
18.2.4	Data Structure Documentation	408
18.2.5	Macro Definition Documentation	414
18.2.6	Typedef Documentation	414
18.2.7	Enumeration Type Documentation	415
18.2.8	Function Documentation	420
18.2.9	Variable Documentation	435
18.3	LPSPI eDMA Driver	436
18.3.1	Overview	436
18.3.2	Data Structure Documentation	437
18.3.3	Macro Definition Documentation	441
18.3.4	Typedef Documentation	441
18.3.5	Function Documentation	441
18.4	LPSPI FreeRTOS Driver	447
18.4.1	Overview	447
18.4.2	Macro Definition Documentation	447
18.4.3	Function Documentation	447
18.5	LPSPI CMSIS Driver	450
18.5.1	Function groups	450
18.5.2	Typical use case	451
Chapter 19 LPTMR: Low-Power Timer		
19.1	Overview	452
19.2	Function groups	452
19.2.1	Initialization and deinitialization	452
19.2.2	Timer period Operations	452
19.2.3	Start and Stop timer operations	452
19.2.4	Status	453
19.2.5	Interrupt	453
19.3	Typical use case	453
19.3.1	LPTMR tick example	453
19.4	Data Structure Documentation	455

Section No.	Title	Page No.
19.4.1	struct lptmr_config_t	455
19.5	Enumeration Type Documentation	456
19.5.1	lptmr_pin_select_t	456
19.5.2	lptmr_pin_polarity_t	456
19.5.3	lptmr_timer_mode_t	456
19.5.4	lptmr_prescaler_glitch_value_t	456
19.5.5	lptmr_prescaler_clock_select_t	457
19.5.6	lptmr_interrupt_enable_t	457
19.5.7	lptmr_status_flags_t	457
19.6	Function Documentation	457
19.6.1	LPTMR_Init	457
19.6.2	LPTMR_Deinit	458
19.6.3	LPTMR_GetDefaultConfig	458
19.6.4	LPTMR_EnableInterrupts	458
19.6.5	LPTMR_DisableInterrupts	459
19.6.6	LPTMR_GetEnabledInterrupts	459
19.6.7	LPTMR_EnableTimerDMA	459
19.6.8	LPTMR_GetStatusFlags	459
19.6.9	LPTMR_ClearStatusFlags	460
19.6.10	LPTMR_SetTimerPeriod	460
19.6.11	LPTMR_GetCurrentTimerCount	460
19.6.12	LPTMR_StartTimer	461
19.6.13	LPTMR_StopTimer	461

Chapter 20 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

20.1	Overview	462
20.2	LPUART Driver	463
20.2.1	Overview	463
20.2.2	Typical use case	463
20.2.3	Data Structure Documentation	468
20.2.4	Macro Definition Documentation	471
20.2.5	Typedef Documentation	471
20.2.6	Enumeration Type Documentation	471
20.2.7	Function Documentation	474
20.3	LPUART eDMA Driver	491
20.3.1	Overview	491
20.3.2	Data Structure Documentation	492
20.3.3	Macro Definition Documentation	492
20.3.4	Typedef Documentation	492
20.3.5	Function Documentation	493

Section No.	Title	Page No.
20.4 LPUART FreeRTOS Driver		497
20.4.1 Overview		497
20.4.2 Data Structure Documentation		497
20.4.3 Macro Definition Documentation		498
20.4.4 Function Documentation		498
20.5 LPUART CMSIS Driver		501
20.5.1 Function groups		501

Chapter 21 PMC: Power Management Controller

21.1 Overview		503
21.2 Data Structure Documentation		503
21.2.1 struct pmc_low_volt_detect_config_t		503
21.2.2 struct pmc_low_volt_warning_config_t		504
21.3 Macro Definition Documentation		504
21.3.1 FSL_PMC_DRIVER_VERSION		504
21.4 Function Documentation		504
21.4.1 PMC_ConfigureLowVoltDetect		504
21.4.2 PMC_GetLowVoltDetectFlag		504
21.4.3 PMC_ClearLowVoltDetectFlag		505
21.4.4 PMC_ConfigureLowVoltWarning		505
21.4.5 PMC_GetLowVoltWarningFlag		505
21.4.6 PMC_ClearLowVoltWarningFlag		506

Chapter 22 PORT: Port Control and Interrupts

22.1 Overview		508
22.2 Data Structure Documentation		510
22.2.1 struct port_digital_filter_config_t		510
22.2.2 struct port_pin_config_t		510
22.3 Macro Definition Documentation		511
22.3.1 FSL_PORT_DRIVER_VERSION		511
22.4 Enumeration Type Documentation		511
22.4.1 _port_pull		511
22.4.2 _port_passive_filter_enable		511
22.4.3 _port_drive_strength		511
22.4.4 _port_lock_register		511
22.4.5 port_mux_t		511
22.4.6 port_interrupt_t		512

Section No.	Title	Page No.
22.4.7	<code>port_digital_filter_clock_source_t</code>	512
22.5	Function Documentation	512
22.5.1	<code>PORT_SetPinConfig</code>	513
22.5.2	<code>PORT_SetMultiplePinsConfig</code>	513
22.5.3	<code>PORT_SetPinMux</code>	514
22.5.4	<code>PORT_EnablePinsDigitalFilter</code>	514
22.5.5	<code>PORT_SetDigitalFilterConfig</code>	515
22.5.6	<code>PORT_SetPinInterruptConfig</code>	515
22.5.7	<code>PORT_SetPinDriveStrength</code>	516
22.5.8	<code>PORT_GetPinsInterruptFlags</code>	516
22.5.9	<code>PORT_ClearPinsInterruptFlags</code>	516

Chapter 23 PWT: Pulse Width Timer

23.1	Overview	517
23.2	Function groups	517
23.2.1	<code>Initialization and deinitialization</code>	517
23.2.2	<code>Reset</code>	517
23.2.3	<code>Status</code>	517
23.2.4	<code>Interrupt</code>	517
23.2.5	<code>Start & Stop timer</code>	517
23.2.6	<code>GetInterrupt</code>	518
23.2.7	<code>Get Timer value</code>	518
23.2.8	<code>PWT Operations</code>	518
23.3	Typical use case	518
23.3.1	<code>PWT measure</code>	518
23.4	Data Structure Documentation	520
23.4.1	<code>struct pwt_config_t</code>	520
23.5	Enumeration Type Documentation	520
23.5.1	<code>pwt_clock_source_t</code>	520
23.5.2	<code>pwt_clock_prescale_t</code>	521
23.5.3	<code>pwt_input_select_t</code>	521
23.5.4	<code>_pwt_interrupt_enable</code>	521
23.5.5	<code>_pwt_status_flags</code>	521
23.6	Function Documentation	521
23.6.1	<code>PWT_Init</code>	521
23.6.2	<code>PWT_Deinit</code>	522
23.6.3	<code>PWT_GetDefaultConfig</code>	522
23.6.4	<code>PWT_EnableInterrupts</code>	522
23.6.5	<code>PWT_DisableInterrupts</code>	522

Section No.	Title	Page No.
23.6.6	PWT_GetEnabledInterrupts	523
23.6.7	PWT_GetStatusFlags	523
23.6.8	PWT_ClearStatusFlags	523
23.6.9	PWT_StartTimer	524
23.6.10	PWT_StopTimer	525
23.6.11	PWT_GetCurrentTimerCount	525
23.6.12	PWT_ReadPositivePulseWidth	525
23.6.13	PWT_ReadNegativePulseWidth	525
23.6.14	PWT_Reset	526

Chapter 24 RCM: Reset Control Module Driver

24.1	Overview	527
24.2	Data Structure Documentation	528
24.2.1	struct rcm_version_id_t	528
24.2.2	struct rcm_reset_pin_filter_config_t	529
24.3	Macro Definition Documentation	529
24.3.1	FSL_RCM_DRIVER_VERSION	529
24.4	Enumeration Type Documentation	529
24.4.1	rcm_reset_source_t	529
24.4.2	rcm_run_wait_filter_mode_t	530
24.4.3	rcm_reset_delay_t	530
24.4.4	rcm_interrupt_enable_t	530
24.5	Function Documentation	530
24.5.1	RCM_GetVersionId	530
24.5.2	RCM_GetPreviousResetSources	531
24.5.3	RCM_GetStickyResetSources	531
24.5.4	RCM_ClearStickyResetSources	532
24.5.5	RCM_ConfigureResetPinFilter	532
24.5.6	RCM_SetSystemResetInterruptConfig	533

Chapter 25 SIM: System Integration Module Driver

25.1	Overview	534
25.2	Data Structure Documentation	534
25.2.1	struct sim_uid_t	534
25.3	Enumeration Type Documentation	535
25.3.1	_sim_flash_mode	535
25.4	Function Documentation	535

Section No.	Title	Page No.
25.4.1	SIM_GetUniqueId	535
25.4.2	SIM_SetFlashMode	535

Chapter 26 SMC: System Mode Controller Driver

26.1	Overview	536
26.2	Typical use case	536
26.2.1	Enter wait or stop modes	536
26.3	Data Structure Documentation	538
26.3.1	struct smc_version_id_t	538
26.3.2	struct smc_param_t	539
26.4	Enumeration Type Documentation	539
26.4.1	smc_power_mode_protection_t	539
26.4.2	smc_power_state_t	539
26.4.3	smc_run_mode_t	539
26.4.4	smc_stop_mode_t	540
26.4.5	smc_partial_stop_option_t	540
26.4.6	anonymous enum	540
26.5	Function Documentation	540
26.5.1	SMC_GetVersionId	540
26.5.2	SMC_GetParam	540
26.5.3	SMC_SetPowerModeProtection	541
26.5.4	SMC_GetPowerModeState	541
26.5.5	SMC_PreEnterStopModes	542
26.5.6	SMC_PostExitStopModes	542
26.5.7	SMC_PreEnterWaitModes	542
26.5.8	SMC_PostExitWaitModes	542
26.5.9	SMC_SetPowerModeRun	542
26.5.10	SMC_SetPowerModeWait	542
26.5.11	SMC_SetPowerModeStop	543
26.5.12	SMC_SetPowerModeVlpr	543
26.5.13	SMC_SetPowerModeVlpw	543
26.5.14	SMC_SetPowerModeVlps	544

Chapter 27 TRGMUX: Trigger Mux Driver

27.1	Overview	545
27.2	Typical use case	545
27.3	Macro Definition Documentation	545
27.3.1	FSL_TRGMUX_DRIVER_VERSION	545

Section No.	Title	Page No.
27.4 Enumeration Type Documentation	545
27.4.1 anonymous enum	546
27.4.2 trgmux_trigger_input_t	546
27.5 Function Documentation	546
27.5.1 TRGMUX_LockRegister	546
27.5.2 TRGMUX_SetTriggerSource	546
Chapter 28 WDOG32: 32-bit Watchdog Timer		
28.1 Overview	548
28.2 Typical use case	548
28.3 Data Structure Documentation	550
28.3.1 struct wdog32_work_mode_t	550
28.3.2 struct wdog32_config_t	550
28.4 Macro Definition Documentation	550
28.4.1 FSL_WDOG32_DRIVER_VERSION	550
28.5 Enumeration Type Documentation	550
28.5.1 wdog32_clock_source_t	551
28.5.2 wdog32_clock_prescaler_t	551
28.5.3 wdog32_test_mode_t	551
28.5.4 _wdog32_interrupt_enable_t	551
28.5.5 _wdog32_status_flags_t	551
28.6 Function Documentation	552
28.6.1 WDOG32_GetDefaultConfig	552
28.6.2 WDOG32_Init	552
28.6.3 WDOG32_Deinit	553
28.6.4 WDOG32_Unlock	553
28.6.5 WDOG32_Enable	553
28.6.6 WDOG32_Disable	553
28.6.7 WDOG32_EnableInterrupts	555
28.6.8 WDOG32_DisableInterrupts	555
28.6.9 WDOG32_GetStatusFlags	555
28.6.10 WDOG32_ClearStatusFlags	556
28.6.11 WDOG32_SetTimeoutValue	556
28.6.12 WDOG32_SetWindowValue	557
28.6.13 WDOG32_Refresh	557
28.6.14 WDOG32_GetCounterValue	557

Section No.	Title	Page No.
Chapter 29 Debug Console		
29.1 Overview	558
29.2 Function groups	558
29.2.1 Initialization	558
29.2.2 Advanced Feature	559
29.2.3 SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_UART	563
29.3 Typical use case	564
29.4 Macro Definition Documentation	566
29.4.1 DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN	566
29.4.2 DEBUGCONSOLE_REDIRECT_TO_SDK	566
29.4.3 DEBUGCONSOLE_DISABLE	566
29.4.4 SDK_DEBUGCONSOLE	566
29.4.5 PRINTF	566
29.5 Function Documentation	566
29.5.1 DbgConsole_Init	566
29.5.2 DbgConsole_Deinit	567
29.5.3 DbgConsole_EnterLowpower	567
29.5.4 DbgConsole_ExitLowpower	568
29.5.5 DbgConsole_Printf	568
29.5.6 DbgConsole_Vprintf	568
29.5.7 DbgConsole_Putchar	568
29.5.8 DbgConsole_Scanf	569
29.5.9 DbgConsole_Getchar	569
29.5.10 DbgConsole_BlockingPrintf	570
29.5.11 DbgConsole_BlockingVprintf	570
29.5.12 DbgConsole_Flush	570
29.5.13 StrFormatPrintf	571
29.5.14 StrFormatScanf	571
29.6 Semihosting	572
29.6.1 Guide Semihosting for IAR	572
29.6.2 Guide Semihosting for Keil µVision	572
29.6.3 Guide Semihosting for MCUXpresso IDE	573
29.6.4 Guide Semihosting for ARMGCC	573
Chapter 30 Notification Framework		
30.1 Overview	576
30.2 Notifier Overview	576

Section No.	Title	Page No.
30.3	Data Structure Documentation	578
30.3.1	struct notifier_notification_block_t	578
30.3.2	struct notifier_callback_config_t	579
30.3.3	struct notifier_handle_t	579
30.4	Typedef Documentation	580
30.4.1	notifier_user_config_t	580
30.4.2	notifier_user_function_t	580
30.4.3	notifier_callback_t	581
30.5	Enumeration Type Documentation	581
30.5.1	_notifier_status	581
30.5.2	notifier_policy_t	582
30.5.3	notifier_notification_type_t	582
30.5.4	notifier_callback_type_t	582
30.6	Function Documentation	582
30.6.1	NOTIFIER_CreateHandle	583
30.6.2	NOTIFIER_SwitchConfig	584
30.6.3	NOTIFIER_GetErrorCallbackIndex	585

Chapter 31 Shell

31.1	Overview	586
31.2	Function groups	586
31.2.1	Initialization	586
31.2.2	Advanced Feature	586
31.2.3	Shell Operation	586
31.3	Data Structure Documentation	588
31.3.1	struct shell_command_t	588
31.4	Macro Definition Documentation	589
31.4.1	SHELL_NON_BLOCKING_MODE	589
31.4.2	SHELL_AUTO_COMPLETE	589
31.4.3	SHELL_BUFFER_SIZE	589
31.4.4	SHELL_MAX_ARGS	589
31.4.5	SHELL_HISTORY_COUNT	589
31.4.6	SHELL_HANDLE_SIZE	589
31.4.7	SHELL_USE_COMMON_TASK	589
31.4.8	SHELL_TASK_PRIORITY	589
31.4.9	SHELL_TASK_STACK_SIZE	589
31.4.10	SHELL_HANDLE_DEFINE	590
31.4.11	SHELL_COMMAND_DEFINE	590
31.4.12	SHELL_COMMAND	591

Section No.	Title	Page No.
31.5	Typedef Documentation	591
31.5.1	cmd_function_t	591
31.6	Enumeration Type Documentation	591
31.6.1	shell_status_t	591
31.7	Function Documentation	591
31.7.1	SHELL_Init	591
31.7.2	SHELL_RegisterCommand	592
31.7.3	SHELL_UnregisterCommand	593
31.7.4	SHELL_Write	593
31.7.5	SHELL_Printf	593
31.7.6	SHELL_WriteSynchronization	594
31.7.7	SHELL_PrintfSynchronization	594
31.7.8	SHELL_ChangePrompt	595
31.7.9	SHELL_PrintPrompt	595
31.7.10	SHELL_Task	595
31.7.11	SHELL_checkRunningInIsr	596

Chapter 32 Serial Manager

32.1	Overview	597
32.2	Data Structure Documentation	600
32.2.1	struct serial_manager_config_t	600
32.2.2	struct serial_manager_callback_message_t	600
32.3	Macro Definition Documentation	601
32.3.1	SERIAL_MANAGER_WRITE_TIME_DELAY_DEFAULT_VALUE	601
32.3.2	SERIAL_MANAGER_READ_TIME_DELAY_DEFAULT_VALUE	601
32.3.3	SERIAL_MANAGER_USE_COMMON_TASK	601
32.3.4	SERIAL_MANAGER_HANDLE_SIZE	601
32.3.5	SERIAL_MANAGER_HANDLE_DEFINE	601
32.3.6	SERIAL_MANAGER_WRITE_HANDLE_DEFINE	601
32.3.7	SERIAL_MANAGER_READ_HANDLE_DEFINE	602
32.3.8	SERIAL_MANAGER_TASK_PRIORITY	602
32.3.9	SERIAL_MANAGER_TASK_STACK_SIZE	602
32.4	Enumeration Type Documentation	602
32.4.1	serial_port_type_t	602
32.4.2	serial_manager_type_t	603
32.4.3	serial_manager_status_t	603
32.5	Function Documentation	603
32.5.1	SerialManager_Init	603
32.5.2	SerialManager_Deinit	604

Section No.	Title	Page No.
32.5.3	SerialManager_OpenWriteHandle	605
32.5.4	SerialManager_CloseWriteHandle	606
32.5.5	SerialManager_OpenReadHandle	606
32.5.6	SerialManager_CloseReadHandle	607
32.5.7	SerialManager_WriteBlocking	608
32.5.8	SerialManager_ReadBlocking	608
32.5.9	SerialManager_EnterLowpower	609
32.5.10	SerialManager_ExitLowpower	609
32.5.11	SerialManager_SetLowpowerCriticalCb	610
32.6	Serial Port Uart	611
32.6.1	Overview	611
32.6.2	Enumeration Type Documentation	611

Chapter 33 Tsi_v5_driver

33.1	Overview	612
33.2	Data Structure Documentation	621
33.2.1	struct tsi_calibration_data_t	621
33.2.2	struct tsi_common_config_t	621
33.2.3	struct tsi_selfCap_config_t	622
33.2.4	struct tsi_mutualCap_config_t	623
33.3	Enumeration Type Documentation	624
33.3.1	tsi_main_clock_selection_t	624
33.3.2	tsi_sensing_mode_selection_t	624
33.3.3	tsi_dvolt_option_t	625
33.3.4	tsi_sensitivity_xdn_option_t	625
33.3.5	tsi_shield_t	625
33.3.6	tsi_sensitivity_ctrim_option_t	626
33.3.7	tsi_current_multiple_input_t	626
33.3.8	tsi_current_multiple_charge_t	626
33.3.9	tsi_mutual_pre_current_t	627
33.3.10	tsi_mutual_pre_resistor_t	627
33.3.11	tsi_mutual_sense_resistor_t	627
33.3.12	tsi_mutual_tx_channel_t	628
33.3.13	tsi_mutual_rx_channel_t	628
33.3.14	tsi_mutual_sense_boost_current_t	629
33.3.15	tsi_mutual_tx_drive_mode_t	630
33.3.16	tsi_mutual_pmos_current_left_t	630
33.3.17	tsi_mutual_pmos_current_right_t	631
33.3.18	tsi_mutual_nmos_current_t	631
33.3.19	tsi_sinc_cutoff_div_t	632
33.3.20	tsi_sinc_filter_order_t	632

Section No.	Title	Page No.
33.3.21	<code>tsi_sinc_decimation_value_t</code>	632
33.3.22	<code>tsi_ssc_charge_num_t</code>	634
33.3.23	<code>tsi_ssc_nocharge_num_t</code>	634
33.3.24	<code>tsi_ssc_prbs_outsel_t</code>	635
33.3.25	<code>tsi_status_flags_t</code>	636
33.3.26	<code>tsi_interrupt_enable_t</code>	636
33.3.27	<code>tsi_ssc_mode_t</code>	636
33.3.28	<code>tsi_ssc_prescaler_t</code>	636
33.4	Function Documentation	637
33.4.1	<code>TSIGetInstance</code>	637
33.4.2	<code>TSI_InitSelfCapMode</code>	637
33.4.3	<code>TSI_InitMutualCapMode</code>	637
33.4.4	<code>TSI_Deinit</code>	638
33.4.5	<code>TSI_GetSelfCapModeDefaultConfig</code>	638
33.4.6	<code>TSI_GetMutualCapModeDefaultConfig</code>	639
33.4.7	<code>TSI_SelfCapCalibrate</code>	639
33.4.8	<code>TSI_EnableInterrupts</code>	640
33.4.9	<code>TSI_DisableInterrupts</code>	640
33.4.10	<code>TSI_GetStatusFlags</code>	640
33.4.11	<code>TSI_ClearStatusFlags</code>	641
33.4.12	<code>TSI_GetScanTriggerMode</code>	641
33.4.13	<code>TSI_IsScanInProgress</code>	641
33.4.14	<code>TSI_EnableModule</code>	642
33.4.15	<code>TSI_EnableLowPower</code>	643
33.4.16	<code>TSI_EnableHardwareTriggerScan</code>	643
33.4.17	<code>TSI_StartSoftwareTrigger</code>	644
33.4.18	<code>TSI_SetSelfCapMeasuredChannel</code>	644
33.4.19	<code>TSI_GetSelfCapMeasuredChannel</code>	644
33.4.20	<code>TSI_EnableDmaTransfer</code>	645
33.4.21	<code>TSI_EnableEndOfScanDmaTransferOnly</code>	645
33.4.22	<code>TSI_GetCounter</code>	646
33.4.23	<code>TSI_SetLowThreshold</code>	647
33.4.24	<code>TSI_SetHighThreshold</code>	647
33.4.25	<code>TSI_SetMainClock</code>	647
33.4.26	<code>TSI_SetSensingMode</code>	648
33.4.27	<code>TSI_GetSensingMode</code>	648
33.4.28	<code>TSI_SetDvolt</code>	648
33.4.29	<code>TSI_EnableNoiseCancellation</code>	649
33.4.30	<code>TSI_SetMutualCapTxChannel</code>	649
33.4.31	<code>TSI_GetTxMutualCapMeasuredChannel</code>	649
33.4.32	<code>TSI_SetMutualCapRxChannel</code>	650
33.4.33	<code>TSI_GetRxMutualCapMeasuredChannel</code>	650
33.4.34	<code>TSI_SetSscMode</code>	650
33.4.35	<code>TSI_SetSscPrescaler</code>	651

Section No.	Title	Page No.
33.4.36	TSI_SetUsedTxChannel	651
33.4.37	TSI_ClearUsedTxChannel	651

Chapter 1

Introduction

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

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

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

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

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



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

MCUXpresso SDK Folder Structure

Chapter 2

Trademarks

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

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

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EM-BRACE, GREENCHIP, HITAG, I2C BUS,ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4M-OBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TD-MI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2021 NXP B.V.

Chapter 3

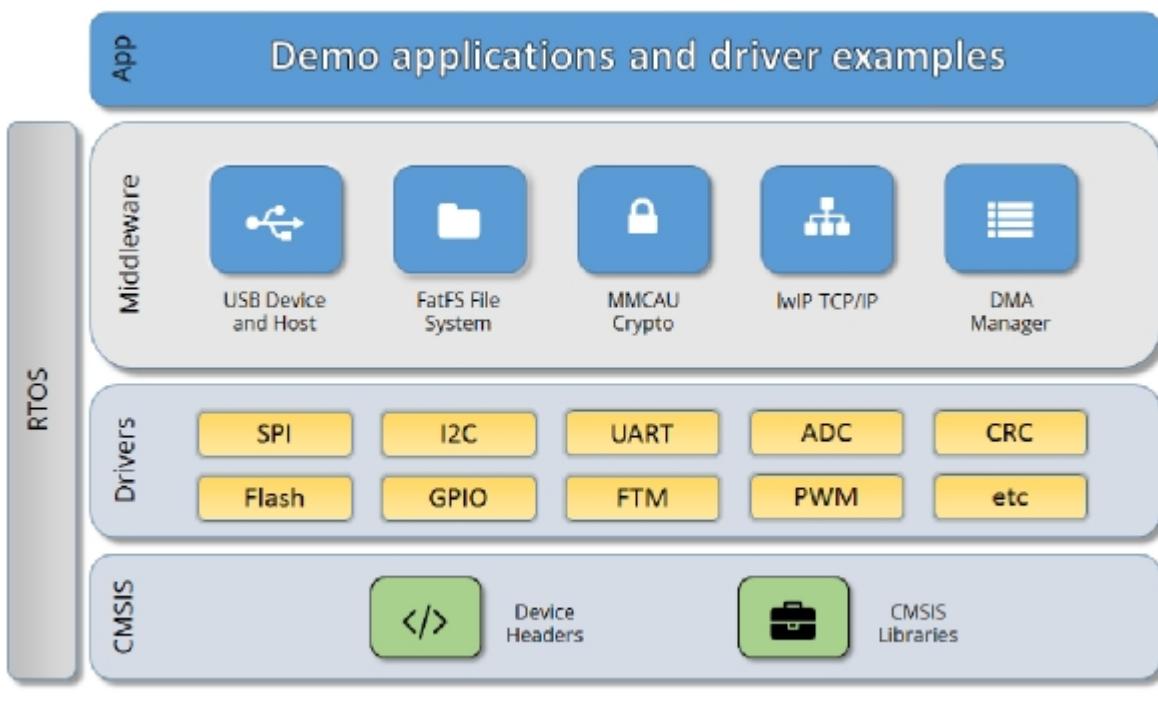
Architectural Overview

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

Overview

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

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



MCU header files

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

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

CMSIS Support

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

MCUXpresso SDK Peripheral Drivers

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

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

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

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

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

Interrupt handling for transactional APIs

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

```
PUBWEAK SPI0_IRQHandler  
PUBWEAK SPI0_DriverIRQHandler  
SPI0_IRQHandler
```

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

The first level of the weak implementation are the functions defined in the vector table. In the devices/<DEVICE_NAME>/<TOOLCHAIN>/startup_<DEVICE_NAME>.s/.S file, the implementation of the first layer weak function calls the second layer of weak function. The implementation of the second layer weak function (ex. SPI0_DriverIRQHandler) jumps to itself (B). The MCUXpresso SDK drivers with transactional APIs provide the reimplementation of the second layer function inside of the peripheral driver. If the MCUXpresso SDK drivers with transactional APIs are linked into the image, the SPI0_DriverIRQHandler is replaced with the function implemented in the MCUXpresso SDK SPI driver.

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

The limitation of the double weak mechanism is that it cannot be used for peripherals that share the same vector entry. For this use case, redefine the first layer weak function to enable the desired peripheral interrupt functionality. For example, if the MCU's UART0 and UART1 share the same vector entry, redefine the UART0_UART1_IRQHandler according to the use case requirements.

Feature Header Files

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

Application

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

Chapter 4

Clock Driver

4.1 Overview

The MCUXpresso SDK provides APIs for MCUXpresso SDK devices' clock operation.

The clock driver supports:

- Clock generator (PLL, FLL, and so on) configuration
- Clock mux and divider configuration
- Getting clock frequency

Modules

- System Clock Generator (SCG)

Files

- file `fsl_clock.h`

Data Structures

- struct `scg_sys_clk_config_t`
SCG system clock configuration. [More...](#)
- struct `scg_sosc_config_t`
SCG system OSC configuration. [More...](#)
- struct `scg_sirc_config_t`
SCG slow IRC clock configuration. [More...](#)
- struct `scg_firc_trim_config_t`
SCG fast IRC clock trim configuration. [More...](#)
- struct `scg_firc_config_t`
SCG fast IRC clock configuration. [More...](#)
- struct `scg_lppll_trim_config_t`
SCG LPFLL clock trim configuration. [More...](#)
- struct `scg_lppll_config_t`
SCG low power FLL configuration. [More...](#)

Macros

- `#define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0`
Configure whether driver controls clock.
- `#define DMAMUX_CLOCKS`
Clock ip name array for DMAMUX.
- `#define PORT_CLOCKS`
Clock ip name array for PORT.
- `#define LPI2C_CLOCKS`

- `#define FLEXIO_CLOCKS`
Clock ip name array for FLEXIO.
- `#define TSI_CLOCKS`
Clock ip name array for TSI.
- `#define EDMA_CLOCKS`
Clock ip name array for EDMA.
- `#define LPUART_CLOCKS`
Clock ip name array for LPUART.
- `#define LPTMR_CLOCKS`
Clock ip name array for LPTMR.
- `#define ADC12_CLOCKS`
Clock ip name array for ADC12.
- `#define LPSPI_CLOCKS`
Clock ip name array for LPSPI.
- `#define LPIT_CLOCKS`
Clock ip name array for LPIT.
- `#define CRC_CLOCKS`
Clock ip name array for CRC.
- `#define CMP_CLOCKS`
Clock ip name array for CMP.
- `#define FLASH_CLOCKS`
Clock ip name array for FLASH.
- `#define EWM_CLOCKS`
Clock ip name array for EWM.
- `#define FTM_CLOCKS`
Clock ip name array for FLEXTMR.
- `#define PWT_CLOCKS`
Clock ip name array for PWT.
- `#define FLEXIOTRIG_CLOCKS`
Clock ip name array for FLEXIO_TRIGGER0/1 Async clock.
- `#define LPO_CLK_FREQ 128000U`
LPO clock frequency.
- `#define CLOCK_GetOsc0ErClkFreq CLOCK_GetErClkFreq`
For compatible with other MCG platforms.

Enumerations

- enum `clock_name_t` {

kCLOCK_CoreSysClk,
 kCLOCK_BusClk,
 kCLOCK_FlashClk,
 kCLOCK_ScgSysOscClk,
 kCLOCK_ScgSircClk,
 kCLOCK_ScgFircClk,
 kCLOCK_ScgLpFllClk,
 kCLOCK_ScgSysOscAsyncDiv2Clk,
 kCLOCK_ScgSircAsyncDiv2Clk,
 kCLOCK_ScgFircAsyncDiv2Clk,
 kCLOCK_ScgLpFllAsyncDiv2Clk,
 kCLOCK_LpoClk,
 kCLOCK_ErClk }

Clock name used to get clock frequency.

- enum `clock_ip_src_t` {

kCLOCK_IpSrcNoneOrExt = 0U,
 kCLOCK_IpSrcSysOscAsync = 1U,
 kCLOCK_IpSrcSircAsync = 2U,
 kCLOCK_IpSrcFircAsync = 3U,
 kCLOCK_IpSrcLpFllAsync = 5U }

Clock source for peripherals that support various clock selections.

- enum `clock_ip_name_t`

Peripheral clock name definition used for clock gate, clock source and clock divider setting.

- enum {

kStatus_SCG_Busy = MAKE_STATUS(kStatusGroup_SCG, 1),
 kStatus_SCG_InvalidSrc = MAKE_STATUS(kStatusGroup_SCG, 2) }

SCG status return codes.

- enum `scg_sys_clk_t` {

kSCG_SysClkSlow,
 kSCG_SysClkCore }

SCG system clock type.

- enum `scg_sys_clk_src_t` {

kSCG_SysClkSrcSysOsc = 1U,
 kSCG_SysClkSrcSirc = 2U,
 kSCG_SysClkSrcFirc = 3U,
 kSCG_SysClkSrcLpFll = 5U }

SCG system clock source.

- enum `scg_sys_clk_div_t` {

```

kSCG_SysClkDivBy1 = 0U,
kSCG_SysClkDivBy2 = 1U,
kSCG_SysClkDivBy3 = 2U,
kSCG_SysClkDivBy4 = 3U,
kSCG_SysClkDivBy5 = 4U,
kSCG_SysClkDivBy6 = 5U,
kSCG_SysClkDivBy7 = 6U,
kSCG_SysClkDivBy8 = 7U,
kSCG_SysClkDivBy9 = 8U,
kSCG_SysClkDivBy10 = 9U,
kSCG_SysClkDivBy11 = 10U,
kSCG_SysClkDivBy12 = 11U,
kSCG_SysClkDivBy13 = 12U,
kSCG_SysClkDivBy14 = 13U,
kSCG_SysClkDivBy15 = 14U,
kSCG_SysClkDivBy16 = 15U }

```

SCG system clock divider value.

- enum `clock_clkout_src_t` {

```

kClockClkoutSelScgSlow = 0U,
kClockClkoutSelSysOsc = 1U,
kClockClkoutSelSirc = 2U,
kClockClkoutSelFirc = 3U,
kClockClkoutSelLpFll = 5U }

```

SCG clock out configuration (CLKOUTSEL).

- enum `scg_async_clk_t` { `kSCG_AsyncDiv2Clk` }

SCG asynchronous clock type.

- enum `scg_async_clk_div_t` {

```

kSCG_AsyncClkDisable = 0U,
kSCG_AsyncClkDivBy1 = 1U,
kSCG_AsyncClkDivBy2 = 2U,
kSCG_AsyncClkDivBy4 = 3U,
kSCG_AsyncClkDivBy8 = 4U,
kSCG_AsyncClkDivBy16 = 5U,
kSCG_AsyncClkDivBy32 = 6U,
kSCG_AsyncClkDivBy64 = 7U }

```

SCG asynchronous clock divider value.

- enum `scg_sosc_monitor_mode_t` {

```

kSCG_SysOscMonitorDisable = 0U,
kSCG_SysOscMonitorInt = SCG_SOSCCSR_SOSCCM_MASK,
kSCG_SysOscMonitorReset }

```

SCG system OSC monitor mode.

- enum `scg_sosc_mode_t` {

```

kSCG_SysOscModeExt = 0U,
kSCG_SysOscModeOscLowPower = SCG_SOSCCFG_EREFS_MASK,
kSCG_SysOscModeOscHighGain = SCG_SOSCCFG_EREFS_MASK | SCG_SOSCCFG_HGO_-
MASK }

```

- *OSC work mode.*
- enum {

 kSCG_SysOscEnable = SCG_SOSCCSR_SOSCEN_MASK,
 kSCG_SysOscEnableInStop = SCG_SOSCCSR_SOSCSTEN_MASK,
 kSCG_SysOscEnableInLowPower = SCG_SOSCCSR_SOSCLPEN_MASK,
 kSCG_SysOscEnableErClk = SCG_SOSCCSR_SOSCERCLKEN_MASK }
- *OSC enable mode.*
- enum `scg_sirc_range_t` {

 kSCG_SircRangeLow,
 kSCG_SircRangeHigh }
- *SCG slow IRC clock frequency range.*
- enum {

 kSCG_SircEnable = SCG_SIRCCSR_SIRCEN_MASK,
 kSCG_SircEnableInStop = SCG_SIRCCSR_SIRCSTEN_MASK,
 kSCG_SircEnableInLowPower = SCG_SIRCCSR_SIRCLPEN_MASK }
- *SIRC enable mode.*
- enum `scg_firc_trim_mode_t` {

 kSCG_FircTrimNonUpdate = SCG_FIRCCSR_FIRCTREN_MASK,
 kSCG_FircTrimUpdate = SCG_FIRCCSR_FIRCTREN_MASK | SCG_FIRCCSR_FIRCTRUP_-
 MASK }
- *SCG fast IRC trim mode.*
- enum `scg_firc_trim_div_t` {

 kSCG_FircTrimDivBy1,
 kSCG_FircTrimDivBy128,
 kSCG_FircTrimDivBy256,
 kSCG_FircTrimDivBy512,
 kSCG_FircTrimDivBy1024,
 kSCG_FircTrimDivBy2048 }
- *SCG fast IRC trim predivided value for system OSC.*
- enum `scg_firc_trim_src_t` { kSCG_FircTrimSrcSysOsc = 2U }
- *SCG fast IRC trim source.*
- enum `scg_firc_range_t` { kSCG_FircRange48M }
- *SCG fast IRC clock frequency range.*
- enum {

 kSCG_FircEnable = SCG_FIRCCSR_FIRCEN_MASK,
 kSCG_FircEnableInStop = SCG_FIRCCSR_FIRCSTEN_MASK,
 kSCG_FircEnableInLowPower = SCG_FIRCCSR_FIRCLPEN_MASK,
 kSCG_FircDisableRegulator = SCG_FIRCCSR_FIRCREGOFF_MASK }
- *FIRC enable mode.*
- enum { kSCG_LpFllEnable = SCG_LPFLCSR_LPFLLEN_MASK }
- *LPFLL enable mode.*
- enum `scg_lpfl_range_t` {

 kSCG_LpFllRange48M = 0U,
 kSCG_LpFllRange72M = 1U }
- *SCG LPFLL clock frequency range.*
- enum `scg_lpfl_trim_mode_t` {

```

kSCG_LpFllTrimNonUpdate = SCG_LPFLCSR_LPFLLTREN_MASK,
kSCG_LpFllTrimUpdate = SCG_LPFLCSR_LPFLLTREN_MASK | SCG_LPFLCSR_LPFL-
TRUP_MASK }

    SCG LPFLL trim mode.
• enum scg_lpfl TrimSrc {
    kSCG_LpFllTrimSrcSirc = 0U,
    kSCG_LpFllTrimSrcFirc = 1U,
    kSCG_LpFllTrimSrcSysOsc = 2U,
    kSCG_LpFllTrimSrcRtcOsc = 3U }

    SCG LPFLL trim source.
• enum scg_lpfl LockMode {
    kSCG_LpFllLock1Lsb = 0U,
    kSCG_LpFllLock2Lsb = 1U }

    SCG LPFLL lock mode.

```

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_SetIpSrc** (clock_ip_name_t name, clock_ip_src_t src)
Set the clock source for specific IP module.
- static void **CLOCK_SetIpSrcDiv** (clock_ip_name_t name, clock_ip_src_t src, uint16_t divValue, uint8_t fracValue)
Set the clock source and divider for specific IP module.
- 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_GetBusClkFreq** (void)
Get the bus clock frequency.
- uint32_t **CLOCK_GetFlashClkFreq** (void)
Get the flash clock frequency.
- uint32_t **CLOCK_GetErClkFreq** (void)
Get the external reference clock frequency (ERCLK).
- uint32_t **CLOCK_GetIpFreq** (clock_ip_name_t name)
Gets the clock frequency for a specific IP module.

Variables

- volatile uint32_t **g_xtal0Freq**
External XTAL0 (OSC0/SYROS) clock frequency.

Driver version

- #define **FSL_CLOCK_DRIVER_VERSION** (MAKE_VERSION(2, 0, 0))
CLOCK driver version 2.0.0.

MCU System Clock.

- `uint32_t CLOCK_GetSysClkFreq (scg_sys_clk_t type)`
Gets the SCG system clock frequency.
- `static void CLOCK_SetVlprModeSysClkConfig (const scg_sys_clk_config_t *config)`
Sets the system clock configuration for VLPR mode.
- `static void CLOCK_SetRunModeSysClkConfig (const scg_sys_clk_config_t *config)`
Sets the system clock configuration for RUN mode.
- `static void CLOCK_GetCurSysClkConfig (scg_sys_clk_config_t *config)`
Gets the system clock configuration in the current power mode.
- `static void CLOCK_SetClkOutSel (clock_clkout_src_t setting)`
Sets the clock out selection.

SCG System OSC Clock.

- `status_t CLOCK_InitSysOsc (const scg_sosc_config_t *config)`
Initializes the SCG system OSC.
- `status_t CLOCK_DeinitSysOsc (void)`
De-initializes the SCG system OSC.
- `static void CLOCK_SetSysOscAsyncClkDiv (scg_async_clk_t asyncClk, scg_async_clk_div_t divider)`
Set the asynchronous clock divider.
- `uint32_t CLOCK_GetSysOscFreq (void)`
Gets the SCG system OSC clock frequency (SYSOSC).
- `uint32_t CLOCK_GetSysOscAsyncFreq (scg_async_clk_t type)`
Gets the SCG asynchronous clock frequency from the system OSC.
- `static bool CLOCK_IsSysOscErr (void)`
Checks whether the system OSC clock error occurs.
- `static void CLOCK_ClearSysOscErr (void)`
Clears the system OSC clock error.
- `static void CLOCK_SetSysOscMonitorMode (scg_sosc_monitor_mode_t mode)`
Sets the system OSC monitor mode.
- `static bool CLOCK_IsSysOscValid (void)`
Checks whether the system OSC clock is valid.

SCG Slow IRC Clock.

- `status_t CLOCK_InitSirc (const scg_sirc_config_t *config)`
Initializes the SCG slow IRC clock.
- `status_t CLOCK_DeinitSirc (void)`
De-initializes the SCG slow IRC.
- `static void CLOCK_SetSircAsyncClkDiv (scg_async_clk_t asyncClk, scg_async_clk_div_t divider)`
Set the asynchronous clock divider.
- `uint32_t CLOCK_GetSircFreq (void)`
Gets the SCG SIRC clock frequency.
- `uint32_t CLOCK_GetSircAsyncFreq (scg_async_clk_t type)`
Gets the SCG asynchronous clock frequency from the SIRC.
- `static bool CLOCK_IsSircValid (void)`
Checks whether the SIRC clock is valid.

SCG Fast IRC Clock.

- `status_t CLOCK_InitFirc (const scg_firc_config_t *config)`
Initializes the SCG fast IRC clock.
- `status_t CLOCK_DeinitFirc (void)`
De-initializes the SCG fast IRC.
- `static void CLOCK_SetFircAsyncClkDiv (scg_async_clk_t asyncClk, scg_async_clk_div_t divider)`
Set the asynchronous clock divider.
- `uint32_t CLOCK_GetFircFreq (void)`
Gets the SCG FIRC clock frequency.
- `uint32_t CLOCK_GetFircAsyncFreq (scg_async_clk_t type)`
Gets the SCG asynchronous clock frequency from the FIRC.
- `static bool CLOCK_IsFircValid (void)`
Checks whether the FIRC clock is valid.

SCG Low Power FLL Clock.

- `status_t CLOCK_InitLpFll (const scg_lppll_config_t *config)`
Initializes the SCG LPFLL clock.
- `status_t CLOCK_DeinitLpFll (void)`
De-initializes the SCG LPFLL.
- `static void CLOCK_SetLpFllAsyncClkDiv (scg_async_clk_t asyncClk, scg_async_clk_div_t divider)`
Set the asynchronous clock divider.
- `uint32_t CLOCK_GetLpFllFreq (void)`
Gets the SCG LPFLL clock frequency.
- `uint32_t CLOCK_GetLpFllAsyncFreq (scg_async_clk_t type)`
Gets the SCG asynchronous clock frequency from the LPFLL.
- `static bool CLOCK_IsLpFllValid (void)`
Checks whether the LPFLL clock is valid.

External clock frequency

- `static void CLOCK_SetXtal0Freq (uint32_t freq)`
Sets the XTAL0 frequency based on board settings.

4.2 Data Structure Documentation

4.2.1 struct scg_sys_clk_config_t

Data Fields

- `uint32_t divSlow: 4`
Slow clock divider, see `scg_sys_clk_div_t`.
- `uint32_t __pad0__: 4`
Reserved.
- `uint32_t __pad1__: 4`
Reserved.
- `uint32_t __pad2__: 4`
Reserved.

- `uint32_t divCore`: 4
Core clock divider, see `scg_sys_clk_div_t`.
- `uint32_t __pad3__`: 4
Reserved.
- `uint32_t src`: 4
System clock source, see `scg_sys_clk_src_t`.
- `uint32_t __pad4__`: 4
reserved.

Field Documentation

- (1) `uint32_t scg_sys_clk_config_t::divSlow`
- (2) `uint32_t scg_sys_clk_config_t::__pad0__`
- (3) `uint32_t scg_sys_clk_config_t::__pad1__`
- (4) `uint32_t scg_sys_clk_config_t::__pad2__`
- (5) `uint32_t scg_sys_clk_config_t::divCore`
- (6) `uint32_t scg_sys_clk_config_t::__pad3__`
- (7) `uint32_t scg_sys_clk_config_t::src`
- (8) `uint32_t scg_sys_clk_config_t::__pad4__`

4.2.2 struct scg_sosc_config_t

Data Fields

- `uint32_t freq`
System OSC frequency.
- `scg_sosc_monitor_mode_t monitorMode`
Clock monitor mode selected.
- `uint8_t enableMode`
Enable mode, OR'ed value of _scg_sosc_enable_mode.
- `scg_async_clk_div_t div2`
SOSCDIV2 value.
- `scg_sosc_mode_t workMode`
OSC work mode.

Field Documentation

- (1) `uint32_t scg_sosc_config_t::freq`
- (2) `scg_sosc_monitor_mode_t scg_sosc_config_t::monitorMode`
- (3) `uint8_t scg_sosc_config_t::enableMode`

- (4) scg_async_clk_div_t scg_sosc_config_t::div2
- (5) scg_sosc_mode_t scg_sosc_config_t::workMode

4.2.3 struct scg_sirc_config_t

Data Fields

- uint32_t enableMode
Enable mode, OR'ed value of _scg_sirc_enable_mode.
- scg_async_clk_div_t div2
SIRCDIV2 value.
- scg_sirc_range_t range
Slow IRC frequency range.

Field Documentation

- (1) uint32_t scg_sirc_config_t::enableMode
- (2) scg_async_clk_div_t scg_sirc_config_t::div2
- (3) scg_sirc_range_t scg_sirc_config_t::range

4.2.4 struct scg_firc_trim_config_t

Data Fields

- scg_firc_trim_mode_t trimMode
FIRC trim mode.
- scg_firc_trim_src_t trimSrc
Trim source.
- scg_firc_trim_div_t trimDiv
Trim predivided value for the system OSC.
- uint8_t trimCoar
Trim coarse value; Irrelevant if trimMode is kSCG_FircTrimUpdate.
- uint8_t trimFine
Trim fine value; Irrelevant if trimMode is kSCG_FircTrimUpdate.

Field Documentation

- (1) scg_firc_trim_mode_t scg_firc_trim_config_t::trimMode
- (2) scg_firc_trim_src_t scg_firc_trim_config_t::trimSrc
- (3) scg_firc_trim_div_t scg_firc_trim_config_t::trimDiv
- (4) uint8_t scg_firc_trim_config_t::trimCoar
- (5) uint8_t scg_firc_trim_config_t::trimFine

4.2.5 struct scg_firc_config_t

Data Fields

- `uint32_t enableMode`
Enable mode, OR'ed value of _scg_firc_enable_mode.
- `scg_async_clk_div_t div2`
FIRCDIV2 value.
- `scg_firc_range_t range`
Fast IRC frequency range.
- `const scg_firc_trim_config_t * trimConfig`
Pointer to the FIRC trim configuration; set NULL to disable trim.

Field Documentation

- (1) `uint32_t scg_firc_config_t::enableMode`
- (2) `scg_async_clk_div_t scg_firc_config_t::div2`
- (3) `scg_firc_range_t scg_firc_config_t::range`
- (4) `const scg_firc_trim_config_t* scg_firc_config_t::trimConfig`

4.2.6 struct scg_lppll_trim_config_t

Data Fields

- `scg_lppll_trim_mode_t trimMode`
Trim mode.
- `scg_lppll_lock_mode_t lockMode`
Lock mode; Irrelevant if the trimMode is kSCG_LpFllTrimNonUpdate.
- `scg_lppll_trim_src_t trimSrc`
Trim source.
- `uint8_t trimDiv`
Trim predivideds value, which can be 0 ~ 31.
- `uint8_t trimValue`
Trim value; Irrelevant if trimMode is the kSCG_LpFllTrimUpdate.

Field Documentation

- (1) `scg_lppll_trim_mode_t scg_lppll_trim_config_t::trimMode`
- (2) `scg_lppll_lock_mode_t scg_lppll_trim_config_t::lockMode`
- (3) `scg_lppll_trim_src_t scg_lppll_trim_config_t::trimSrc`
- (4) `uint8_t scg_lppll_trim_config_t::trimDiv`

[Trim source frequency / (trimDiv + 1)] must be 2 MHz or 32768 Hz.

(5) `uint8_t scg_lpfill_trim_config_t::trimValue`

4.2.7 struct scg_lpfill_config_t

Data Fields

- `uint8_t enableMode`
Enable mode, OR'ed value of _scg_lpfill_enable_mode.
- `scg_async_clk_div_t div2`
LPFLLDIV2 value.
- `scg_lpfill_range_t range`
LPFLL frequency range.
- `const scg_lpfill_trim_config_t * trimConfig`
Trim configuration; set NULL to disable trim.

Field Documentation

(1) `scg_async_clk_div_t scg_lpfill_config_t::div2`

(2) `scg_lpfill_range_t scg_lpfill_config_t::range`

(3) `const scg_lpfill_trim_config_t* scg_lpfill_config_t::trimConfig`

4.3 Macro Definition Documentation

4.3.1 #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.

4.3.2 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

4.3.3 #define DMAMUX_CLOCKS

Value:

```
{
    \_kCLOCK_Dmamux0 \
}
```

4.3.4 #define PORT_CLOCKS

Value:

```
{\n    kCLOCK_PortA, kCLOCK_PortB, kCLOCK_PortC, kCLOCK_PortD, kCLOCK_PortE \n}
```

4.3.5 #define LPI2C_CLOCKS

Value:

```
{\n    kCLOCK_Lpi2c0 \n}
```

4.3.6 #define FLEXIO_CLOCKS

Value:

```
{\n    kCLOCK_Flexio0 \n}
```

4.3.7 #define TSI_CLOCKS

Value:

```
{\n    kCLOCK_Tsi0, kCLOCK_Tsil \n}
```

4.3.8 #define EDMA_CLOCKS

Value:

```
{\n    kCLOCK_Dma0 \n}
```

4.3.9 #define LPUART_CLOCKS

Value:

```
{\n    kCLOCK_Lpuart0, kCLOCK_Lpuart1, kCLOCK_Lpuart2 \n}
```

4.3.10 #define LPTMR_CLOCKS

Value:

```
{\n    kCLOCK_Lptmr0 \n}
```

4.3.11 #define ADC12_CLOCKS

Value:

```
{\n    kCLOCK_Adc0 \n}
```

4.3.12 #define LPSPI_CLOCKS

Value:

```
{\n    kCLOCK_Lpspi0 \n}
```

4.3.13 #define LPIT_CLOCKS

Value:

```
{\n    kCLOCK_Lpit0 \n}
```

4.3.14 #define CRC_CLOCKS

Value:

```
{           \
    kCLOCK_Crc0 \
}
```

4.3.15 #define CMP_CLOCKS

Value:

```
{           \
    kCLOCK_Cmp0 \
}
```

4.3.16 #define FLASH_CLOCKS

Value:

```
{           \
    kCLOCK_Flash0 \
}
```

4.3.17 #define EWM_CLOCKS

Value:

```
{           \
    kCLOCK_Ewm0 \
}
```

4.3.18 #define FTM_CLOCKS

Value:

```
{           \
    kCLOCK_Ftm0, kCLOCK_Ftm1, kCLOCK_Ftm2 \
}
```

4.3.19 #define PWT_CLOCKS

Value:

```
{
    \_kCLOCK_Pwt0 \
}
```

4.3.20 #define FLEXIOTRIG_CLOCKS

Value:

```
{
    \_kCLOCK_FlexioTrig0, _kCLOCK_FlexioTrig1 \
}
```

4.3.21 #define CLOCK_GetOsc0ErClkFreq CLOCK_GetErClkFreq

4.4 Enumeration Type Documentation

4.4.1 enum clock_name_t

Enumerator

kCLOCK_CoreSysClk Core/system clock.
kCLOCK_BusClk Bus clock.
kCLOCK_FlashClk Flash clock.
kCLOCK_ScgSysOscClk SCG system OSC clock. (SYSOSC)
kCLOCK_ScgSircClk SCG SIRC clock.
kCLOCK_ScgFircClk SCG FIRClk clock.
kCLOCK_ScgLpFllClk SCG low power FLL clock. (LPFLL)
kCLOCK_ScgSysOscAsyncDiv2Clk SOSCDIV2_CLK.
kCLOCK_ScgSircAsyncDiv2Clk SIRCDIV2_CLK.
kCLOCK_ScgFircAsyncDiv2Clk FIRCDIV2_CLK.
kCLOCK_ScgLpFllAsyncDiv2Clk LPFLLDIV2_CLK.
kCLOCK_LpoClk LPO clock.
kCLOCK_ErClk ERCLK. The external reference clock from SCG.

4.4.2 enum clock_ip_src_t

Enumerator

kCLOCK_IpSrcNoneOrExt Clock is off or external clock is used.

kCLOCK_IpSrcSysOscAsync System Oscillator async clock.

kCLOCK_IpSrcSircAsync Slow IRC async clock.

kCLOCK_IpSrcFircAsync Fast IRC async clock.

kCLOCK_IpSrcLpFllAsync LPFLL async clock.

4.4.3 enum clock_ip_name_t

It is defined as the corresponding register address.

4.4.4 anonymous enum

Enumerator

kStatus_SCG_Busy Clock is busy.

kStatus_SCG_InvalidSrc Invalid source.

4.4.5 enum scg_sys_clk_t

Enumerator

kSCG_SysClkSlow System slow clock.

kSCG_SysClkCore Core clock.

4.4.6 enum scg_sys_clk_src_t

Enumerator

kSCG_SysClkSrcSysOsc System OSC.

kSCG_SysClkSrcSirc Slow IRC.

kSCG_SysClkSrcFirc Fast IRC.

kSCG_SysClkSrcLpFll Low power FLL.

4.4.7 enum scg_sys_clk_div_t

Enumerator

kSCG_SysClkDivBy1 Divided by 1.

kSCG_SysClkDivBy2 Divided by 2.

kSCG_SysClkDivBy3 Divided by 3.

kSCG_SysClkDivBy4 Divided by 4.
kSCG_SysClkDivBy5 Divided by 5.
kSCG_SysClkDivBy6 Divided by 6.
kSCG_SysClkDivBy7 Divided by 7.
kSCG_SysClkDivBy8 Divided by 8.
kSCG_SysClkDivBy9 Divided by 9.
kSCG_SysClkDivBy10 Divided by 10.
kSCG_SysClkDivBy11 Divided by 11.
kSCG_SysClkDivBy12 Divided by 12.
kSCG_SysClkDivBy13 Divided by 13.
kSCG_SysClkDivBy14 Divided by 14.
kSCG_SysClkDivBy15 Divided by 15.
kSCG_SysClkDivBy16 Divided by 16.

4.4.8 enum clock_clkout_src_t

Enumerator

kClockClkoutSelScgSlow SCG slow clock.
kClockClkoutSelSysOsc System OSC.
kClockClkoutSelSirc Slow IRC.
kClockClkoutSelFirc Fast IRC.
kClockClkoutSelLpFll Low power FLL.

4.4.9 enum scg_async_clk_t

Enumerator

kSCG_AsyncDiv2Clk The async clock by DIV2, e.g. SOSCDIV2_CLK, SIRCDIV2_CLK.

4.4.10 enum scg_async_clk_div_t

Enumerator

kSCG_AsyncClkDisable Clock output is disabled.
kSCG_AsyncClkDivBy1 Divided by 1.
kSCG_AsyncClkDivBy2 Divided by 2.
kSCG_AsyncClkDivBy4 Divided by 4.
kSCG_AsyncClkDivBy8 Divided by 8.
kSCG_AsyncClkDivBy16 Divided by 16.
kSCG_AsyncClkDivBy32 Divided by 32.
kSCG_AsyncClkDivBy64 Divided by 64.

4.4.11 enum scg_sosc_monitor_mode_t

Enumerator

kSCG_SysOscMonitorDisable Monitor disabled.

kSCG_SysOscMonitorInt Interrupt when the system OSC error is detected.

kSCG_SysOscMonitorReset Reset when the system OSC error is detected.

4.4.12 enum scg_sosc_mode_t

Enumerator

kSCG_SysOscModeExt Use external clock.

kSCG_SysOscModeOscLowPower Oscillator low power.

kSCG_SysOscModeOscHighGain Oscillator high gain.

4.4.13 anonymous enum

Enumerator

kSCG_SysOscEnable Enable OSC clock.

kSCG_SysOscEnableInStop Enable OSC in stop mode.

kSCG_SysOscEnableInLowPower Enable OSC in low power mode.

kSCG_SysOscEnableErClk Enable OSCERCLK.

4.4.14 enum scg_sirc_range_t

Enumerator

kSCG_SircRangeLow Slow IRC low range clock (2 MHz, 4 MHz for i.MX 7 ULP).

kSCG_SircRangeHigh Slow IRC high range clock (8 MHz, 16 MHz for i.MX 7 ULP).

4.4.15 anonymous enum

Enumerator

kSCG_SircEnable Enable SIRC clock.

kSCG_SircEnableInStop Enable SIRC in stop mode.

kSCG_SircEnableInLowPower Enable SIRC in low power mode.

4.4.16 enum scg_firc_trim_mode_t

Enumerator

kSCG_FircTrimNonUpdate FIRC trim enable but not enable trim value update. In this mode, the trim value is fixed to the initialized value which is defined by trimCoar and trimFine in configure structure [scg_firc_trim_config_t](#).

kSCG_FircTrimUpdate FIRC trim enable and trim value update enable. In this mode, the trim value is auto update.

4.4.17 enum scg_firc_trim_div_t

Enumerator

kSCG_FircTrimDivBy1 Divided by 1.

kSCG_FircTrimDivBy128 Divided by 128.

kSCG_FircTrimDivBy256 Divided by 256.

kSCG_FircTrimDivBy512 Divided by 512.

kSCG_FircTrimDivBy1024 Divided by 1024.

kSCG_FircTrimDivBy2048 Divided by 2048.

4.4.18 enum scg_firc_trim_src_t

Enumerator

kSCG_FircTrimSrcSysOsc System OSC.

4.4.19 enum scg_firc_range_t

Enumerator

kSCG_FircRange48M Fast IRC is trimmed to 48 MHz.

4.4.20 anonymous enum

Enumerator

kSCG_FircEnable Enable FIRC clock.

kSCG_FircEnableInStop Enable FIRC in stop mode.

kSCG_FircEnableInLowPower Enable FIRC in low power mode.

kSCG_FircDisableRegulator Disable regulator.

4.4.21 anonymous enum

Enumerator

kSCG_LpFllEnable Enable LPFLL clock.

4.4.22 enum scg_lpfull_range_t

Enumerator

kSCG_LpFllRange48M LPFLL is trimmed to 48MHz.

kSCG_LpFllRange72M LPFLL is trimmed to 72MHz.

4.4.23 enum scg_lpfull_trim_mode_t

Enumerator

kSCG_LpFllTrimNonUpdate LPFLL trim is enabled but the trim value update is not enabled. In this mode, the trim value is fixed to the initialized value, which is defined by the Member variable `trimValue` in the structure `scg_lpfull_trim_config_t`.

kSCG_LpFllTrimUpdate FIRC trim is enabled and trim value update is enabled. In this mode, the trim value is automatically updated.

4.4.24 enum scg_lpfull_trim_src_t

Enumerator

kSCG_LpFllTrimSrcSirc SIRC.

kSCG_LpFllTrimSrcFirc FIRC.

kSCG_LpFllTrimSrcSysOsc System OSC.

kSCG_LpFllTrimSrcRtcOsc RTC OSC (32.768 kHz).

4.4.25 enum scg_lpfull_lock_mode_t

Enumerator

kSCG_LpFllLock1Lsb Lock with 1 LSB.

kSCG_LpFllLock2Lsb Lock with 2 LSB.

4.5 Function Documentation

4.5.1 **static void CLOCK_EnableClock (clock_ip_name_t *name*) [inline], [static]**

Parameters

<i>name</i>	Which clock to enable, see clock_ip_name_t .
-------------	--

4.5.2 static void CLOCK_DisableClock (*clock_ip_name_t name*) [inline], [static]

Parameters

<i>name</i>	Which clock to disable, see clock_ip_name_t .
-------------	---

4.5.3 static void CLOCK_SetIpSrc (*clock_ip_name_t name*, *clock_ip_src_t src*) [inline], [static]

Set the clock source for specific IP, not all modules need to set the clock source, should only use this function for the modules need source setting.

Parameters

<i>name</i>	Which peripheral to check, see clock_ip_name_t .
<i>src</i>	Clock source to set.

4.5.4 static void CLOCK_SetIpSrcDiv (*clock_ip_name_t name*, *clock_ip_src_t src*, *uint16_t divValue*, *uint8_t fracValue*) [inline], [static]

Set the clock source and divider for specific IP, not all modules need to set the clock source and divider, should only use this function for the modules need source and divider setting.

Divider output clock = Divider input clock x [(fracValue+1)/(divValue+1)].

Parameters

<i>name</i>	Which peripheral to check, see clock_ip_name_t .
<i>src</i>	Clock source to set.

<i>divValue</i>	The divider value.
<i>fracValue</i>	The fraction multiply value.

4.5.5 `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`.

Parameters

<i>clockName</i>	Clock names defined in <code>clock_name_t</code>
------------------	--

Returns

Clock frequency value in hertz

4.5.6 `uint32_t CLOCK_GetCoreSysClkFreq (void)`

Returns

Clock frequency in Hz.

4.5.7 `uint32_t CLOCK_GetBusClkFreq (void)`

Returns

Clock frequency in Hz.

4.5.8 `uint32_t CLOCK_GetFlashClkFreq (void)`

Returns

Clock frequency in Hz.

4.5.9 `uint32_t CLOCK_GetErClkFreq (void)`

Returns

Clock frequency in Hz.

4.5.10 `uint32_t CLOCK_GetIpFreq(clock_ip_name_t name)`

This function gets the IP module clock frequency based on PCC registers. It is only used for the IP modules which could select clock source by PCC[PCS].

Parameters

<i>name</i>	Which peripheral to get, see clock_ip_name_t .
-------------	--

Returns

Clock frequency value in hertz

4.5.11 **uint32_t CLOCK_GetSysClkFreq (scg_sys_clk_t *type*)**

This function gets the SCG system clock frequency. These clocks are used for core, platform, external, and bus clock domains.

Parameters

<i>type</i>	Which type of clock to get, core clock or slow clock.
-------------	---

Returns

Clock frequency.

4.5.12 **static void CLOCK_SetVlprModeSysClkConfig (const scg_sys_clk_config_t * *config*) [inline], [static]**

This function sets the system clock configuration for VLPR mode.

Parameters

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

4.5.13 **static void CLOCK_SetRunModeSysClkConfig (const scg_sys_clk_config_t * *config*) [inline], [static]**

This function sets the system clock configuration for RUN mode.

Parameters

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

4.5.14 static void CLOCK_GetCurSysClkConfig (scg_sys_clk_config_t * *config*) [inline], [static]

This function gets the system configuration in the current power mode.

Parameters

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

4.5.15 static void CLOCK_SetClkOutSel (clock_clkout_src_t *setting*) [inline], [static]

This function sets the clock out selection (CLKOUTSEL).

Parameters

<i>setting</i>	The selection to set.
----------------	-----------------------

Returns

The current clock out selection.

4.5.16 status_t CLOCK_InitSysOsc (const scg_sosc_config_t * *config*)

This function enables the SCG system OSC clock according to the configuration.

Parameters

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

Return values

<i>kStatus_Success</i>	System OSC is initialized.
------------------------	----------------------------

<i>kStatus_SCG_Busy</i>	System OSC has been enabled and is used by the system clock.
<i>kStatus_ReadOnly</i>	System OSC control register is locked.

Note

This function can't detect whether the system OSC has been enabled and used by an IP.

4.5.17 **status_t CLOCK_DeinitSysOsc (void)**

This function disables the SCG system OSC clock.

Return values

<i>kStatus_Success</i>	System OSC is deinitialized.
<i>kStatus_SCG_Busy</i>	System OSC is used by the system clock.
<i>kStatus_ReadOnly</i>	System OSC control register is locked.

Note

This function can't detect whether the system OSC is used by an IP.

4.5.18 **static void CLOCK_SetSysOscAsyncClkDiv (scg_async_clk_t *asyncClk*, scg_async_clk_div_t *divider*) [inline], [static]**

Parameters

<i>asyncClk</i>	Which asynchronous clock to configure.
<i>divider</i>	The divider value to set.

Note

There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

4.5.19 **uint32_t CLOCK_GetSysOscFreq (void)**

Returns

Clock frequency; If the clock is invalid, returns 0.

4.5.20 `uint32_t CLOCK_GetSysOscAsyncFreq (scg_async_clk_t type)`

Parameters

<i>type</i>	The asynchronous clock type.
-------------	------------------------------

Returns

Clock frequency; If the clock is invalid, returns 0.

4.5.21 static bool CLOCK_IsSysOscErr(void) [inline], [static]

Returns

True if the error occurs, false if not.

4.5.22 static void CLOCK_SetSysOscMonitorMode(scg_sosc_monitor_mode_t mode) [inline], [static]

This function sets the system OSC monitor mode. The mode can be disabled, it can generate an interrupt when the error is disabled, or reset when the error is detected.

Parameters

<i>mode</i>	Monitor mode to set.
-------------	----------------------

4.5.23 static bool CLOCK_IsSysOscValid(void) [inline], [static]

Returns

True if clock is valid, false if not.

4.5.24 status_t CLOCK_InitSirc(const scg_sirc_config_t * config)

This function enables the SCG slow IRC clock according to the configuration.

Parameters

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

Return values

<i>kStatus_Success</i>	SIRC is initialized.
<i>kStatus_SCG_Busy</i>	SIRC has been enabled and is used by system clock.
<i>kStatus_ReadOnly</i>	SIRC control register is locked.

Note

This function can't detect whether the system OSC has been enabled and used by an IP.

4.5.25 status_t CLOCK_DeinitSirc (void)

This function disables the SCG slow IRC.

Return values

<i>kStatus_Success</i>	SIRC is deinitialized.
<i>kStatus_SCG_Busy</i>	SIRC is used by system clock.
<i>kStatus_ReadOnly</i>	SIRC control register is locked.

Note

This function can't detect whether the SIRC is used by an IP.

4.5.26 static void CLOCK_SetSircAsyncClkDiv (scg_async_clk_t *asyncClk*, scg_async_clk_div_t *divider*) [inline], [static]

Parameters

<i>asyncClk</i>	Which asynchronous clock to configure.
<i>divider</i>	The divider value to set.

Note

There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

4.5.27 uint32_t CLOCK_GetSircFreq (void)

Returns

Clock frequency; If the clock is invalid, returns 0.

4.5.28 uint32_t CLOCK_GetSircAsyncFreq (scg_async_clk_t type)

Parameters

<i>type</i>	The asynchronous clock type.
-------------	------------------------------

Returns

Clock frequency; If the clock is invalid, returns 0.

4.5.29 static bool CLOCK_IsSircValid (void) [inline], [static]

Returns

True if clock is valid, false if not.

4.5.30 status_t CLOCK_InitFirc (const scg_firc_config_t * config)

This function enables the SCG fast IRC clock according to the configuration.

Parameters

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

Return values

<i>kStatus_Success</i>	FIRC is initialized.
<i>kStatus_SCG_Busy</i>	FIRC has been enabled and is used by the system clock.

<i>kStatus_ReadOnly</i>	FIRC control register is locked.
-------------------------	----------------------------------

Note

This function can't detect whether the FIRC has been enabled and used by an IP.

4.5.31 status_t CLOCK_DeinitFirc (void)

This function disables the SCG fast IRC.

Return values

<i>kStatus_Success</i>	FIRC is deinitialized.
<i>kStatus_SCG_Busy</i>	FIRC is used by the system clock.
<i>kStatus_ReadOnly</i>	FIRC control register is locked.

Note

This function can't detect whether the FIRC is used by an IP.

4.5.32 static void CLOCK_SetFircAsyncClkDiv (scg_async_clk_t *asyncClk*, scg_async_clk_div_t *divider*) [inline], [static]

Parameters

<i>asyncClk</i>	Which asynchronous clock to configure.
<i>divider</i>	The divider value to set.

Note

There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

4.5.33 uint32_t CLOCK_GetFircFreq (void)

Returns

Clock frequency; If the clock is invalid, returns 0.

4.5.34 uint32_t CLOCK_GetFircAsyncFreq (scg_async_clk_t *type*)

Parameters

<i>type</i>	The asynchronous clock type.
-------------	------------------------------

Returns

Clock frequency; If the clock is invalid, returns 0.

4.5.35 static bool CLOCK_IsFircValid (void) [inline], [static]

Returns

True if clock is valid, false if not.

4.5.36 status_t CLOCK_InitLpFll (const scg_lppll_config_t * config)

This function enables the SCG LPFLL clock according to the configuration.

Parameters

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

Return values

<i>kStatus_Success</i>	LPFLL is initialized.
<i>kStatus_SCG_Busy</i>	LPFLL has been enabled and is used by the system clock.
<i>kStatus_ReadOnly</i>	LPFLL control register is locked.

Note

This function can't detect whether the LPFLL has been enabled and used by an IP.

4.5.37 status_t CLOCK_DeinitLpFll (void)

This function disables the SCG LPFLL.

Return values

<i>kStatus_Success</i>	LPFLL is deinitialized.
<i>kStatus_SCG_Busy</i>	LPFLL is used by the system clock.
<i>kStatus_ReadOnly</i>	LPFLL control register is locked.

Note

This function can't detect whether the LPFLL is used by an IP.

4.5.38 **static void CLOCK_SetLpFIISyncClkDiv (scg_async_clk_t *asyncClk*, scg_async_clk_div_t *divider*) [inline], [static]**

Parameters

<i>asyncClk</i>	Which asynchronous clock to configure.
<i>divider</i>	The divider value to set.

Note

There might be glitch when changing the asynchronous divider, so make sure the asynchronous clock is not used while changing divider.

4.5.39 **uint32_t CLOCK_GetLpFIIFreq (void)**

Returns

Clock frequency in Hz; If the clock is invalid, returns 0.

4.5.40 **uint32_t CLOCK_GetLpFIISyncFreq (scg_async_clk_t *type*)**

Parameters

<i>type</i>	The asynchronous clock type.
-------------	------------------------------

Returns

Clock frequency in Hz; If the clock is invalid, returns 0.

4.5.41 static bool CLOCK_IsLpFIIValid(void) [inline], [static]

Returns

True if the clock is valid, false if not.

4.5.42 static void CLOCK_SetXtal0Freq(uint32_t freq) [inline], [static]

Parameters

<i>freq</i>	The XTAL0/EXTAL0 input clock frequency in Hz.
-------------	---

4.6 Variable Documentation

4.6.1 volatile uint32_t g_xtal0Freq

The XTAL0/EXTAL0 (OSC0/SYSOSC) 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_InitSysOsc(...);
* CLOCK_SetXtal0Freq(80000000);
*
```

This is important for the multicore platforms where only one core needs to set up the OSC0/SYSOSC using CLOCK_InitSysOsc. All other cores need to call the CLOCK_SetXtal0Freq to get a valid clock frequency.

4.7 System Clock Generator (SCG)

The MCUXpresso SDK provides a peripheral driver for the System Clock Generator (SCG) module of MCUXpresso SDK devices.

4.7.1 Function description

The SCG module contains the system PLL (SPLL), a slow internal reference clock (SIRC), a fast internal reference clock (FIRC), a low power FLL, and the system oscillator clock (SOSC). They can be configured separately as the source of MCU system clocks. Accordingly, the SCG driver provides these functions:

- MCU system clock configuration.
- SCG SOSC configuration.
- SCG SIRC configuration.
- SCG FIRC configuration.
- SCG SPLL configuration.
- SCG LPFLL configuration.

4.7.1.1 MCU System Clock

MCU system clock configurations include the clock source selection and the clock dividers. The configurations for VLPR, RUN, and HSRUN modes are set separately using the [CLOCK_SetVlprModeSysClkConfig\(\)](#), [CLOCK_SetRunModeSysClkConfig\(\)](#), and the [CLOCK_SetHsrunModeSysClkConfig\(\)](#) functions to configure the MCU system clock.

The current MCU system clock configuration can be obtained with the function [CLOCK_GetCurSysClkConfig\(\)](#). The current MCU system clock frequency can be obtained with the [CLOCK_GetSysClkFreq\(\)](#) function.

4.7.1.2 SCG System OSC Clock

The functions [CLOCK_InitSysOsc\(\)](#)/[CLOCK_DeinitSysOsc\(\)](#) are used for the SOSC clock initialization. The function [CLOCK_InitSysOsc](#) disables the SOSC internally and re-configures it. As a result, ensure that the SOSC is not used while calling these functions.

The SOSC clock can be used directly as the MCU system clock source. The SOSCDIV1_CLK, SOSCDIV2_CLK, and SOSCDIV3_CLK can be used as the peripheral clock source. The clocks frequencies can be obtained by functions [CLOCK_GetSysOscFreq\(\)](#) and [CLOCK_GetSysOscAsyncFreq\(\)](#).

To configure the SOSC monitor mode, use the function [CLOCK_SetSysOscMonitorMode\(\)](#). The clock error status can be received and cleared with the [CLOCK_IsSysOscErr\(\)](#) and [CLOCK_ClearSysOscErr\(\)](#) functions.

4.7.1.3 SCG Slow IRC Clock

The functions [CLOCK_InitSirc\(\)](#)/[CLOCK_DeinitSirc\(\)](#) are used for the SIRC clock initialization. The function [CLOCK_InitSirc](#) disables the SIRC internally and re-configures it. Ensure that the SIRC is not used while calling these functions.

The SIRC clock can be used directly as the MCU system clock source. The SIRCDIV1_CLK, SIRCDIV2_CLK, and SIRCDIV3_CLK can be used as the peripheral clock source. The clocks frequencies can be received with functions [CLOCK_GetSircFreq\(\)](#) and [CLOCK_GetSircAsyncFreq\(\)](#).

4.7.1.4 SCG Fast IRC Clock

The functions [CLOCK_InitFirc\(\)](#)/[CLOCK_DeinitFirc\(\)](#) are used for the FIRC clock initialization. The function [CLOCK_InitFirc](#) disables the FIRC internally and re-configures it. Ensure that the FIRC is not used while calling these functions.

The FIRC clock can be used directly as the MCU system clock source. The FIRCDIV1_CLK, FIRCDIV2_CLK, and FIRCDIV3_CLK can be used as the peripheral clock source. The clocks frequencies could be obtained by functions [CLOCK_GetFircFreq\(\)](#) and [CLOCK_GetFircAsyncFreq\(\)](#).

The FIRC can be trimmed by the external clock. See the Section "Typical use case" to enable the FIRC trim.

4.7.1.5 SCG Low Power FLL Clock

The functions [CLOCK_InitLpFll\(\)](#)/[CLOCK_DeinitLpFll\(\)](#) are used for the LPFLL clock initialization. The function [CLOCK_InitLpFll](#) disables the LPFLL internally and re-configures it. Ensure that the LPFLL is not used while calling these functions.

The LPFLL clock can be used directly as the MCU system clock source. The LPFLLDIV1_CLK, LPFLLDIV2_CLK, and LPFLLDIV3_CLK can be used as the peripheral clock source. The clocks frequencies could be obtained by functions [CLOCK_GetLpFllFreq\(\)](#) and [CLOCK_GetLpFllAsyncFreq\(\)](#).

The LPFLL can be trimmed by the external clock, specific the trimConfig in [scg_lppll_config_t](#) to enable the clock trim.

4.7.1.6 SCG System PLL Clock

The functions [CLOCK_InitSysPll\(\)](#)/[CLOCK_DeinitSysPll\(\)](#) are used for the SPLL clock initialization. The function [CLOCK_InitSysPll](#) disables the SPLL internally and re-configures it. Ensure that the SPLL is not used while calling these functions.

To generate the desired SPLL frequency, PREDIV and MULT value must be set properly while initializing the SPLL. The function [CLOCK_GetSysPllMultDiv\(\)](#) calculates the PREDIV and MULT. Passing in the reference clock frequency and the desired output frequency, the function returns the PREDIV and MULT which generate the frequency closest to the desired frequency.

Because the SPLL is based on the FIRC or SOSC, the FIRC or SOSC must be enabled first before the SPLL initialization. Also, when re-configuring the FIRC or SOSC, be careful with the SPLL.

The SPLL clock can be used directly as the MCU system clock source. The SPLLDIV1_CLK, SPLLDIV2_CLK, and SPLLDIV3_CLK can be used as the peripheral clock source. The clocks frequencies can be obtained with functions `CLOCK_GetSysPllFreq()` and `CLOCK_GetSysPllAsyncFreq()`.

To configure the SPLL monitor mode, use the function `CLOCK_SetSysPllMonitorMode()`. The clock error status can be received and cleared by the `CLOCK_IsSysPllErr()` and `CLOCK_ClearSysPllErr()`.

4.7.1.7 SCG clock valid check

The functions such as the `CLOCK_IsFircValid()` are used to check whether a specific clock is valid or not. See "Typical use case" for details.

The clocks are valid after the initialization functions such as the `CLOCK_InitFirc()`. As a result, it is not necessary to call the `CLOCK_IsFircValid()` after the `CLOCK_InitFirc()`.

4.7.2 Typical use case

4.7.2.1 FIRC clock trim

During the FIRC initialization, applications can choose whether to enable trim or not.

1. Trim is not enabled. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/scg
2. Trim is enabled. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/scg

4.7.2.2 SPLL initialization

The following code shows how to set up the SCG SPLL. The SPLL uses the SOSC as a reference clock. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/scg

4.7.2.3 System clock configuration

While changing the system clock configuration, the actual system clock does not change until the target clock source is valid. Ensure that the clock source is valid before using it. The functions such as `CLOCK_IsSircValid()` are used for this purpose.

The SCG has a dedicated system clock configuration registers for VLPR, RUN, and HSRUN modes. During the power mode change, the system clock configuration may change too. In this case, check whether the clock source is valid during the power mode change.

In the following example, the SIRC is used as the system clock source in VLPR mode, the FIRC is used as a system clock source in RUN mode, and the SPLL is used as a system clock source in HSRUN mode.

The example work flow:

1. SIRC, FIRC, and SPLL are all enabled in RUN mode.
2. MCU enters VLPR mode. In VLPR mode, FIRC, and SPLL are disabled automatically.
3. MCU enters RUN mode. Wait for the FIRC to become valid.
4. MCU enters HSRUN mode. In step 3, the SPLL is already enabled, but may not be valid. Wait for it to become valid when entering HSRUN mode. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/scg

Chapter 5

ACMP: Analog Comparator Driver

5.1 Overview

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

The ACMP driver is created to help the user operate the ACMP module better. This driver can be considered as a basic comparator with advanced features. The APIs for basic comparator can make the C-MP work as a general comparator, which compares the two input channel's voltage and creates the output of the comparator result immediately. The APIs for advanced feature can be used as the plug-in function based on the basic comparator, and can provide more ways to process the comparator's output.

5.2 Typical use case

5.2.1 Normal Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/acmp

5.2.2 Interrupt Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/acmp

5.2.3 Round robin Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/acmp

Data Structures

- struct `acmp_config_t`
Configuration for ACMP. [More...](#)
- struct `acmp_channel_config_t`
Configuration for channel. [More...](#)
- struct `acmp_filter_config_t`
Configuration for filter. [More...](#)
- struct `acmp_dac_config_t`
Configuration for DAC. [More...](#)
- struct `acmp_round_robin_config_t`
Configuration for round robin mode. [More...](#)

Macros

- #define **CMP_C0_CFx_MASK** (CMP_C0_CFR_MASK | CMP_C0_CFF_MASK)
The mask of status flags cleared by writing 1.

Enumerations

- enum **_acmp_interrupt_enable** {

kACMP_OutputRisingInterruptEnable = (1U << 0U),

kACMP_OutputFallingInterruptEnable = (1U << 1U),

kACMP_RoundRobinInterruptEnable = (1U << 2U) }

Interrupt enable/disable mask.
- enum **_acmp_status_flags** {

kACMP_OutputRisingEventFlag = CMP_C0_CFR_MASK,

kACMP_OutputFallingEventFlag = CMP_C0_CFF_MASK,

kACMP_OutputAssertEventFlag = CMP_C0_COUT_MASK } }

Status flag mask.
- enum **acmp_offset_mode_t** {

kACMP_OffsetLevel0 = 0U,

kACMP_OffsetLevel1 = 1U } }

Comparator hard block offset control.
- enum **acmp_hysteresis_mode_t** {

kACMP_HysteresisLevel0 = 0U,

kACMP_HysteresisLevel1 = 1U,

kACMP_HysteresisLevel2 = 2U,

kACMP_HysteresisLevel3 = 3U } }

Comparator hard block hysteresis control.
- enum **acmp_reference_voltage_source_t** {

kACMP_VrefSourceVin1 = 0U,

kACMP_VrefSourceVin2 = 1U } }

CMP Voltage Reference source.
- enum **acmp_port_input_t** {

kACMP_PortInputFromDAC = 0U,

kACMP_PortInputFromMux = 1U } }

Port input source.
- enum **acmp_fixed_port_t** {

kACMP_FixedPlusPort = 0U,

kACMP_FixedMinusPort = 1U } }

Fixed mux port.

Driver version

- #define **FSL_ACMP_DRIVER_VERSION** (MAKE_VERSION(2U, 0U, 6U))
ACMP driver version 2.0.6.

Initialization and deinitialization

- void **ACMP_Init** (CMP_Type *base, const **acmp_config_t** *config)

- `void ACMP_Deinit (CMP_Type *base)`
Deinitializes the ACMP.
- `void ACMP_GetDefaultConfig (acmp_config_t *config)`
Gets the default configuration for ACMP.

Basic Operations

- `void ACMP_Enable (CMP_Type *base, bool enable)`
Enables or disables the ACMP.
- `void ACMP_SetChannelConfig (CMP_Type *base, const acmp_channel_config_t *config)`
Sets the channel configuration.

Advanced Operations

- `void ACMP_EnableDMA (CMP_Type *base, bool enable)`
Enables or disables DMA.
- `void ACMP_EnableWindowMode (CMP_Type *base, bool enable)`
Enables or disables window mode.
- `void ACMP_SetFilterConfig (CMP_Type *base, const acmp_filter_config_t *config)`
Configures the filter.
- `void ACMP_SetDACConfig (CMP_Type *base, const acmp_dac_config_t *config)`
Configures the internal DAC.
- `void ACMP_SetRoundRobinConfig (CMP_Type *base, const acmp_round_robin_config_t *config)`
Configures the round robin mode.
- `void ACMP_SetRoundRobinPreState (CMP_Type *base, uint32_t mask)`
Defines the pre-set state of channels in round robin mode.
- `static uint32_t ACMP_GetRoundRobinStatusFlags (CMP_Type *base)`
Gets the channel input changed flags in round robin mode.
- `void ACMP_ClearRoundRobinStatusFlags (CMP_Type *base, uint32_t mask)`
Clears the channel input changed flags in round robin mode.
- `static uint32_t ACMP_GetRoundRobinResult (CMP_Type *base)`
Gets the round robin result.

Interrupts

- `void ACMP_EnableInterrupts (CMP_Type *base, uint32_t mask)`
Enables interrupts.
- `void ACMP_DisableInterrupts (CMP_Type *base, uint32_t mask)`
Disables interrupts.

Status

- `uint32_t ACMP_GetStatusFlags (CMP_Type *base)`
Gets status flags.
- `void ACMP_ClearStatusFlags (CMP_Type *base, uint32_t mask)`
Clears status flags.

5.3 Data Structure Documentation

5.3.1 struct acmp_config_t

Data Fields

- `acmp_offset_mode_t offsetMode`
Offset mode.
- `acmp_hysteresis_mode_t hysteresisMode`
Hysteresis mode.
- `bool enableHighSpeed`
Enable High Speed (HS) comparison mode.
- `bool enableInvertOutput`
Enable inverted comparator output.
- `bool useUnfilteredOutput`
Set compare output(COUT) to equal COUTA(true) or COUT(false).
- `bool enablePinOut`
The comparator output is available on the associated pin.

Field Documentation

- (1) `acmp_offset_mode_t acmp_config_t::offsetMode`
- (2) `acmp_hysteresis_mode_t acmp_config_t::hysteresisMode`
- (3) `bool acmp_config_t::enableHighSpeed`
- (4) `bool acmp_config_t::enableInvertOutput`
- (5) `bool acmp_config_t::useUnfilteredOutput`
- (6) `bool acmp_config_t::enablePinOut`

5.3.2 struct acmp_channel_config_t

The comparator's port can be input from channel mux or DAC. If port input is from channel mux, detailed channel number for the mux should be configured.

Data Fields

- `acmp_port_input_t positivePortInput`
Input source of the comparator's positive port.
- `uint32_t plusMuxInput`
Plus mux input channel(0~7).
- `acmp_port_input_t negativePortInput`
Input source of the comparator's negative port.
- `uint32_t minusMuxInput`
Minus mux input channel(0~7).

Field Documentation

- (1) acmp_port_input_t acmp_channel_config_t::positivePortInput
- (2) uint32_t acmp_channel_config_t::plusMuxInput
- (3) acmp_port_input_t acmp_channel_config_t::negativePortInput
- (4) uint32_t acmp_channel_config_t::minusMuxInput

5.3.3 struct acmp_filter_config_t

Data Fields

- bool enableSample
Using external SAMPLE as sampling clock input, or using divided bus clock.
- uint32_t filterCount
Filter Sample Count.
- uint32_t filterPeriod
Filter Sample Period.

Field Documentation

- (1) bool acmp_filter_config_t::enableSample
- (2) uint32_t acmp_filter_config_t::filterCount

Available range is 1-7, 0 would cause the filter disabled.

- (3) uint32_t acmp_filter_config_t::filterPeriod

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

5.3.4 struct acmp_dac_config_t

Data Fields

- acmp_reference_voltage_source_t referenceVoltageSource
Supply voltage reference source.
- uint32_t DACValue
Value for DAC Output Voltage.

Field Documentation

- (1) acmp_reference_voltage_source_t acmp_dac_config_t::referenceVoltageSource
- (2) uint32_t acmp_dac_config_t::DACValue

Available range is 0-255.

5.3.5 struct acmp_round_robin_config_t

Data Fields

- `acmp_fixed_port_t fixedPort`
Fixed mux port.
- `uint32_t fixedChannelNumber`
Indicates which channel is fixed in the fixed mux port.
- `uint32_t checkerChannelMask`
Mask of checker channel index.
- `uint32_t sampleClockCount`
Specifies how many round-robin clock cycles(0~3) later the sample takes place.
- `uint32_t delayModulus`
Comparator and DAC initialization delay modulus.

Field Documentation

(1) `acmp_fixed_port_t acmp_round_robin_config_t::fixedPort`

(2) `uint32_t acmp_round_robin_config_t::fixedChannelNumber`

(3) `uint32_t acmp_round_robin_config_t::checkerChannelMask`

Available range is channel0:0x01 to channel7:0x80 for round-robin checker.

(4) `uint32_t acmp_round_robin_config_t::sampleClockCount`

(5) `uint32_t acmp_round_robin_config_t::delayModulus`

5.4 Macro Definition Documentation

5.4.1 `#define FSL_ACMP_DRIVER_VERSION (MAKE_VERSION(2U, 0U, 6U))`

5.4.2 `#define CMP_C0_CFx_MASK (CMP_C0_CFR_MASK | CMP_C0_CFF_MASK)`

5.5 Enumeration Type Documentation

5.5.1 enum _acmp_interrupt_enable

Enumerator

kACMP_OutputRisingInterruptEnable Enable the interrupt when comparator outputs rising.

kACMP_OutputFallingInterruptEnable Enable the interrupt when comparator outputs falling.

kACMP_RoundRobinInterruptEnable Enable the Round-Robin interrupt.

5.5.2 enum _acmp_status_flags

Enumerator

kACMP_OutputRisingEventFlag Rising-edge on compare output has occurred.

kACMP_OutputFallingEventFlag Falling-edge on compare output has occurred.

kACMP_OutputAssertEventFlag Return the current value of the analog comparator output.

5.5.3 enum acmp_offset_mode_t

If OFFSET level is 1, then there is no hysteresis in the case of positive port input crossing negative port input in the positive direction (or negative port input crossing positive port input in the negative direction). Hysteresis still exists for positive port input crossing negative port input in the falling direction. If OFFSET level is 0, then the hysteresis selected by acmp_hysteresis_mode_t is valid for both directions.

Enumerator

kACMP_OffsetLevel0 The comparator hard block output has level 0 offset internally.

kACMP_OffsetLevel1 The comparator hard block output has level 1 offset internally.

5.5.4 enum acmp_hysteresis_mode_t

See chip data sheet to get the actual hysteresis value with each level.

Enumerator

kACMP_HysteresisLevel0 Offset is level 0 and Hysteresis is level 0.

kACMP_HysteresisLevel1 Offset is level 0 and Hysteresis is level 1.

kACMP_HysteresisLevel2 Offset is level 0 and Hysteresis is level 2.

kACMP_HysteresisLevel3 Offset is level 0 and Hysteresis is level 3.

5.5.5 enum acmp_reference_voltage_source_t

Enumerator

kACMP_VrefSourceVin1 Vin1 is selected as resistor ladder network supply reference Vin.

kACMP_VrefSourceVin2 Vin2 is selected as resistor ladder network supply reference Vin.

5.5.6 enum acmp_port_input_t

Enumerator

kACMP_PortInputFromDAC Port input from the 8-bit DAC output.

kACMP_PortInputFromMux Port input from the analog 8-1 mux.

5.5.7 enum acmp_fixed_port_t

Enumerator

kACMP_FixedPlusPort Only the inputs to the Minus port are swept in each round.

kACMP_FixedMinusPort Only the inputs to the Plus port are swept in each round.

5.6 Function Documentation

5.6.1 void ACMP_Init (CMP_Type * *base*, const acmp_config_t * *config*)

The default configuration can be got by calling [ACMP_GetDefaultConfig\(\)](#).

Parameters

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

5.6.2 void ACMP_Deinit (CMP_Type * *base*)

Parameters

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

5.6.3 void ACMP_GetDefaultConfig (acmp_config_t * *config*)

This function initializes the user configuration structure to default value. The default value are:

Example:

```
config->enableHighSpeed = false;
config->enableInvertOutput = false;
config->useUnfilteredOutput = false;
config->enablePinOut = false;
config->enableHysteresisBothDirections = false;
config->hysteresisMode = kACMP_hysteresisMode0;
```

Parameters

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

5.6.4 void ACMP_Enable (CMP_Type * *base*, bool *enable*)

Parameters

<i>base</i>	ACMP peripheral base address.
<i>enable</i>	True to enable the ACMP.

5.6.5 void ACMP_SetChannelConfig (CMP_Type * *base*, const acmp_channel_config_t * *config*)

Note that the plus/minus mux's setting is only valid when the positive/negative port's input isn't from DAC but from channel mux.

Example:

```
acmp_channel_config_t configStruct = {0};
configStruct.positivePortInput = kACMP_PortInputFromDAC;
configStruct.negativePortInput = kACMP_PortInputFromMux;
configStruct.minusMuxInput = 1U;
ACMP_SetChannelConfig(CMP0, &configStruct);
```

Parameters

<i>base</i>	ACMP peripheral base address.
<i>config</i>	Pointer to channel configuration structure.

5.6.6 void ACMP_EnableDMA (CMP_Type * *base*, bool *enable*)

Parameters

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

<i>enable</i>	True to enable DMA.
---------------	---------------------

5.6.7 void ACMP_EnableWindowMode (CMP_Type * *base*, bool *enable*)

Parameters

<i>base</i>	ACMP peripheral base address.
<i>enable</i>	True to enable window mode.

5.6.8 void ACMP_SetFilterConfig (CMP_Type * *base*, const acmp_filter_config_t * *config*)

The filter can be enabled when the filter count is bigger than 1, the filter period is greater than 0 and the sample clock is from divided bus clock or the filter is bigger than 1 and the sample clock is from external clock. Detailed usage can be got from the reference manual.

Example:

```
acmp_filter_config_t configStruct = {0};
configStruct.filterCount = 5U;
configStruct.filterPeriod = 200U;
configStruct.enableSample = false;
ACMP_SetFilterConfig(CMP0, &configStruct);
```

Parameters

<i>base</i>	ACMP peripheral base address.
<i>config</i>	Pointer to filter configuration structure.

5.6.9 void ACMP_SetDACCConfig (CMP_Type * *base*, const acmp_dac_config_t * *config*)

Example:

```
acmp_dac_config_t configStruct = {0};
configStruct.referenceVoltageSource = kACMP_VrefSourceVin1;
configStruct.DACValue = 20U;
configStruct.enableOutput = false;
configStruct.workMode = kACMP_DACWorkLowSpeedMode;
ACMP_SetDACCConfig(CMP0, &configStruct);
```

Parameters

<i>base</i>	ACMP peripheral base address.
<i>config</i>	Pointer to DAC configuration structure. "NULL" is for disabling the feature.

5.6.10 void ACMP_SetRoundRobinConfig (CMP_Type * *base*, const acmp_round_robin_config_t * *config*)

Example:

```
acmp_round_robin_config_t configStruct = {0};
configStruct.fixedPort = kACMP_FixedPlusPort;
configStruct.fixedChannelNumber = 3U;
configStruct.checkerChannelMask = 0xF7U;
configStruct.sampleClockCount = 0U;
configStruct.delayModulus = 0U;
ACMP_SetRoundRobinConfig(CMP0, &configStruct);
```

Parameters

<i>base</i>	ACMP peripheral base address.
<i>config</i>	Pointer to round robin mode configuration structure. "NULL" is for disabling the feature.

5.6.11 void ACMP_SetRoundRobinPreState (CMP_Type * *base*, uint32_t *mask*)

Note: The pre-state has different circuit with get-round-robin-result in the SOC even though they are same bits. So get-round-robin-result can't return the same value as the value are set by pre-state.

Parameters

<i>base</i>	ACMP peripheral base address.
<i>mask</i>	Mask of round robin channel index. Available range is channel0:0x01 to channel7-:0x80.

5.6.12 static uint32_t ACMP_GetRoundRobinStatusFlags (CMP_Type * *base*) [inline], [static]

Parameters

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

Returns

Mask of channel input changed asserted flags. Available range is channel0:0x01 to channel7:0x80.

5.6.13 void ACMP_ClearRoundRobinStatusFlags (CMP_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	ACMP peripheral base address.
<i>mask</i>	Mask of channel index. Available range is channel0:0x01 to channel7:0x80.

5.6.14 static uint32_t ACMP_GetRoundRobinResult (CMP_Type * *base*) [inline], [static]

Note that the set-pre-state has different circuit with get-round-robin-result in the SOC even though they are same bits. So [ACMP_GetRoundRobinResult\(\)](#) can't return the same value as the value are set by ACMP_SetRoundRobinPreState.

Parameters

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

Returns

Mask of round robin channel result. Available range is channel0:0x01 to channel7:0x80.

5.6.15 void ACMP_EnableInterrupts (CMP_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	ACMP peripheral base address.
<i>mask</i>	Interrupts mask. See "_acmp_interrupt_enable".

5.6.16 void ACMP_DisableInterrupts (CMP_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	ACMP peripheral base address.
<i>mask</i>	Interrupts mask. See "_acmp_interrupt_enable".

5.6.17 uint32_t ACMP_GetStatusFlags (CMP_Type * *base*)

Parameters

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

Returns

Status flags asserted mask. See "_acmp_status_flags".

5.6.18 void ACMP_ClearStatusFlags (CMP_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	ACMP peripheral base address.
<i>mask</i>	Status flags mask. See "_acmp_status_flags".

Chapter 6

ADC12: Analog-to-Digital Converter

6.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Analog-to-Digital Converter (ADC12) module of MCUXpresso SDK devices.

The ADC12 driver is created to help the user better operate the ADC12 module. This driver can be considered a basic analog-to-digital converter with advanced features. The APIs for basic operations can make the ADC12 work as a general converter, which can convert the analog input to be a digital value. The APIs for advanced operations can be used as the plug-in function based on the basic operations. They can provide more ways to process the converter's conversion results, such DMA trigger, hardware compare, hardware average, and so on.

Note that channel 26 of ADC12 is connected to a internal temperature sensor of the module. If you want to get the best conversion result of the temperature value, set the field "sampleClockCount" in the structure "adc12_config_t" to be maximum value when you call the API "ADC12_Init()". This field indicates the sample time of the analog input signal. A longer sample time makes the conversion result of the analog input signal more stable and accurate.

6.2 Function groups

6.2.1 Initialization and deinitialization

This function group implement ADC12 initialization and deinitialization API.

6.2.2 Basic Operations

This function group implement basic ADC12 operation API.

6.2.3 Advanced Operations

This function group implement advanced ADC12 operation API.

6.3 Typical use case

6.3.1 Normal Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/adc12

6.3.2 Interrupt Configuration

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/adc12

Data Structures

- struct `adc12_config_t`
Converter configuration. [More...](#)
- struct `adc12_hardware_compare_config_t`
Hardware compare configuration. [More...](#)
- struct `adc12_channel_config_t`
Channel conversion configuration. [More...](#)

Macros

- #define `FSL_ADC12_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 6)`)
ADC12 driver version.

Enumerations

- enum `_adc12_channel_status_flags` { `kADC12_ChannelConversionCompletedFlag` = ADC_SC1_COCO_MASK }
Channel status flags' mask.
- enum `_adc12_status_flags` {
`kADC12_ActiveFlag` = ADC_SC2_ADACT_MASK,
`kADC12_CalibrationFailedFlag` = (ADC_SC2_ADACT_MASK << 1U) }
Converter status flags' mask.
- enum `adc12_clock_divider_t` {
`kADC12_ClockDivider1` = 0U,
`kADC12_ClockDivider2` = 1U,
`kADC12_ClockDivider4` = 2U,
`kADC12_ClockDivider8` = 3U }
Clock divider for the converter.
- enum `adc12_resolution_t` {
`kADC12_Resolution8Bit` = 0U,
`kADC12_Resolution12Bit` = 1U,
`kADC12_Resolution10Bit` = 2U }
Converter's resolution.
- enum `adc12_clock_source_t` {
`kADC12_ClockSourceAlt0` = 0U,
`kADC12_ClockSourceAlt1` = 1U,
`kADC12_ClockSourceAlt2` = 2U,
`kADC12_ClockSourceAlt3` = 3U }
Conversion clock source.
- enum `adc12_reference_voltage_source_t` {
`kADC12_ReferenceVoltageSourceVref` = 0U,
`kADC12_ReferenceVoltageSourceValt` = 1U }
Reference voltage source.

- enum `adc12_hardware_average_mode_t` {

 `kADC12_HardwareAverageCount4` = 0U,
`kADC12_HardwareAverageCount8` = 1U,
`kADC12_HardwareAverageCount16` = 2U,
`kADC12_HardwareAverageCount32` = 3U,
`kADC12_HardwareAverageDisabled` = 4U }

Hardware average mode.

- enum `adc12_hardware_compare_mode_t` {

 `kADC12_HardwareCompareMode0` = 0U,
`kADC12_HardwareCompareMode1` = 1U,
`kADC12_HardwareCompareMode2` = 2U,
`kADC12_HardwareCompareMode3` = 3U }

Hardware compare mode.

Initialization

- void `ADC12_Init` (`ADC_Type` *base, const `adc12_config_t` *config)
Initialize the ADC12 module.
- void `ADC12_Deinit` (`ADC_Type` *base)
De-initialize the ADC12 module.
- void `ADC12_GetDefaultConfig` (`adc12_config_t` *config)
Gets an available pre-defined settings for converter's configuration.

Basic Operations

- void `ADC12_SetChannelConfig` (`ADC_Type` *base, `uint32_t` channelGroup, const `adc12_channel_config_t` *config)
Configure the conversion channel.
- static `uint32_t` `ADC12_GetChannelConversionValue` (`ADC_Type` *base, `uint32_t` channelGroup)
Get the conversion value.
- `uint32_t` `ADC12_GetChannelStatusFlags` (`ADC_Type` *base, `uint32_t` channelGroup)
Get the status flags of channel.

Advanced Operations

- `status_t` `ADC12_DoAutoCalibration` (`ADC_Type` *base)
Automate the hardware calibration.
- static void `ADC12_SetOffsetValue` (`ADC_Type` *base, `uint32_t` value)
Set the offset value for the conversion result.
- static void `ADC12_SetGainValue` (`ADC_Type` *base, `uint32_t` value)
Set the gain value for the conversion result.
- static void `ADC12_EnabledDMA` (`ADC_Type` *base, `bool` enable)
Enable generating the DMA trigger when conversion is completed.
- static void `ADC12_EnableHardwareTrigger` (`ADC_Type` *base, `bool` enable)
Enable or disable the hardware trigger mode.
- void `ADC12_SetHardwareCompareConfig` (`ADC_Type` *base, const `adc12_hardware_compare_config_t` *config)
Configure the hardware compare mode.
- void `ADC12_SetHardwareAverage` (`ADC_Type` *base, `adc12_hardware_average_mode_t` mode)

- `Set the hardware average mode.`
`uint32_t ADC12_GetStatusFlags (ADC_Type *base)`
`Get the status flags of the converter.`

6.4 Data Structure Documentation

6.4.1 struct adc12_config_t

Data Fields

- `adc12_reference_voltage_source_t referenceVoltageSource`
`Select the reference voltage source.`
- `adc12_clock_source_t clockSource`
`Select the input clock source to converter.`
- `adc12_clock_divider_t clockDivider`
`Select the divider of input clock source.`
- `adc12_resolution_t resolution`
`Select the sample resolution mode.`
- `uint32_t sampleClockCount`
`Select the sample clock count.`
- `bool enableContinuousConversion`
`Enable continuous conversion mode.`

Field Documentation

- (1) `adc12_reference_voltage_source_t adc12_config_t::referenceVoltageSource`
- (2) `adc12_clock_source_t adc12_config_t::clockSource`
- (3) `adc12_clock_divider_t adc12_config_t::clockDivider`
- (4) `adc12_resolution_t adc12_config_t::resolution`
- (5) `uint32_t adc12_config_t::sampleClockCount`

Add its value may improve the stability of the conversion result.

- (6) `bool adc12_config_t::enableContinuousConversion`

6.4.2 struct adc12_hardware_compare_config_t

Data Fields

- `adc12_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.`

Field Documentation

- (1) `adc12_hardware_compare_mode_t adc12_hardware_compare_config_t::hardwareCompareMode`
- (2) `int16_t adc12_hardware_compare_config_t::value1`
- (3) `int16_t adc12_hardware_compare_config_t::value2`

6.4.3 struct adc12_channel_config_t**Data Fields**

- `uint32_t channelNumber`
Setting the conversion channel number.
- `bool enableInterruptOnConversionCompleted`
Generate a interrupt request once the conversion is completed.

Field Documentation

- (1) `uint32_t adc12_channel_config_t::channelNumber`

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

- (2) `bool adc12_channel_config_t::enableInterruptOnConversionCompleted`

6.5 Macro Definition Documentation**6.5.1 #define FSL_ADC12_DRIVER_VERSION (MAKE_VERSION(2, 0, 6))**

Version 2.0.6.

6.6 Enumeration Type Documentation**6.6.1 enum _adc12_channel_status_flags**

Enumerator

kADC12_ChannelConversionCompletedFlag Conversion done.

6.6.2 enum _adc12_status_flags

Enumerator

kADC12_ActiveFlag Converter is active.

kADC12_CalibrationFailedFlag Calibration is failed.

6.6.3 enum adc12_clock_divider_t

Enumerator

- kADC12_ClockDivider1* For divider 1 from the input clock to the module.
- kADC12_ClockDivider2* For divider 2 from the input clock to the module.
- kADC12_ClockDivider4* For divider 4 from the input clock to the module.
- kADC12_ClockDivider8* For divider 8 from the input clock to the module.

6.6.4 enum adc12_resolution_t

Enumerator

- kADC12_Resolution8Bit* 8 bit resolution.
- kADC12_Resolution12Bit* 12 bit resolution.
- kADC12_Resolution10Bit* 10 bit resolution.

6.6.5 enum adc12_clock_source_t

Enumerator

- kADC12_ClockSourceAlt0* Alternate clock 1 (ADC_ALTCLK1).
- kADC12_ClockSourceAlt1* Alternate clock 2 (ADC_ALTCLK2).
- kADC12_ClockSourceAlt2* Alternate clock 3 (ADC_ALTCLK3).
- kADC12_ClockSourceAlt3* Alternate clock 4 (ADC_ALTCLK4).

6.6.6 enum adc12_reference_voltage_source_t

Enumerator

- kADC12_ReferenceVoltageSourceVref* For external pins pair of VrefH and VrefL.
- kADC12_ReferenceVoltageSourceValt* For alternate reference pair of ValtH and ValtL.

6.6.7 enum adc12_hardware_average_mode_t

Enumerator

- kADC12_HardwareAverageCount4* For hardware average with 4 samples.
- kADC12_HardwareAverageCount8* For hardware average with 8 samples.
- kADC12_HardwareAverageCount16* For hardware average with 16 samples.
- kADC12_HardwareAverageCount32* For hardware average with 32 samples.
- kADC12_HardwareAverageDisabled* Disable the hardware average feature.

6.6.8 enum adc12_hardware_compare_mode_t

Enumerator

kADC12_HardwareCompareMode0 x < value1.
kADC12_HardwareCompareMode1 x > value1.
kADC12_HardwareCompareMode2 if value1 <= value2, then x < value1 || x > value2; else, value1 > x > value2.
kADC12_HardwareCompareMode3 if value1 <= value2, then value1 <= x <= value2; else x >= value1 || x <= value2.

6.7 Function Documentation

6.7.1 void ADC12_Init (ADC_Type * *base*, const adc12_config_t * *config*)

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>config</i>	Pointer to "adc12_config_t" structure.

6.7.2 void ADC12_Deinit (ADC_Type * *base*)

Parameters

<i>base</i>	ADC12 peripheral base address.
-------------	--------------------------------

6.7.3 void ADC12_GetDefaultConfig (adc12_config_t * *config*)

This function initializes the converter configuration structure with an available settings. The default values are:

Example:

```
config->referenceVoltageSource = kADC12_ReferenceVoltageSourceVref;
config->clockSource = kADC12_ClockSourceAlt0;
config->clockDivider = kADC12_ClockDivider1;
config->resolution = kADC12_Resolution8Bit;
config->sampleClockCount = 12U;
config->enableContinuousConversion = false;
```

Parameters

<i>config</i>	Pointer to "adc12_config_t" structure.
---------------	--

6.7.4 void ADC12_SetChannelConfig (ADC_Type * *base*, uint32_t *channelGroup*, const adc12_channel_config_t * *config*)

This operation triggers the conversion in software trigger mode. 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 can have more than one group of status and control register, one for each conversion. The channel group parameter indicates which group of registers are used, 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 time, only one of the channel groups is actively controlling ADC conversions. Channel group 0 is used for both software and hardware trigger modes of operation. Channel groups 1 and greater indicate potentially multiple channel group registers for use only in hardware trigger mode. See the chip configuration information in the MCU reference manual about the number of SC1n registers (channel groups) specific to this device. None of the channel groups 1 or greater are used for software trigger operation and therefore writes to these channel groups do not initiate a new conversion. Updating channel group 0 while a different channel group is 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>	ADC12 peripheral base address.
<i>channelGroup</i>	Channel group index.
<i>config</i>	Pointer to "adc12_channel_config_t" structure.

6.7.5 static uint32_t ADC12_GetChannelConversionValue (ADC_Type * *base*, uint32_t *channelGroup*) [inline], [static]

Parameters

<i>base</i>	ADC12 peripheral base address.
-------------	--------------------------------

<i>channelGroup</i>	Channel group index.
---------------------	----------------------

Returns

Conversion value.

6.7.6 `uint32_t ADC12_GetChannelStatusFlags (ADC_Type * base, uint32_t channelGroup)`

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>channelGroup</i>	Channel group index.

Returns

Flags' mask if indicated flags are asserted. See to "_adc12_channel_status_flags".

6.7.7 `status_t ADC12_DoAutoCalibration (ADC_Type * base)`

This auto calibration helps to adjust the gain automatically according to the converter's working environment. Execute the calibration before conversion. Note that the software trigger should be used during calibration.

Parameters

<i>base</i>	ADC12 peripheral base address.
-------------	--------------------------------

Return values

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

6.7.8 `static void ADC12_SetOffsetValue (ADC_Type * base, uint32_t value) [inline], [static]`

This offset value takes effect on the conversion result. If the offset value is not zero, the conversion result is subtracted by it.

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>value</i>	Offset value.

6.7.9 static void ADC12_SetGainValue (**ADC_Type** * *base*, **uint32_t** *value*) [**inline**], [**static**]

This gain value takes effect on the conversion result. If the gain value is not zero, the conversion result is amplified as it.

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>value</i>	Gain value.

6.7.10 static void ADC12_EnableDMA (**ADC_Type** * *base*, **bool** *enable*) [**inline**], [**static**]

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>enable</i>	Switcher of DMA feature. "true" means to enable, "false" means to disable.

6.7.11 static void ADC12_EnableHardwareTrigger (**ADC_Type** * *base*, **bool** *enable*) [**inline**], [**static**]

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>enable</i>	Switcher of hardware trigger feature. "true" means to enable, "false" means not.

6.7.12 void ADC12_SetHardwareCompareConfig (ADC_Type * *base*, const adc12_hardware_compare_config_t * *config*)

The hardware compare mode provides a way to process the conversion result automatically by hardware. Only the result in compare range is available. To compare the range, see "adc12_hardware_compare_mode_t", or the reference manual document for more detailed information.

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>config</i>	Pointer to "adc12_hardware_compare_config_t" structure. Pass "NULL" to disable the feature.

6.7.13 void ADC12_SetHardwareAverage (ADC_Type * *base*, adc12_hardware_average_mode_t *mode*)

Hardware average mode provides a way to process the conversion result automatically by hardware. The multiple conversion results are accumulated and averaged internally. This aids to get more accurate conversion result.

Parameters

<i>base</i>	ADC12 peripheral base address.
<i>mode</i>	Setting hardware average mode. See to "adc12_hardware_average_mode_t".

6.7.14 uint32_t ADC12_GetStatusFlags (ADC_Type * *base*)

Parameters

<i>base</i>	ADC12 peripheral base address.
-------------	--------------------------------

Returns

Flags' mask if indicated flags are asserted. See to "_adc12_status_flags".

Chapter 7

Common Driver

7.1 Overview

The MCUXpresso SDK provides a driver for the common module of MCUXpresso SDK devices.

Macros

- `#define FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ 1`
Macro to use the default weak IRQ handler in drivers.
- `#define MAKE_STATUS(group, code) (((group)*100L) + (code)))`
Construct a status code value from a group and code number.
- `#define MAKE_VERSION(major, minor, bugfix) (((major) * 65536L) + ((minor) * 256L) + (bugfix))`
Construct the version number for drivers.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U`
No debug console.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U`
Debug console based on UART.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U`
Debug console based on LPUART.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U`
Debug console based on LPSCI.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U`
Debug console based on USBCDC.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U`
Debug console based on FLEXCOMM.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_IUART 6U`
Debug console based on i.MX UART.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_VUSART 7U`
Debug console based on LPC_VUSART.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART 8U`
Debug console based on LPC_USART.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_SWO 9U`
Debug console based on SWO.
- `#define DEBUG_CONSOLE_DEVICE_TYPE_QSCI 10U`
Debug console based on QSCI.
- `#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))`
Computes the number of elements in an array.

Typedefs

- `typedef int32_t status_t`
Type used for all status and error return values.

Enumerations

- enum `_status_groups` {
 `kStatusGroup_Generic` = 0,
 `kStatusGroup_FLASH` = 1,
 `kStatusGroup_LP SPI` = 4,
 `kStatusGroup_FLEXIO_SPI` = 5,
 `kStatusGroup_DSPI` = 6,
 `kStatusGroup_FLEXIO_UART` = 7,
 `kStatusGroup_FLEXIO_I2C` = 8,
 `kStatusGroup_LPI2C` = 9,
 `kStatusGroup_UART` = 10,
 `kStatusGroup_I2C` = 11,
 `kStatusGroup_LPSCI` = 12,
 `kStatusGroup_LPUART` = 13,
 `kStatusGroup_SPI` = 14,
 `kStatusGroup_XRDC` = 15,
 `kStatusGroup_SEMA42` = 16,
 `kStatusGroup_SDHC` = 17,
 `kStatusGroup_SDMMC` = 18,
 `kStatusGroup_SAI` = 19,
 `kStatusGroup_MCG` = 20,
 `kStatusGroup_SCG` = 21,
 `kStatusGroup_SD SPI` = 22,
 `kStatusGroup_FLEXIO_I2S` = 23,
 `kStatusGroup_FLEXIO_MCULCD` = 24,
 `kStatusGroup_FLASHIAP` = 25,
 `kStatusGroup_FLEXCOMM_I2C` = 26,
 `kStatusGroup_I2S` = 27,
 `kStatusGroup_IUART` = 28,
 `kStatusGroup_CSI` = 29,
 `kStatusGroup_MIPI_DSI` = 30,
 `kStatusGroup_SDRAMC` = 35,
 `kStatusGroup_POWER` = 39,
 `kStatusGroup_ENET` = 40,
 `kStatusGroup_PHY` = 41,
 `kStatusGroup_TRGMUX` = 42,
 `kStatusGroup_SMARTCARD` = 43,
 `kStatusGroup_LMEM` = 44,
 `kStatusGroup_QSPI` = 45,
 `kStatusGroup_DMA` = 50,
 `kStatusGroup_EDMA` = 51,
 `kStatusGroup_DMAMGR` = 52,
 `kStatusGroup_FLEXCAN` = 53,
 `kStatusGroup_LTC` = 54,
 `kStatusGroup_FLEXIO_CAMERA` = 55,
 `kStatusGroup_LPC_SPI` = 56,
 `kStatusGroup_LPC_USACARD` = 58,
 `kStatusGroup_SDIF` = 59,

```

kStatusGroup_BMA = 164 }

Status group numbers.
• enum {
    kStatus_Success = MAKE_STATUS(kStatusGroup_Generic, 0),
    kStatus_Fail = MAKE_STATUS(kStatusGroup_Generic, 1),
    kStatus_ReadOnly = MAKE_STATUS(kStatusGroup_Generic, 2),
    kStatus_OutOfRange = MAKE_STATUS(kStatusGroup_Generic, 3),
    kStatus_InvalidArgument = MAKE_STATUS(kStatusGroup_Generic, 4),
    kStatus_Timeout = MAKE_STATUS(kStatusGroup_Generic, 5),
    kStatus_NoTransferInProgress,
    kStatus_Busy = MAKE_STATUS(kStatusGroup_Generic, 7),
    kStatus_NoData }
Generic status return codes.

```

Functions

- void * **SDK_Malloc** (size_t size, size_t alignbytes)
Allocate memory with given alignment and aligned size.
- void **SDK_Free** (void *ptr)
Free memory.
- void **SDK_DelayAtLeastUs** (uint32_t delayTime_us, uint32_t coreClock_Hz)
Delay at least for some time.

Driver version

- #define **FSL_COMMON_DRIVER_VERSION** (MAKE_VERSION(2, 3, 2))
common driver version.

Min/max macros

- #define **MIN**(a, b) (((a) < (b)) ? (a) : (b))
- #define **MAX**(a, b) (((a) > (b)) ? (a) : (b))

UINT16_MAX/UINT32_MAX value

- #define **UINT16_MAX** ((uint16_t)-1)
- #define **UINT32_MAX** ((uint32_t)-1)

Suppress fallthrough warning macro

- #define **SUPPRESS_FALL_THROUGH_WARNING()**

7.2 Macro Definition Documentation

7.2.1 #define FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ 1

7.2.2 #define MAKE_STATUS(*group*, *code*) (((group)*100L + (code)))

7.2.3 #define MAKE_VERSION(major, minor, bugfix) (((major) * 65536L) + ((minor) * 256L) + (bugfix))

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused	Major Version	Minor Version	Bug Fix	
31	25 24	17 16	9 8	0

7.2.4 #define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 3, 2))

7.2.5 #define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U

7.2.6 #define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U

7.2.7 #define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U

7.2.8 #define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U

7.2.9 #define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U

7.2.10 #define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U

7.2.11 #define DEBUG_CONSOLE_DEVICE_TYPE_IUART 6U

7.2.12 #define DEBUG_CONSOLE_DEVICE_TYPE_VUSART 7U

7.2.13 #define DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART 8U

7.2.14 #define DEBUG_CONSOLE_DEVICE_TYPE_SWO 9U

7.2.15 #define DEBUG_CONSOLE_DEVICE_TYPE_QSCI 10U

7.2.16 #define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

7.3 Typedef Documentation

7.3.1 typedef int32_t status_t

7.4 Enumeration Type Documentation

7.4.1 enum _status_groups

Enumerator

- kStatusGroup_Generic*** Group number for generic status codes.
- kStatusGroup_FLASH*** Group number for FLASH status codes.
- kStatusGroup_LP SPI*** Group number for LP SPI status codes.
- kStatusGroup_FLEXIO_SPI*** Group number for FLEXIO SPI status codes.
- kStatusGroup_DSPI*** Group number for DSPI status codes.
- kStatusGroup_FLEXIO_UART*** Group number for FLEXIO UART status codes.
- kStatusGroup_FLEXIO_I2C*** Group number for FLEXIO I2C status codes.
- kStatusGroup_LPI2C*** Group number for LPI2C status codes.
- kStatusGroup_UART*** Group number for UART status codes.
- kStatusGroup_I2C*** Group number for I2C status codes.
- kStatusGroup_LPSCI*** Group number for LPSCI status codes.
- kStatusGroup_LPUART*** Group number for LPUART status codes.
- kStatusGroup_SPI*** Group number for SPI status code.
- kStatusGroup_XRDC*** Group number for XRDC status code.
- kStatusGroup_SEMA42*** Group number for SEMA42 status code.
- kStatusGroup_SDHC*** Group number for SDHC status code.
- kStatusGroup_SDMMC*** Group number for SDMMC status code.
- kStatusGroup_SAI*** Group number for SAI status code.
- kStatusGroup_MCG*** Group number for MCG status codes.
- kStatusGroup_SCG*** Group number for SCG status codes.
- kStatusGroup_SD SPI*** Group number for SD SPI status codes.
- kStatusGroup_FLEXIO_I2S*** Group number for FLEXIO I2S status codes.
- kStatusGroup_FLEXIO_MCU LCD*** Group number for FLEXIO LCD status codes.
- kStatusGroup_FLASHIAP*** Group number for FLASHIAP status codes.
- kStatusGroup_FLEXCOMM_I2C*** Group number for FLEXCOMM I2C status codes.
- kStatusGroup_I2S*** Group number for I2S status codes.
- kStatusGroup_IUART*** Group number for IUART status codes.
- kStatusGroup_CSI*** Group number for CSI status codes.
- kStatusGroup_MIPI_DSI*** Group number for MIPI DSI status codes.
- kStatusGroup_SDRAMC*** Group number for SDRAMC status codes.
- kStatusGroup_POWER*** Group number for POWER status codes.
- kStatusGroup_ENET*** Group number for ENET status codes.
- kStatusGroup_PHY*** Group number for PHY status codes.
- kStatusGroup_TRGMUX*** Group number for TRGMUX status codes.
- kStatusGroup_SMARTCARD*** Group number for SMARTCARD status codes.
- kStatusGroup_LMEM*** Group number for LMEM status codes.
- kStatusGroup_QSPI*** Group number for QSPI status codes.
- kStatusGroup_DMA*** Group number for DMA status codes.
- kStatusGroup_EDMA*** Group number for EDMA status codes.
- kStatusGroup_DMAMGR*** Group number for DMAMGR status codes.

kStatusGroup_FLEXCAN Group number for FlexCAN status codes.
kStatusGroup_LTC Group number for LTC status codes.
kStatusGroup_FLEXIO_CAMERA Group number for FLEXIO CAMERA status codes.
kStatusGroup_LPC_SPI Group number for LPC_SPI status codes.
kStatusGroup_LPC_USART Group number for LPC_USART status codes.
kStatusGroup_DMIC Group number for DMIC status codes.
kStatusGroup_SDIF Group number for SDIF status codes.
kStatusGroup_SPIFI Group number for SPIFI status codes.
kStatusGroup OTP Group number for OTP status codes.
kStatusGroup_MCAN Group number for MCAN status codes.
kStatusGroup_CAAM Group number for CAAM status codes.
kStatusGroup_ECSPI Group number for ECSPI status codes.
kStatusGroup_USDHC Group number for USDHC status codes.
kStatusGroup_LPC_I2C Group number for LPC_I2C status codes.
kStatusGroup_DCP Group number for DCP status codes.
kStatusGroup_MSCAN Group number for MSCAN status codes.
kStatusGroup_ESAI Group number for ESAI status codes.
kStatusGroup_FLEXSPI Group number for FLEXSPI status codes.
kStatusGroup_MMDC Group number for MMDC status codes.
kStatusGroup_PDM Group number for MIC status codes.
kStatusGroup_SDMA Group number for SDMA status codes.
kStatusGroup_ICS Group number for ICS status codes.
kStatusGroup_SPDIF Group number for SPDIF status codes.
kStatusGroup_LPC_MINISPI Group number for LPC_MINISPI status codes.
kStatusGroup_HASHCRYPT Group number for Hashcrypt status codes.
kStatusGroup_LPC_SPI_SSP Group number for LPC_SPI_SSP status codes.
kStatusGroup_I3C Group number for I3C status codes.
kStatusGroup_LPC_I2C_1 Group number for LPC_I2C_1 status codes.
kStatusGroup_NOTIFIER Group number for NOTIFIER status codes.
kStatusGroup_DebugConsole Group number for debug console status codes.
kStatusGroup_SEMC Group number for SEMC status codes.
kStatusGroup_ApplicationRangeStart Starting number for application groups.
kStatusGroup_IAP Group number for IAP status codes.
kStatusGroup_SFA Group number for SFA status codes.
kStatusGroup_SPC Group number for SPC status codes.
kStatusGroup_PUF Group number for PUF status codes.
kStatusGroup_TOUCH_PANEL Group number for touch panel status codes.
kStatusGroup_HAL_GPIO Group number for HAL GPIO status codes.
kStatusGroup_HAL_UART Group number for HAL UART status codes.
kStatusGroup_HAL_TIMER Group number for HAL TIMER status codes.
kStatusGroup_HAL_SPI Group number for HAL SPI status codes.
kStatusGroup_HAL_I2C Group number for HAL I2C status codes.
kStatusGroup_HAL_FLASH Group number for HAL FLASH status codes.
kStatusGroup_HAL_PWM Group number for HAL PWM status codes.
kStatusGroup_HAL_RNG Group number for HAL RNG status codes.

kStatusGroup_HAL_I2S Group number for HAL I2S status codes.
kStatusGroup_TIMERMANAGER Group number for TiMER MANAGER status codes.
kStatusGroup_SERIALMANAGER Group number for SERIAL MANAGER status codes.
kStatusGroup_LED Group number for LED status codes.
kStatusGroup_BUTTON Group number for BUTTON status codes.
kStatusGroup_EXTERN_EEPROM Group number for EXTERN EEPROM status codes.
kStatusGroup_SHELL Group number for SHELL status codes.
kStatusGroup_MEM_MANAGER Group number for MEM MANAGER status codes.
kStatusGroup_LIST Group number for List status codes.
kStatusGroup_OSA Group number for OSA status codes.
kStatusGroup_COMMON_TASK Group number for Common task status codes.
kStatusGroup_MSG Group number for messaging status codes.
kStatusGroup_SDK_OCOTP Group number for OCOTP status codes.
kStatusGroup_SDK_FLEXSPINOR Group number for FLEXSPINOR status codes.
kStatusGroup_CODEC Group number for codec status codes.
kStatusGroup_ASRC Group number for codec status ASRC.
kStatusGroup_OTFAD Group number for codec status codes.
kStatusGroup_SDIOSLV Group number for SDIOSLV status codes.
kStatusGroup_MECC Group number for MECC status codes.
kStatusGroup_ENET_QOS Group number for ENET_QOS status codes.
kStatusGroup_LOG Group number for LOG status codes.
kStatusGroup_I3CBUS Group number for I3CBUS status codes.
kStatusGroup_QSCI Group number for QSCI status codes.
kStatusGroup_SNT Group number for SNT status codes.
kStatusGroup_QUEUEDSPI Group number for QSPI status codes.
kStatusGroup_POWER_MANAGER Group number for POWER_MANAGER status codes.
kStatusGroup_IPED Group number for IPED status codes.
kStatusGroup_CSS_PKC Group number for CSS PKC status codes.
kStatusGroup_HOSTIF Group number for HOSTIF status codes.
kStatusGroup_CLIF Group number for CLIF status codes.
kStatusGroup_BMA Group number for BMA status codes.

7.4.2 anonymous enum

Enumerator

kStatus_Success Generic status for Success.
kStatus_Fail Generic status for Fail.
kStatus_ReadOnly Generic status for read only failure.
kStatus_OutOfRange Generic status for out of range access.
kStatus_InvalidArgument Generic status for invalid argument check.
kStatus_Timeout Generic status for timeout.
kStatus_NoTransferInProgress Generic status for no transfer in progress.
kStatus_Busy Generic status for module is busy.

kStatus_NoData Generic status for no data is found for the operation.

7.5 Function Documentation

7.5.1 void* SDK_Malloc (size_t *size*, size_t *alignbytes*)

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

<i>size</i>	The length required to malloc.
<i>alignbytes</i>	The alignment size.

Return values

<i>The</i>	allocated memory.
------------	-------------------

7.5.2 void SDK_Free (void * *ptr*)

Parameters

<i>ptr</i>	The memory to be release.
------------	---------------------------

7.5.3 void SDK_DelayAtLeastUs (uint32_t *delayTime_us*, uint32_t *coreClock_Hz*)

Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

<i>delayTime_us</i>	Delay time in unit of microsecond.
<i>coreClock_Hz</i>	Core clock frequency with Hz.

Chapter 8

CRC: Cyclic Redundancy Check Driver

8.1 Overview

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

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

8.2 CRC Driver Initialization and Configuration

`CRC_Init()` function enables the clock gate for the CRC module in the SIM module and fully (re-)configures the CRC module according to the configuration structure. The seed member of the configuration structure is the initial checksum for which new data can be added to. When starting a new checksum computation, the seed is set to the initial checksum per the CRC protocol specification. For continued checksum operation, the seed is set to the intermediate checksum value as obtained from previous calls to `CRC_Get16bitResult()` or `CRC_Get32bitResult()` function. After calling the `CRC_Init()`, one or multiple `CRC_WriteData()` calls follow to update the checksum with data and `CRC_Get16bitResult()` or `CRC_Get32bitResult()` follow to read the result. The `crcResult` member of the configuration structure determines whether the `CRC_Get16bitResult()` or `CRC_Get32bitResult()` return value is a final checksum or an intermediate checksum. The `CRC_Init()` function can be called as many times as required allowing for runtime changes of the CRC protocol.

`CRC_GetDefaultConfig()` function can be used to set the module configuration structure with parameters for CRC-16/CCIT-FALSE protocol.

8.3 CRC Write Data

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

8.4 CRC Get Checksum

The `CRC_Get16bitResult()` or `CRC_Get32bitResult()` function reads the CRC module data register. Depending on the prior CRC module usage, the return value is either an intermediate checksum or the final checksum. For example, for 16-bit CRCs the following call sequences can be used.

`CRC_Init() / CRC_WriteData() / CRC_Get16bitResult()` to get the final checksum.

`CRC_Init() / CRC_WriteData() / ... / CRC_WriteData() / CRC_Get16bitResult()` to get the final checksum.

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

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

8.5 Comments about API usage in RTOS

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

The triplets

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

The triplets are protected by the RTOS mutex to protect the CRC module against concurrent accesses from different tasks. This is an example. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crcRefer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/crc

Data Structures

- struct `crc_config_t`
CRC protocol configuration. [More...](#)

Macros

- #define `CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT` 1
Default configuration structure filled by `CRC_GetDefaultConfig()`.

Enumerations

- enum `crc_bits_t` {
`kCrcBits16` = 0U,
`kCrcBits32` = 1U }
CRC bit width.
- enum `crc_result_t` {
`kCrcFinalChecksum` = 0U,
`kCrcIntermediateChecksum` = 1U }
CRC result type.

Functions

- void `CRC_Init` (CRC_Type *base, const `crc_config_t` *config)
Enables and configures the CRC peripheral module.
- static void `CRC_Deinit` (CRC_Type *base)
Disables the CRC peripheral module.
- void `CRC_GetDefaultConfig` (`crc_config_t` *config)

- `void CRC_WriteData(CRC_Type *base, const uint8_t *data, size_t dataSize)`
Writes data to the CRC module.
- `uint32_t CRC_Get32bitResult(CRC_Type *base)`
Reads the 32-bit checksum from the CRC module.
- `uint16_t CRC_Get16bitResult(CRC_Type *base)`
Reads a 16-bit checksum from the CRC module.

Driver version

- `#define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))`
CRC driver version.

8.6 Data Structure Documentation

8.6.1 struct crc_config_t

This structure holds the configuration for the CRC protocol.

Data Fields

- `uint32_t polynomial`
CRC Polynomial, MSBit first.
- `uint32_t seed`
Starting checksum value.
- `bool reflectIn`
Reflect bits on input.
- `bool reflectOut`
Reflect bits on output.
- `bool complementChecksum`
True if the result shall be complement of the actual checksum.
- `crc_bits_t crcBits`
Selects 16- or 32- bit CRC protocol.
- `crc_result_t crcResult`
Selects final or intermediate checksum return from `CRC_Get16bitResult()` or `CRC_Get32bitResult()`

Field Documentation

(1) `uint32_t crc_config_t::polynomial`

Example polynomial: $0x1021 = 1_0000_0010_0001 = x^{12}+x^5+1$

(2) `bool crc_config_t::reflectIn`

(3) `bool crc_config_t::reflectOut`

(4) `bool crc_config_t::complementChecksum`

(5) `crc_bits_t crc_config_t::crcBits`

8.7 Macro Definition Documentation

8.7.1 #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

Version 2.0.3.

Current version: 2.0.3

Change log:

- Version 2.0.3
 - Fix MISRA issues
- Version 2.0.2
 - Fix MISRA issues
- Version 2.0.1
 - move DATA and DATALL macro definition from header file to source file

8.7.2 #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1

Use CRC16-CCIT-FALSE as default.

8.8 Enumeration Type Documentation

8.8.1 enum crc_bits_t

Enumerator

kCrcBits16 Generate 16-bit CRC code.

kCrcBits32 Generate 32-bit CRC code.

8.8.2 enum crc_result_t

Enumerator

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

kCrcIntermediateChecksum CRC data register read value is intermediate checksum (raw value).

Reflect out and final xor protocol feature are not applied. Intermediate checksum can be used as a seed for [CRC_Init\(\)](#) to continue adding data to this checksum.

8.9 Function Documentation

8.9.1 void CRC_Init (**CRC_Type** * *base*, **const crc_config_t** * *config*)

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

Parameters

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

8.9.2 static void CRC_Deinit (**CRC_Type** * *base*) [inline], [static]

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

Parameters

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

8.9.3 void CRC_GetDefaultConfig (**crc_config_t** * *config*)

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

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

Parameters

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

8.9.4 void CRC_WriteData (**CRC_Type** * *base*, **const uint8_t** * *data*, **size_t** *dataSize*)

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

Parameters

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

8.9.5 `uint32_t CRC_Get32bitResult (CRC_Type * base)`

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

Parameters

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

Returns

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

8.9.6 `uint16_t CRC_Get16bitResult (CRC_Type * base)`

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

Parameters

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

Returns

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

Chapter 9

DMAMUX: Direct Memory Access Multiplexer Driver

9.1 Overview

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

9.2 Typical use case

9.2.1 DMAMUX Operation

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/dmamux

Driver version

- #define `FSL_DMAMUX_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 5)`)
DMAMUX driver version 2.0.5.

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.

9.3 Macro Definition Documentation

9.3.1 #define `FSL_DMAMUX_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 5)`)

9.4 Function Documentation

9.4.1 void DMAMUX_Init(**DMAMUX_Type** * *base*)

This function ungates the DMAMUX clock.

Parameters

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

9.4.2 void DMAMUX_Deinit (DMAMUX_Type * *base*)

This function gates the DMAMUX clock.

Parameters

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

9.4.3 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.

9.4.4 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.
-------------	---------------------------------

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

9.4.5 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.

9.4.6 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.

9.4.7 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 10

eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

10.1 Overview

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

10.2 Typical use case

10.2.1 eDMA Operation

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/edma

Data Structures

- struct [edma_config_t](#)
eDMA global configuration structure. [More...](#)
- struct [edma_transfer_config_t](#)
eDMA transfer configuration [More...](#)
- struct [edma_channel_Preemption_config_t](#)
eDMA channel priority configuration [More...](#)
- struct [edma_minor_offset_config_t](#)
eDMA minor offset configuration [More...](#)
- struct [edma_tcd_t](#)
eDMA TCD. [More...](#)
- struct [edma_handle_t](#)
eDMA transfer handle structure [More...](#)

Macros

- #define [DMA_DCHPRI_INDEX](#)(channel) (((channel) & ~0x03U) | (3U - ((channel)&0x03U)))
Compute the offset unit from DCHPRI3.

Typedefs

- typedef void(* [edma_callback](#))(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tcds)
Define callback function for eDMA.

Enumerations

- enum `edma_transfer_size_t` {

 `kEDMA_TransferSize1Bytes` = 0x0U,

 `kEDMA_TransferSize2Bytes` = 0x1U,

 `kEDMA_TransferSize4Bytes` = 0x2U,

 `kEDMA_TransferSize8Bytes` = 0x3U,

 `kEDMA_TransferSize16Bytes` = 0x4U,

 `kEDMA_TransferSize32Bytes` = 0x5U }

eDMA transfer configuration

- enum `edma_modulo_t` {

 `kEDMA_ModuloDisable` = 0x0U,

 `kEDMA_Modulo2bytes`,

 `kEDMA_Modulo4bytes`,

 `kEDMA_Modulo8bytes`,

 `kEDMA_Modulo16bytes`,

 `kEDMA_Modulo32bytes`,

 `kEDMA_Modulo64bytes`,

 `kEDMA_Modulo128bytes`,

 `kEDMA_Modulo256bytes`,

 `kEDMA_Modulo512bytes`,

 `kEDMA_Modulo1Kbytes`,

 `kEDMA_Modulo2Kbytes`,

 `kEDMA_Modulo4Kbytes`,

 `kEDMA_Modulo8Kbytes`,

 `kEDMA_Modulo16Kbytes`,

 `kEDMA_Modulo32Kbytes`,

 `kEDMA_Modulo64Kbytes`,

 `kEDMA_Modulo128Kbytes`,

 `kEDMA_Modulo256Kbytes`,

 `kEDMA_Modulo512Kbytes`,

 `kEDMA_Modulo1Mbytes`,

 `kEDMA_Modulo2Mbytes`,

 `kEDMA_Modulo4Mbytes`,

 `kEDMA_Modulo8Mbytes`,

 `kEDMA_Modulo16Mbytes`,

 `kEDMA_Modulo32Mbytes`,

 `kEDMA_Modulo64Mbytes`,

 `kEDMA_Modulo128Mbytes`,

 `kEDMA_Modulo256Mbytes`,

 `kEDMA_Modulo512Mbytes`,

 `kEDMA_Modulo1Gbytes`,

 `kEDMA_Modulo2Gbytes` }

eDMA modulo configuration

- enum `edma_bandwidth_t` {

```

kEDMA_BandwidthStallNone = 0x0U,
kEDMA_BandwidthStall4Cycle = 0x2U,
kEDMA_BandwidthStall8Cycle = 0x3U }

Bandwidth control.
• enum edma_channel_link_type_t {
    kEDMA_LinkNone = 0x0U,
    kEDMA_MinorLink,
    kEDMA_MajorLink }

Channel link type.
• enum {
    kEDMA_DoneFlag = 0x1U,
    kEDMA_ErrorFlag = 0x2U,
    kEDMA_InterruptFlag = 0x4U }

_edma_channel_status_flags eDMA channel status flags.
• enum {
    kEDMA_DestinationBusErrorFlag = DMA_ES_DBE_MASK,
    kEDMA_SourceBusErrorFlag = DMA_ES_SBE_MASK,
    kEDMA_ScatterGatherErrorFlag = DMA_ES_SGE_MASK,
    kEDMA_NbytesErrorFlag = DMA_ES_NCE_MASK,
    kEDMA_DestinationOffsetErrorFlag = DMA_ES_DOE_MASK,
    kEDMA_DestinationAddressErrorFlag = DMA_ES_DAE_MASK,
    kEDMA_SourceOffsetErrorFlag = DMA_ES_SOE_MASK,
    kEDMA_SourceAddressErrorFlag = DMA_ES_SAE_MASK,
    kEDMA_ErrorChannelFlag = DMA_ES_ERRCHN_MASK,
    kEDMA_ChannelPriorityErrorFlag = DMA_ES_CPE_MASK,
    kEDMA_TransferCanceledFlag = DMA_ES_ECX_MASK,
    kEDMA_ValidFlag = (int)DMA_ES_VLD_MASK }

_edma_error_status_flags eDMA channel error status flags.
• enum edma_interrupt_enable_t {
    kEDMA_ErrorInterruptEnable = 0x1U,
    kEDMA_MajorInterruptEnable = DMA_CSR_INTMAJOR_MASK,
    kEDMA_HalfInterruptEnable = DMA_CSR_INTHALF_MASK }

eDMA interrupt source
• enum edma_transfer_type_t {
    kEDMA_MemoryToMemory = 0x0U,
    kEDMA_PeripheralToMemory,
    kEDMA_MemoryToPeripheral,
    kEDMA_PeripheralToPeripheral }

eDMA transfer type
• enum {
    kStatus_EDMA_QueueFull = MAKE_STATUS(kStatusGroup_EDMA, 0),
    kStatus_EDMA_Busy = MAKE_STATUS(kStatusGroup_EDMA, 1) }

_edma_transfer_status eDMA transfer status

```

Driver version

- #define `FSL_EDMA_DRIVER_VERSION` (`MAKE_VERSION(2, 4, 3)`)

eDMA driver version

eDMA initialization and de-initialization

- void **EDMA_Init** (DMA_Type *base, const **edma_config_t** *config)
Initializes the eDMA peripheral.
- void **EDMA_Deinit** (DMA_Type *base)
Deinitializes the eDMA peripheral.
- void **EDMA_InstallTCD** (DMA_Type *base, uint32_t channel, **edma_tcd_t** *tcd)
Push content of TCD structure into hardware TCD register.
- void **EDMA_GetDefaultConfig** (**edma_config_t** *config)
Gets the eDMA default configuration structure.
- static void **EDMA_EnableContinuousChannelLinkMode** (DMA_Type *base, bool enable)
Enable/Disable continuous channel link mode.
- static void **EDMA_EnableMinorLoopMapping** (DMA_Type *base, bool enable)
Enable/Disable minor loop mapping.

eDMA Channel Operation

- void **EDMA_ResetChannel** (DMA_Type *base, uint32_t channel)
Sets all TCD registers to default values.
- void **EDMA_SetTransferConfig** (DMA_Type *base, uint32_t channel, const **edma_transfer_config_t** *config, **edma_tcd_t** *nextTcd)
Configures the eDMA transfer attribute.
- void **EDMA_SetMinorOffsetConfig** (DMA_Type *base, uint32_t channel, const **edma_minor_offset_config_t** *config)
Configures the eDMA minor offset feature.
- void **EDMA_SetChannelPreemptionConfig** (DMA_Type *base, uint32_t channel, const **edma_channel_Preemption_config_t** *config)
Configures the eDMA channel preemption feature.
- void **EDMA_SetChannelLink** (DMA_Type *base, uint32_t channel, **edma_channel_link_type_t** linkType, uint32_t linkedChannel)
Sets the channel link for the eDMA transfer.
- void **EDMA_SetBandWidth** (DMA_Type *base, uint32_t channel, **edma_bandwidth_t** bandWidth)
Sets the bandwidth for the eDMA transfer.
- void **EDMA_SetModulo** (DMA_Type *base, uint32_t channel, **edma_modulo_t** srcModulo, **edma_modulo_t** destModulo)
Sets the source modulo and the destination modulo for the eDMA transfer.
- static void **EDMA_EnableAsyncRequest** (DMA_Type *base, uint32_t channel, bool enable)
Enables an async request for the eDMA transfer.
- static void **EDMA_EnableAutoStopRequest** (DMA_Type *base, uint32_t channel, bool enable)
Enables an auto stop request for the eDMA transfer.
- void **EDMA_EnableChannelInterrupts** (DMA_Type *base, uint32_t channel, uint32_t mask)
Enables the interrupt source for the eDMA transfer.
- void **EDMA_DisableChannelInterrupts** (DMA_Type *base, uint32_t channel, uint32_t mask)
Disables the interrupt source for the eDMA transfer.
- void **EDMA_SetMajorOffsetConfig** (DMA_Type *base, uint32_t channel, int32_t sourceOffset, int32_t destOffset)
Configures the eDMA channel TCD major offset feature.

eDMA TCD Operation

- void [EDMA_TcdReset](#) (edma_tcd_t *tcd)
Sets all fields to default values for the TCD structure.
- void [EDMA_TcdSetTransferConfig](#) (edma_tcd_t *tcd, const edma_transfer_config_t *config, edma_tcd_t *nextTcd)
Configures the eDMA TCD transfer attribute.
- void [EDMA_TcdSetMinorOffsetConfig](#) (edma_tcd_t *tcd, const edma_minor_offset_config_t *config)
Configures the eDMA TCD minor offset feature.
- void [EDMA_TcdSetChannelLink](#) (edma_tcd_t *tcd, edma_channel_link_type_t linkType, uint32_t linkedChannel)
Sets the channel link for the eDMA TCD.
- static void [EDMA_TcdSetBandWidth](#) (edma_tcd_t *tcd, edma_bandwidth_t bandWidth)
Sets the bandwidth for the eDMA TCD.
- void [EDMA_TcdSetModulo](#) (edma_tcd_t *tcd, edma_modulo_t srcModulo, edma_modulo_t destModulo)
Sets the source modulo and the destination modulo for the eDMA TCD.
- static void [EDMA_TcdEnableAutoStopRequest](#) (edma_tcd_t *tcd, bool enable)
Sets the auto stop request for the eDMA TCD.
- void [EDMA_TcdEnableInterrupts](#) (edma_tcd_t *tcd, uint32_t mask)
Enables the interrupt source for the eDMA TCD.
- void [EDMA_TcdDisableInterrupts](#) (edma_tcd_t *tcd, uint32_t mask)
Disables the interrupt source for the eDMA TCD.
- void [EDMA_TcdSetMajorOffsetConfig](#) (edma_tcd_t *tcd, int32_t sourceOffset, int32_t destOffset)
Configures the eDMA TCD major offset feature.

eDMA Channel Transfer Operation

- static void [EDMA_EnableChannelRequest](#) (DMA_Type *base, uint32_t channel)
Enables the eDMA hardware channel request.
- static void [EDMA_DisableChannelRequest](#) (DMA_Type *base, uint32_t channel)
Disables the eDMA hardware channel request.
- static void [EDMA_TriggerChannelStart](#) (DMA_Type *base, uint32_t channel)
Starts the eDMA transfer by using the software trigger.

eDMA Channel Status Operation

- uint32_t [EDMA_GetRemainingMajorLoopCount](#) (DMA_Type *base, uint32_t channel)
Gets the remaining major loop count from the eDMA current channel TCD.
- static uint32_t [EDMA_GetErrorStatusFlags](#) (DMA_Type *base)
Gets the eDMA channel error status flags.
- uint32_t [EDMA_GetChannelStatusFlags](#) (DMA_Type *base, uint32_t channel)
Gets the eDMA channel status flags.
- void [EDMA_ClearChannelStatusFlags](#) (DMA_Type *base, uint32_t channel, uint32_t mask)
Clears the eDMA channel status flags.

eDMA Transactional Operation

- void [EDMA_CreateHandle](#) (edma_handle_t *handle, DMA_Type *base, uint32_t channel)
Creates the eDMA handle.

- void **EDMA_InstallTCDMemory** (**edma_handle_t** *handle, **edma_tcd_t** *tcdPool, **uint32_t** tcdSize)
Installs the TCDs memory pool into the eDMA handle.
- void **EDMA_SetCallback** (**edma_handle_t** *handle, **edma_callback** callback, void *userData)
Installs a callback function for the eDMA transfer.
- void **EDMA_PreparesTransferConfig** (**edma_transfer_config_t** *config, void *srcAddr, **uint32_t** srcWidth, **int16_t** srcOffset, void *destAddr, **uint32_t** destWidth, **int16_t** destOffset, **uint32_t** bytesEachRequest, **uint32_t** transferBytes)
Prepares the eDMA transfer structure configurations.
- void **EDMA_PreparesTransfer** (**edma_transfer_config_t** *config, void *srcAddr, **uint32_t** srcWidth, void *destAddr, **uint32_t** destWidth, **uint32_t** bytesEachRequest, **uint32_t** transferBytes, **edma_transfer_type_t** transferType)
Prepares the eDMA transfer structure.
- **status_t EDMA_SubmitTransfer** (**edma_handle_t** *handle, const **edma_transfer_config_t** *config)
Submits the eDMA transfer request.
- void **EDMA_StartTransfer** (**edma_handle_t** *handle)
eDMA starts transfer.
- void **EDMA_StopTransfer** (**edma_handle_t** *handle)
eDMA stops transfer.
- void **EDMA_AbortTransfer** (**edma_handle_t** *handle)
eDMA aborts transfer.
- static **uint32_t EDMA_GetUnusedTCDNumber** (**edma_handle_t** *handle)
Get unused TCD slot number.
- static **uint32_t EDMA_GetNextTCDAddress** (**edma_handle_t** *handle)
Get the next tcd address.
- void **EDMA_HandleIRQ** (**edma_handle_t** *handle)
eDMA IRQ handler for the current major loop transfer completion.

10.3 Data Structure Documentation

10.3.1 struct edma_config_t

Data Fields

- bool **enableContinuousLinkMode**
Enable (true) continuous link mode.
- bool **enableHaltOnError**
Enable (true) transfer halt on error.
- bool **enableRoundRobinArbitration**
Enable (true) round robin channel arbitration method or fixed priority arbitration is used for channel selection.
- bool **enableDebugMode**
Enable(true) eDMA debug mode.

Field Documentation

(1) bool **edma_config_t::enableContinuousLinkMode**

Upon minor loop completion, the channel activates again if that channel has a minor loop channel link enabled and the link channel is itself.

(2) bool edma_config_t::enableHaltOnError

Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

(3) bool edma_config_t::enableDebugMode

When in debug mode, the eDMA stalls the start of a new channel. Executing channels are allowed to complete.

10.3.2 struct edma_transfer_config_t

This structure configures the source/destination transfer attribute.

Data Fields

- **uint32_t srcAddr**
Source data address.
- **uint32_t destAddr**
Destination data address.
- **edma_transfer_size_t srcTransferSize**
Source data transfer size.
- **edma_transfer_size_t destTransferSize**
Destination data transfer size.
- **int16_t srcOffset**
Sign-extended offset applied to the current source address to form the next-state value as each source read is completed.
- **int16_t destOffset**
Sign-extended offset applied to the current destination address to form the next-state value as each destination write is completed.
- **uint32_t minorLoopBytes**
Bytes to transfer in a minor loop.
- **uint32_t majorLoopCounts**
Major loop iteration count.

Field Documentation**(1) uint32_t edma_transfer_config_t::srcAddr****(2) uint32_t edma_transfer_config_t::destAddr****(3) edma_transfer_size_t edma_transfer_config_t::srcTransferSize****(4) edma_transfer_size_t edma_transfer_config_t::destTransferSize****(5) int16_t edma_transfer_config_t::srcOffset**

- (6) int16_t edma_transfer_config_t::destOffset
- (7) uint32_t edma_transfer_config_t::majorLoopCounts

10.3.3 struct edma_channel_Preemption_config_t

Data Fields

- bool enableChannelPreemption
If true: a channel can be suspended by other channel with higher priority.
- bool enablePreemptAbility
If true: a channel can suspend other channel with low priority.
- uint8_t channelPriority
Channel priority.

10.3.4 struct edma_minor_offset_config_t

Data Fields

- bool enableSrcMinorOffset
Enable(true) or Disable(false) source minor loop offset.
- bool enableDestMinorOffset
Enable(true) or Disable(false) destination minor loop offset.
- uint32_t minorOffset
Offset for a minor loop mapping.

Field Documentation

- (1) bool edma_minor_offset_config_t::enableSrcMinorOffset
- (2) bool edma_minor_offset_config_t::enableDestMinorOffset
- (3) uint32_t edma_minor_offset_config_t::minorOffset

10.3.5 struct edma_tcd_t

This structure is same as TCD register which is described in reference manual, and is used to configure the scatter/gather feature as a next hardware TCD.

Data Fields

- __IO uint32_t SADDR
SADDR register, used to save source address.
- __IO uint16_t SOFF
SOFF register, save offset bytes every transfer.
- __IO uint16_t ATTR

- **ATTR register; source/destination transfer size and modulo.**
- **_IO uint32_t NBYTES**
Nbytes register, minor loop length in bytes.
- **_IO uint32_t SLAST**
SLAST register.
- **_IO uint32_t DADDR**
DADDR register, used for destination address.
- **_IO uint16_t DOFF**
DOFF register, used for destination offset.
- **_IO uint16_t CITER**
CITER register, current minor loop numbers, for unfinished minor loop.
- **_IO uint32_t DLAST_SGA**
DLASTSGA register, next tcd address used in scatter-gather mode.
- **_IO uint16_t CSR**
CSR register, for TCD control status.
- **_IO uint16_t BITER**
BITER register, begin minor loop count.

Field Documentation

(1) **_IO uint16_t edma_tcd_t::CITER**

(2) **_IO uint16_t edma_tcd_t::BITER**

10.3.6 struct edma_handle_t

Data Fields

- **edma_callback callback**
Callback function for major count exhausted.
- **void * userData**
Callback function parameter.
- **DMA_Type * base**
eDMA peripheral base address.
- **edma_tcd_t * tcdPool**
Pointer to memory stored TCDs.
- **uint8_t channel**
eDMA channel number.
- **volatile int8_t header**
The first TCD index.
- **volatile int8_t tail**
The last TCD index.
- **volatile int8_t tcdUsed**
The number of used TCD slots.
- **volatile int8_t tcdSize**
The total number of TCD slots in the queue.
- **uint8_t flags**
The status of the current channel.

Field Documentation

- (1) `edma_callback edma_handle_t::callback`
- (2) `void* edma_handle_t::userData`
- (3) `DMA_Type* edma_handle_t::base`
- (4) `edma_tcd_t* edma_handle_t::tcdPool`
- (5) `uint8_t edma_handle_t::channel`
- (6) `volatile int8_t edma_handle_t::header`

Should point to the next TCD to be loaded into the eDMA engine.

- (7) `volatile int8_t edma_handle_t::tail`

Should point to the next TCD to be stored into the memory pool.

- (8) `volatile int8_t edma_handle_t::tcdUsed`

Should reflect the number of TCDs can be used/loaded in the memory.

- (9) `volatile int8_t edma_handle_t::tcdSize`

- (10) `uint8_t edma_handle_t::flags`

10.4 Macro Definition Documentation

10.4.1 `#define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 4, 3))`

Version 2.4.3.

10.5 Typedef Documentation

10.5.1 `typedef void(* edma_callback)(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tclds)`

This callback function is called in the EDMA interrupt handle. In normal mode, run into callback function means the transfer users need is done. In scatter gather mode, run into callback function means a transfer control block (tcd) is finished. Not all transfer finished, users can get the finished tcd numbers using interface EDMA_GetUnusedTCDNumber.

Parameters

<i>handle</i>	EDMA handle pointer, users shall not touch the values inside.
<i>userData</i>	The callback user parameter pointer. Users can use this parameter to involve things users need to change in EDMA callback function.
<i>transferDone</i>	If the current loaded transfer done. In normal mode it means if all transfer done. In scatter gather mode, this parameter shows is the current transfer block in EDM-A register is done. As the load of core is different, it will be different if the new tcd loaded into EDMA registers while this callback called. If true, it always means new tcd still not loaded into registers, while false means new tcd already loaded into registers.
<i>tcds</i>	How many tcds are done from the last callback. This parameter only used in scatter gather mode. It tells user how many tcds are finished between the last callback and this.

10.6 Enumeration Type Documentation

10.6.1 enum edma_transfer_size_t

Enumerator

- kEDMA_TransferSize1Bytes*** Source/Destination data transfer size is 1 byte every time.
- kEDMA_TransferSize2Bytes*** Source/Destination data transfer size is 2 bytes every time.
- kEDMA_TransferSize4Bytes*** Source/Destination data transfer size is 4 bytes every time.
- kEDMA_TransferSize8Bytes*** Source/Destination data transfer size is 8 bytes every time.
- kEDMA_TransferSize16Bytes*** Source/Destination data transfer size is 16 bytes every time.
- kEDMA_TransferSize32Bytes*** Source/Destination data transfer size is 32 bytes every time.

10.6.2 enum edma_modulo_t

Enumerator

- kEDMA_ModuloDisable*** Disable modulo.
- kEDMA_Modulo2bytes*** Circular buffer size is 2 bytes.
- kEDMA_Modulo4bytes*** Circular buffer size is 4 bytes.
- kEDMA_Modulo8bytes*** Circular buffer size is 8 bytes.
- kEDMA_Modulo16bytes*** Circular buffer size is 16 bytes.
- kEDMA_Modulo32bytes*** Circular buffer size is 32 bytes.
- kEDMA_Modulo64bytes*** Circular buffer size is 64 bytes.
- kEDMA_Modulo128bytes*** Circular buffer size is 128 bytes.
- kEDMA_Modulo256bytes*** Circular buffer size is 256 bytes.
- kEDMA_Modulo512bytes*** Circular buffer size is 512 bytes.
- kEDMA_Modulo1Kbytes*** Circular buffer size is 1 K bytes.
- kEDMA_Modulo2Kbytes*** Circular buffer size is 2 K bytes.
- kEDMA_Modulo4Kbytes*** Circular buffer size is 4 K bytes.

kEDMA_Modulo8Kbytes Circular buffer size is 8 K bytes.
kEDMA_Modulo16Kbytes Circular buffer size is 16 K bytes.
kEDMA_Modulo32Kbytes Circular buffer size is 32 K bytes.
kEDMA_Modulo64Kbytes Circular buffer size is 64 K bytes.
kEDMA_Modulo128Kbytes Circular buffer size is 128 K bytes.
kEDMA_Modulo256Kbytes Circular buffer size is 256 K bytes.
kEDMA_Modulo512Kbytes Circular buffer size is 512 K bytes.
kEDMA_Modulo1Mbytes Circular buffer size is 1 M bytes.
kEDMA_Modulo2Mbytes Circular buffer size is 2 M bytes.
kEDMA_Modulo4Mbytes Circular buffer size is 4 M bytes.
kEDMA_Modulo8Mbytes Circular buffer size is 8 M bytes.
kEDMA_Modulo16Mbytes Circular buffer size is 16 M bytes.
kEDMA_Modulo32Mbytes Circular buffer size is 32 M bytes.
kEDMA_Modulo64Mbytes Circular buffer size is 64 M bytes.
kEDMA_Modulo128Mbytes Circular buffer size is 128 M bytes.
kEDMA_Modulo256Mbytes Circular buffer size is 256 M bytes.
kEDMA_Modulo512Mbytes Circular buffer size is 512 M bytes.
kEDMA_Modulo1Gbytes Circular buffer size is 1 G bytes.
kEDMA_Modulo2Gbytes Circular buffer size is 2 G bytes.

10.6.3 enum edma_bandwidth_t

Enumerator

kEDMA_BandwidthStallNone No eDMA engine stalls.
kEDMA_BandwidthStall4Cycle eDMA engine stalls for 4 cycles after each read/write.
kEDMA_BandwidthStall8Cycle eDMA engine stalls for 8 cycles after each read/write.

10.6.4 enum edma_channel_link_type_t

Enumerator

kEDMA_LinkNone No channel link.
kEDMA_MinorLink Channel link after each minor loop.
kEDMA_MajorLink Channel link while major loop count exhausted.

10.6.5 anonymous enum

Enumerator

kEDMA_DoneFlag DONE flag, set while transfer finished, CITER value exhausted.
kEDMA_ErrorFlag eDMA error flag, an error occurred in a transfer
kEDMA_InterruptFlag eDMA interrupt flag, set while an interrupt occurred of this channel

10.6.6 anonymous enum

Enumerator

- kEDMA_DestinationBusErrorFlag* Bus error on destination address.
- kEDMA_SourceBusErrorFlag* Bus error on the source address.
- kEDMA_ScatterGatherErrorFlag* Error on the Scatter/Gather address, not 32byte aligned.
- kEDMA_NbytesErrorFlag* NBYTES/CITER configuration error.
- kEDMA_DestinationOffsetErrorFlag* Destination offset not aligned with destination size.
- kEDMA_DestinationAddressErrorFlag* Destination address not aligned with destination size.
- kEDMA_SourceOffsetErrorFlag* Source offset not aligned with source size.
- kEDMA_SourceAddressErrorFlag* Source address not aligned with source size.
- kEDMA_ErrorChannelFlag* Error channel number of the cancelled channel number.
- kEDMA_ChannelPriorityErrorFlag* Channel priority is not unique.
- kEDMA_TransferCanceledFlag* Transfer cancelled.
- kEDMA_ValidFlag* No error occurred, this bit is 0. Otherwise, it is 1.

10.6.7 enum edma_interrupt_enable_t

Enumerator

- kEDMA_ErrorInterruptEnable* Enable interrupt while channel error occurs.
- kEDMA_MajorInterruptEnable* Enable interrupt while major count exhausted.
- kEDMA_HalfInterruptEnable* Enable interrupt while major count to half value.

10.6.8 enum edma_transfer_type_t

Enumerator

- kEDMA_MemoryToMemory* Transfer from memory to memory.
- kEDMA_PeripheralToMemory* Transfer from peripheral to memory.
- kEDMA_MemoryToPeripheral* Transfer from memory to peripheral.
- kEDMA_PeripheralToPeripheral* Transfer from Peripheral to peripheral.

10.6.9 anonymous enum

Enumerator

- kStatus_EDMA_QueueFull* TCD queue is full.
- kStatus_EDMA_Busy* Channel is busy and can't handle the transfer request.

10.7 Function Documentation

10.7.1 void EDMA_Init (DMA_Type * *base*, const edma_config_t * *config*)

This function ungates the eDMA clock and configures the eDMA peripheral according to the configuration structure.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>config</i>	A pointer to the configuration structure, see "edma_config_t".

Note

This function enables the minor loop map feature.

10.7.2 void EDMA_Deinit (DMA_Type * *base*)

This function gates the eDMA clock.

Parameters

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

10.7.3 void EDMA_InstallTCD (DMA_Type * *base*, uint32_t *channel*, edma_tcd_t * *tcd*)

Parameters

<i>base</i>	EDMA peripheral base address.
<i>channel</i>	EDMA channel number.
<i>tcd</i>	Point to TCD structure.

10.7.4 void EDMA_GetDefaultConfig (edma_config_t * *config*)

This function sets the configuration structure to default values. The default configuration is set to the following values.

```
* config.enableContinuousLinkMode = false;
* config.enableHaltOnError = true;
* config.enableRoundRobinArbitration = false;
* config.enableDebugMode = false;
*
```

Parameters

<i>config</i>	A pointer to the eDMA configuration structure.
---------------	--

10.7.5 static void EDMA_EnableContinuousChannelLinkMode (DMA_Type * *base*, bool *enable*) [inline], [static]

Note

Do not use continuous link mode with a channel linking to itself if there is only one minor loop iteration per service request, for example, if the channel's NBYTES value is the same as either the source or destination size. The same data transfer profile can be achieved by simply increasing the NBYTES value, which provides more efficient, faster processing.

Parameters

<i>base</i>	EDMA peripheral base address.
<i>enable</i>	true is enable, false is disable.

10.7.6 static void EDMA_EnableMinorLoopMapping (DMA_Type * *base*, bool *enable*) [inline], [static]

The TCDn.word2 is redefined to include individual enable fields, an offset field, and the NBYTES field.

Parameters

<i>base</i>	EDMA peripheral base address.
<i>enable</i>	true is enable, false is disable.

10.7.7 void EDMA_ResetChannel (DMA_Type * *base*, uint32_t *channel*)

This function sets TCD registers for this channel to default values.

Parameters

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

Note

This function must not be called while the channel transfer is ongoing or it causes unpredictable results.

This function enables the auto stop request feature.

10.7.8 void EDMA_SetTransferConfig (DMA_Type * *base*, uint32_t *channel*, const edma_transfer_config_t * *config*, edma_tcd_t * *nextTcd*)

This function configures the transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the TCD address. Example:

```
* edma_transfer_t config;
* edma_tcd_t tcd;
* config.srcAddr = ...;
* config.destAddr = ...;
* ...
* EDMA_SetTransferConfig(DMA0, channel, &config, &tcd);
*
```

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>config</i>	Pointer to eDMA transfer configuration structure.
<i>nextTcd</i>	Point to TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

Note

If nextTcd is not NULL, it means scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the eDMA_ResetChannel.

10.7.9 void EDMA_SetMinorOffsetConfig (DMA_Type * *base*, uint32_t *channel*, const edma_minor_offset_config_t * *config*)

The minor offset means that the signed-extended value is added to the source address or destination address after each minor loop.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>config</i>	A pointer to the minor offset configuration structure.

10.7.10 void EDMA_SetChannelPreemptionConfig (DMA_Type * *base*, uint32_t *channel*, const edma_channel_Preemption_config_t * *config*)

This function configures the channel preemption attribute and the priority of the channel.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number
<i>config</i>	A pointer to the channel preemption configuration structure.

10.7.11 void EDMA_SetChannelLink (DMA_Type * *base*, uint32_t *channel*, edma_channel_link_type_t *linkType*, uint32_t *linkedChannel*)

This function configures either the minor link or the major link mode. The minor link means that the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>linkType</i>	A channel link type, which can be one of the following: <ul style="list-style-type: none"> • kEDMA_LinkNone • kEDMA_MinorLink • kEDMA_MajorLink

<i>linkedChannel</i>	The linked channel number.
----------------------	----------------------------

Note

Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

10.7.12 void EDMA_SetBandWidth (DMA_Type * *base*, uint32_t *channel*, edma_bandwidth_t *bandWidth*)

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>bandWidth</i>	A bandwidth setting, which can be one of the following: <ul style="list-style-type: none"> • kEDMABandwidthStallNone • kEDMABandwidthStall4Cycle • kEDMABandwidthStall8Cycle

10.7.13 void EDMA_SetModulo (DMA_Type * *base*, uint32_t *channel*, edma_modulo_t *srcModulo*, edma_modulo_t *destModulo*)

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

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

<i>srcModulo</i>	A source modulo value.
<i>destModulo</i>	A destination modulo value.

10.7.14 static void EDMA_EnableAsyncRequest (DMA_Type * *base*, uint32_t *channel*, bool *enable*) [inline], [static]

Parameters

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

10.7.15 static void EDMA_EnableAutoStopRequest (DMA_Type * *base*, uint32_t *channel*, bool *enable*) [inline], [static]

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

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

10.7.16 void EDMA_EnableChannelInterrupts (DMA_Type * *base*, uint32_t *channel*, uint32_t *mask*)

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>mask</i>	The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

10.7.17 void EDMA_DisableChannelInterrupts (DMA_Type * *base*, uint32_t *channel*, uint32_t *mask*)

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>mask</i>	The mask of the interrupt source to be set. Use the defined edma_interrupt_enable_t type.

10.7.18 void EDMA_SetMajorOffsetConfig (DMA_Type * *base*, uint32_t *channel*, int32_t *sourceOffset*, int32_t *destOffset*)

Adjustment value added to the source address at the completion of the major iteration count

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	edma channel number.
<i>sourceOffset</i>	source address offset will be applied to source address after major loop done.
<i>destOffset</i>	destination address offset will be applied to source address after major loop done.

10.7.19 void EDMA_TcdReset (edma_tcd_t * *tcd*)

This function sets all fields for this TCD structure to default value.

Parameters

<i>tcd</i>	Pointer to the TCD structure.
------------	-------------------------------

Note

This function enables the auto stop request feature.

10.7.20 void EDMA_TcdSetTransferConfig (edma_tcd_t * *tcd*, const edma_transfer_config_t * *config*, edma_tcd_t * *nextTcd*)

The TCD is a transfer control descriptor. The content of the TCD is the same as the hardware TCD registers. The TCD is used in the scatter-gather mode. This function configures the TCD transfer attribute, including source address, destination address, transfer size, address offset, and so on. It also configures the scatter gather feature if the user supplies the next TCD address. Example:

```

*   edma_transfer_t config = {
*     ...
*   }
*   edma_tcd_t tcd __aligned(32);
*   edma_tcd_t nextTcd __aligned(32);
*   EDMA_TcdSetTransferConfig(&tcd, &config, &nextTcd);
*

```

Parameters

<i>tcd</i>	Pointer to the TCD structure.
<i>config</i>	Pointer to eDMA transfer configuration structure.
<i>nextTcd</i>	Pointer to the next TCD structure. It can be NULL if users do not want to enable scatter/gather feature.

Note

TCD address should be 32 bytes aligned or it causes an eDMA error.

If the nextTcd is not NULL, the scatter gather feature is enabled and DREQ bit is cleared in the previous transfer configuration, which is set in the EDMA_TcdReset.

10.7.21 void EDMA_TcdSetMinorOffsetConfig (*edma_tcd_t * tcd, const edma_minor_offset_config_t * config*)

A minor offset is a signed-extended value added to the source address or a destination address after each minor loop.

Parameters

<i>tcd</i>	A point to the TCD structure.
<i>config</i>	A pointer to the minor offset configuration structure.

10.7.22 void EDMA_TcdSetChannelLink (*edma_tcd_t * tcd, edma_channel_link_type_t linkType, uint32_t linkedChannel*)

This function configures either a minor link or a major link. The minor link means the channel link is triggered every time CITER decreases by 1. The major link means that the channel link is triggered when the CITER is exhausted.

Note

Users should ensure that DONE flag is cleared before calling this interface, or the configuration is invalid.

Parameters

<i>tcd</i>	Point to the TCD structure.
<i>linkType</i>	Channel link type, it can be one of: <ul style="list-style-type: none"> • kEDMA_LinkNone • kEDMA_MinorLink • kEDMA_MajorLink
<i>linkedChannel</i>	The linked channel number.

10.7.23 static void EDMA_TcdSetBandWidth (*edma_tcd_t * tcd*, *edma_bandwidth_t bandwidth*) [inline], [static]

Because the eDMA processes the minor loop, it continuously generates read/write sequences until the minor count is exhausted. The bandwidth forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth seen by the crossbar switch.

Parameters

<i>tcd</i>	A pointer to the TCD structure.
<i>bandWidth</i>	A bandwidth setting, which can be one of the following: <ul style="list-style-type: none"> • kEDMABandwidthStallNone • kEDMABandwidthStall4Cycle • kEDMABandwidthStall8Cycle

10.7.24 void EDMA_TcdSetModulo (*edma_tcd_t * tcd*, *edma_modulo_t srcModulo*, *edma_modulo_t destModulo*)

This function defines a specific address range specified to be the value after (SADDR + SOFF)/(DADDR + DOFF) calculation is performed or the original register value. It provides the ability to implement a circular data queue easily.

Parameters

<i>tcd</i>	A pointer to the TCD structure.
------------	---------------------------------

<i>srcModulo</i>	A source modulo value.
<i>destModulo</i>	A destination modulo value.

10.7.25 static void EDMA_TcdEnableAutoStopRequest (*edma_tcd_t * tcd, bool enable*) [inline], [static]

If enabling the auto stop request, the eDMA hardware automatically disables the hardware channel request.

Parameters

<i>tcd</i>	A pointer to the TCD structure.
<i>enable</i>	The command to enable (true) or disable (false).

10.7.26 void EDMA_TcdEnableInterrupts (*edma_tcd_t * tcd, uint32_t mask*)

Parameters

<i>tcd</i>	Point to the TCD structure.
<i>mask</i>	The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

10.7.27 void EDMA_TcdDisableInterrupts (*edma_tcd_t * tcd, uint32_t mask*)

Parameters

<i>tcd</i>	Point to the TCD structure.
<i>mask</i>	The mask of interrupt source to be set. Users need to use the defined edma_interrupt_enable_t type.

10.7.28 void EDMA_TcdSetMajorOffsetConfig (*edma_tcd_t * tcd, int32_t sourceOffset, int32_t destOffset*)

Adjustment value added to the source address at the completion of the major iteration count

Parameters

<i>tcd</i>	A point to the TCD structure.
<i>sourceOffset</i>	source address offset will be applied to source address after major loop done.
<i>destOffset</i>	destination address offset will be applied to source address after major loop done.

10.7.29 static void EDMA_EnableChannelRequest (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

This function enables the hardware channel request.

Parameters

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

10.7.30 static void EDMA_DisableChannelRequest (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

This function disables the hardware channel request.

Parameters

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

10.7.31 static void EDMA_TriggerChannelStart (DMA_Type * *base*, uint32_t *channel*) [inline], [static]

This function starts a minor loop transfer.

Parameters

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

10.7.32 **uint32_t EDMA_GetRemainingMajorLoopCount (DMA_Type * *base*, uint32_t *channel*)**

This function checks the TCD (Task Control Descriptor) status for a specified eDMA channel and returns the number of major loop count that has not finished.

Parameters

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

Returns

Major loop count which has not been transferred yet for the current TCD.

Note

1. This function can only be used to get unfinished major loop count of transfer without the next TCD, or it might be inaccuracy.
 1. The unfinished/remaining transfer bytes cannot be obtained directly from registers while the channel is running. Because to calculate the remaining bytes, the initial NBYTES configured in DMA_TCDn_NBYTES_MLNO register is needed while the eDMA IP does not support getting it while a channel is active. In another word, the NBYTES value reading is always the actual (decrementing) NBYTES value the dma_engine is working with while a channel is running. Consequently, to get the remaining transfer bytes, a software-saved initial value of NBYTES (for example copied before enabling the channel) is needed. The formula to calculate it is shown below: RemainingBytes = RemainingMajorLoopCount * NBYTE-S(initially configured)

10.7.33 static uint32_t EDMA_GetErrorStatusFlags (DMA_Type * *base*) [inline], [static]

Parameters

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

Returns

The mask of error status flags. Users need to use the _edma_error_status_flags type to decode the return variables.

10.7.34 uint32_t EDMA_GetChannelStatusFlags (DMA_Type * *base*, uint32_t *channel*)

Parameters

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

Returns

The mask of channel status flags. Users need to use the `_edma_channel_status_flags` type to decode the return variables.

10.7.35 void EDMA_ClearChannelStatusFlags (DMA_Type * *base*, uint32_t *channel*, uint32_t *mask*)

Parameters

<i>base</i>	eDMA peripheral base address.
<i>channel</i>	eDMA channel number.
<i>mask</i>	The mask of channel status to be cleared. Users need to use the defined <code>_edma_channel_status_flags</code> type.

10.7.36 void EDMA_CreateHandle (edma_handle_t * *handle*, DMA_Type * *base*, uint32_t *channel*)

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

Parameters

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

10.7.37 void EDMA_InstallTCDMemory (edma_handle_t * *handle*, edma_tcd_t * *tcdPool*, uint32_t *tcdSize*)

This function is called after the EDMA_CreateHandle to use scatter/gather feature. This function shall only be used while users need to use scatter gather mode. Scatter gather mode enables EDMA to load a new transfer control block (tcd) in hardware, and automatically reconfigure that DMA channel for a

new transfer. Users need to prepare tcd memory and also configure tcds using interface EDMA_SubmitTransfer.

Parameters

<i>handle</i>	eDMA handle pointer.
<i>tcdPool</i>	A memory pool to store TCDs. It must be 32 bytes aligned.
<i>tcdSize</i>	The number of TCD slots.

10.7.38 void EDMA_SetCallback (*edma_handle_t * handle, edma_callback callback, void * userData*)

This callback is called in the eDMA IRQ handler. Use the callback to do something after the current major loop transfer completes. This function will be called every time one tcd finished transfer.

Parameters

<i>handle</i>	eDMA handle pointer.
<i>callback</i>	eDMA callback function pointer.
<i>userData</i>	A parameter for the callback function.

10.7.39 void EDMA_PrepTransferConfig (*edma_transfer_config_t * config, void * srcAddr, uint32_t srcWidth, int16_t srcOffset, void * destAddr, uint32_t destWidth, int16_t destOffset, uint32_t bytesEachRequest, uint32_t transferBytes*)

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

Parameters

<i>config</i>	The user configuration structure of type <i>edma_transfer_t</i> .
<i>srcAddr</i>	eDMA transfer source address.
<i>srcWidth</i>	eDMA transfer source address width(bytes).
<i>srcOffset</i>	source address offset.
<i>destAddr</i>	eDMA transfer destination address.
<i>destWidth</i>	eDMA transfer destination address width(bytes).
<i>destOffset</i>	destination address offset.
<i>bytesEachRequest</i>	eDMA transfer bytes per channel request.
<i>transferBytes</i>	eDMA transfer bytes to be transferred.

Note

The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

10.7.40 void EDMA_PrepTransfer (*edma_transfer_config_t * config, void * srcAddr, uint32_t srcWidth, void * destAddr, uint32_t destWidth, uint32_t bytesEachRequest, uint32_t transferBytes, edma_transfer_type_t transferType*)

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

Parameters

<i>config</i>	The user configuration structure of type <code>edma_transfer_t</code> .
<i>srcAddr</i>	eDMA transfer source address.
<i>srcWidth</i>	eDMA transfer source address width(bytes).
<i>destAddr</i>	eDMA transfer destination address.
<i>destWidth</i>	eDMA transfer destination address width(bytes).
<i>bytesEachRequest</i>	eDMA transfer bytes per channel request.
<i>transferBytes</i>	eDMA transfer bytes to be transferred.
<i>transferType</i>	eDMA transfer type.

Note

The data address and the data width must be consistent. For example, if the SRC is 4 bytes, the source address must be 4 bytes aligned, or it results in source address error (SAE).

10.7.41 status_t EDMA_SubmitTransfer (*edma_handle_t * handle, const edma_transfer_config_t * config*)

This function submits the eDMA transfer request according to the transfer configuration structure. In scatter gather mode, call this function will add a configured tcd to the circular list of tcd pool. The tcd pools is setup by call function `EDMA_InstallTCDMemory` before.

Parameters

<i>handle</i>	eDMA handle pointer.
<i>config</i>	Pointer to eDMA transfer configuration structure.

Return values

<i>kStatus_EDMA_Success</i>	It means submit transfer request succeed.
<i>kStatus_EDMA_Queue-Full</i>	It means TCD queue is full. Submit transfer request is not allowed.
<i>kStatus_EDMA_Busy</i>	It means the given channel is busy, need to submit request later.

10.7.42 void EDMA_StartTransfer (*edma_handle_t * handle*)

This function enables the channel request. Users can call this function after submitting the transfer request or before submitting the transfer request.

Parameters

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

10.7.43 void EDMA_StopTransfer (*edma_handle_t * handle*)

This function disables the channel request to pause the transfer. Users can call [EDMA_StartTransfer\(\)](#) again to resume the transfer.

Parameters

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

10.7.44 void EDMA_AbortTransfer (*edma_handle_t * handle*)

This function disables the channel request and clear transfer status bits. Users can submit another transfer after calling this API.

Parameters

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

10.7.45 static uint32_t EDMA_GetUnusedTCDNumber (*edma_handle_t * handle*) [**inline**], [**static**]

This function gets current tcd index which is run. If the TCD pool pointer is NULL, it will return 0.

Parameters

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

Returns

The unused tcd slot number.

10.7.46 static uint32_t EDMA_GetNextTCDAccount (*edma_handle_t * handle*) [**inline**], [**static**]

This function gets the next tcd address. If this is last TCD, return 0.

Parameters

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

Returns

The next TCD address.

10.7.47 void EDMA_HandleIRQ (*edma_handle_t * handle*)

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

Note: For the case using TCD queue, when the major iteration count is exhausted, additional operations are performed. These include the final address adjustments and reloading of the BITER field into the CITER. Assertion of an optional interrupt request also occurs at this time, as does a possible fetch of a new TCD from memory using the scatter/gather address pointer included in the descriptor (if scatter/gather is enabled).

For instance, when the time interrupt of TCD[0] happens, the TCD[1] has already been loaded into the eDMA engine. As sga and sga_index are calculated based on the DLAST_SGA bitfield lies in the TCD_CSR register, the sga_index in this case should be 2 (DLAST_SGA of TCD[1] stores the address of TCD[2]). Thus, the "tcdUsed" updated should be (tcdUsed - 2U) which indicates the number of TCDs can be loaded in the memory pool (because TCD[0] and TCD[1] have been loaded into the eDMA engine at this point already.).

For the last two continuous ISRs in a scatter/gather process, they both load the last TCD (The last ISR does not load a new TCD) from the memory pool to the eDMA engine when major loop completes. Therefore, ensure that the header and tcdUsed updated are identical for them. tcdUsed are both 0 in this case as no TCD to be loaded.

See the "eDMA basic data flow" in the eDMA Functional description section of the Reference Manual for further details.

Parameters

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

Chapter 11

EWM: External Watchdog Monitor Driver

11.1 Overview

The MCUXpresso SDK provides a peripheral driver for the External Watchdog (EWM) Driver module of MCUXpresso SDK devices.

11.2 Typical use case

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/ewm

Data Structures

- struct `ewm_config_t`
Describes EWM clock source. [More...](#)

Enumerations

- enum `_ewm_interrupt_enable_t` { `kEWM_InterruptEnable` = EWM_CTRL_INTEN_MASK }
EWM interrupt configuration structure with default settings all disabled.
- enum `_ewm_status_flags_t` { `kEWM_RunningFlag` = EWM_CTRL_EWMEN_MASK }
EWM status flags.

Driver version

- #define `FSL_EWM_DRIVER_VERSION` (MAKE_VERSION(2, 0, 3))
EWM driver version 2.0.3.

EWM initialization and de-initialization

- void `EWM_Init` (EWM_Type *base, const `ewm_config_t` *config)
Initializes the EWM peripheral.
- void `EWM_Deinit` (EWM_Type *base)
Deinitializes the EWM peripheral.
- void `EWM_GetDefaultConfig` (`ewm_config_t` *config)
Initializes the EWM configuration structure.

EWM functional Operation

- static void `EWM_EnableInterrupts` (EWM_Type *base, uint32_t mask)
Enables the EWM interrupt.
- static void `EWM_DisableInterrupts` (EWM_Type *base, uint32_t mask)
Disables the EWM interrupt.
- static uint32_t `EWM_GetStatusFlags` (EWM_Type *base)
Gets all status flags.

- void **EWM_Refresh** (EWM_Type *base)
Services the EWM.

11.3 Data Structure Documentation

11.3.1 struct ewm_config_t

Data structure for EWM configuration.

This structure is used to configure the EWM.

Data Fields

- bool **enableEwm**
Enable EWM module.
- bool **enableEwmInput**
Enable EWM_in input.
- bool **setInputAssertLogic**
EWM_in signal assertion state.
- bool **enableInterrupt**
Enable EWM interrupt.
- uint8_t **prescaler**
Clock prescaler value.
- uint8_t **compareLowValue**
Compare low-register value.
- uint8_t **compareHighValue**
Compare high-register value.

11.4 Macro Definition Documentation

11.4.1 #define FSL_EWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

11.5 Enumeration Type Documentation

11.5.1 enum _ewm_interrupt_enable_t

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

Enumerator

kEWM InterruptEnable Enable the EWM to generate an interrupt.

11.5.2 enum _ewm_status_flags_t

This structure contains the constants for the EWM status flags for use in the EWM functions.

Enumerator

kEWM_RunningFlag Running flag, set when EWM is enabled.

11.6 Function Documentation

11.6.1 void EWM_Init (EWM_Type * *base*, const ewm_config_t * *config*)

This function is used to initialize the EWM. After calling, the EWM runs immediately according to the configuration. Note that, except for the interrupt enable control bit, other control bits and registers are write once after a CPU reset. Modifying them more than once generates a bus transfer error.

This is an example.

```
*     ewm_config_t config;
*     EWM_GetDefaultConfig(&config);
*     config.compareHighValue = 0xAAU;
*     EWM_Init(ewm_base, &config);
*
```

Parameters

<i>base</i>	EWM peripheral base address
<i>config</i>	The configuration of the EWM

11.6.2 void EWM_Deinit (EWM_Type * *base*)

This function is used to shut down the EWM.

Parameters

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

11.6.3 void EWM_GetDefaultConfig (ewm_config_t * *config*)

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

```
*     ewmConfig->enableEwm = true;
*     ewmConfig->enableEwmInput = false;
*     ewmConfig->setInputAssertLogic = false;
*     ewmConfig->enableInterrupt = false;
*     ewmConfig->ewm_lpo_clock_source_t = kEWM_LpoClockSource0;
*     ewmConfig->prescaler = 0;
*     ewmConfig->compareLowValue = 0;
*     ewmConfig->compareHighValue = 0xFEU;
*
```

Parameters

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

See Also

[ewm_config_t](#)

11.6.4 static void EWM_EnableInterrupts (**EWM_Type** * *base*, **uint32_t** *mask*) [**inline**], [**static**]

This function enables the EWM interrupt.

Parameters

<i>base</i>	EWM peripheral base address
<i>mask</i>	The interrupts to enable The parameter can be combination of the following source if defined <ul style="list-style-type: none">• kEWM_InterruptEnable

11.6.5 static void EWM_DisableInterrupts (**EWM_Type** * *base*, **uint32_t** *mask*) [**inline**], [**static**]

This function enables the EWM interrupt.

Parameters

<i>base</i>	EWM peripheral base address
<i>mask</i>	The interrupts to disable The parameter can be combination of the following source if defined <ul style="list-style-type: none">• kEWM_InterruptEnable

11.6.6 static **uint32_t** EWM_GetStatusFlags (**EWM_Type** * *base*) [**inline**], [**static**]

This function gets all status flags.

This is an example for getting the running flag.

```
*     uint32_t status;
*     status = EWM_GetStatusFlags(ewm_base) & kEWM_RunningFlag;
*
```

Parameters

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

Returns

State of the status flag: asserted (true) or not-asserted (false).

See Also

[_ewm_status_flags_t](#)

- True: a related status flag has been set.
- False: a related status flag is not set.

11.6.7 void EWM_Refresh (EWM_Type * *base*)

This function resets the EWM counter to zero.

Parameters

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

Chapter 12

C90TFS Flash Driver

12.1 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.

Modules

- [Ftftx CACHE Driver](#)
- [Ftftx FLASH Driver](#)
- [Ftftx FLEXNVM Driver](#)
- [ftfx controller](#)
- [ftfx feature](#)

12.2 Ftftx FLASH Driver

12.2.1 Overview

Data Structures

- union `pflash_prot_status_t`
PFlash protection status. [More...](#)
- struct `flash_config_t`
Flash driver state information. [More...](#)

Enumerations

- enum `flash_prot_state_t` {

`kFLASH_ProtectionStateUnprotected`,

`kFLASH_ProtectionStateProtected`,

`kFLASH_ProtectionStateMixed` }

Enumeration for the three possible flash protection levels.
- enum `flash_property_tag_t` {

`kFLASH_PropertyPflash0SectorSize` = 0x00U,

`kFLASH_PropertyPflash0TotalSize` = 0x01U,

`kFLASH_PropertyPflash0BlockSize` = 0x02U,

`kFLASH_PropertyPflash0BlockCount` = 0x03U,

`kFLASH_PropertyPflash0BlockBaseAddr` = 0x04U,

`kFLASH_PropertyPflash0FacSupport` = 0x05U,

`kFLASH_PropertyPflash0AccessSegmentSize` = 0x06U,

`kFLASH_PropertyPflash0AccessSegmentCount` = 0x07U,

`kFLASH_PropertyPflash1SectorSize` = 0x10U,

`kFLASH_PropertyPflash1TotalSize` = 0x11U,

`kFLASH_PropertyPflash1BlockSize` = 0x12U,

`kFLASH_PropertyPflash1BlockCount` = 0x13U,

`kFLASH_PropertyPflash1BlockBaseAddr` = 0x14U,

`kFLASH_PropertyPflash1FacSupport` = 0x15U,

`kFLASH_PropertyPflash1AccessSegmentSize` = 0x16U,

`kFLASH_PropertyPflash1AccessSegmentCount` = 0x17U,

`kFLASH_PropertyFlexRamBlockBaseAddr` = 0x20U,

`kFLASH_PropertyFlexRamTotalSize` = 0x21U }

Enumeration for various flash properties.

Flash version

- #define `FSL_FLASH_DRIVER_VERSION` (`MAKE_VERSION(3U, 1U, 2U)`)

Flash driver version for SDK.
- #define `FSL_FLASH_DRIVER_VERSION_ROM` (`MAKE_VERSION(3U, 0U, 0U)`)

Flash driver version for ROM.

Initialization

- `status_t FLASH_Init (flash_config_t *config)`
Initializes the global flash properties structure members.

Erasing

- `status_t FLASH_Erase (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)`
Erases the Dflash sectors encompassed by parameters passed into function.
- `status_t FLASH_EraseSectorNonBlocking (flash_config_t *config, uint32_t start, uint32_t key)`
Erases the Dflash sectors encompassed by parameters passed into function.
- `status_t FLASH_EraseAll (flash_config_t *config, uint32_t key)`
Erases entire flexnvm.

Programming

- `status_t FLASH_Program (flash_config_t *config, uint32_t start, uint8_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, uint8_t *src, uint32_t lengthInBytes)`
Program the Program-Once-Field through parameters.

Reading

- `status_t FLASH_ReadResource (flash_config_t *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, ftx_read_resource_opt_t option)`
Reads the resource with data at locations passed in through parameters.
- `status_t FLASH_ReadOnce (flash_config_t *config, uint32_t index, uint8_t *dst, uint32_t lengthInBytes)`
Reads the Program Once Field through parameters.

Verification

- `status_t FLASH_VerifyErase (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, ftx_margin_value_t margin)`
Verifies an erasure of the desired flash area at a specified margin level.
- `status_t FLASH_VerifyEraseAll (flash_config_t *config, ftx_margin_value_t margin)`
Verifies erasure of the entire flash at a specified margin level.
- `status_t FLASH_VerifyProgram (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, ftx_margin_value_t margin, uint32_t *failedAddress, uint32_t *failedData)`
Verifies programming of the desired flash area at a specified margin level.

Security

- `status_t FLASH_GetSecurityState (flash_config_t *config, ftfx_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.

Protection

- `status_t FLASH_IsProtected (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_prot_state_t *protection_state)`
Returns the protection state of the desired flash area via the pointer passed into the function.
- `status_t FLASH_PflashSetProtection (flash_config_t *config, pflash_prot_status_t *protectStatus)`
Sets the PFlash Protection to the intended protection status.
- `status_t FLASH_PflashGetProtection (flash_config_t *config, pflash_prot_status_t *protectStatus)`
Gets the PFlash protection status.

Properties

- `status_t FLASHGetProperty (flash_config_t *config, flash_property_tag_t whichProperty, uint32_t *value)`
Returns the desired flash property.

commandStatus

- `status_t FLASH_GetCommandState (void)`
Get previous command status.

12.2.2 Data Structure Documentation

12.2.2.1 union pflash_prot_status_t

Data Fields

- `uint32_t protl`
PROT[31:0].
- `uint32_t proth`
PROT[63:32].
- `uint8_t protsl`
PROTS[7:0].
- `uint8_t protsh`
PROTS[15:8].

Field Documentation

- (1) `uint32_t pflash_prot_status_t::protl`
- (2) `uint32_t pflash_prot_status_t::proth`
- (3) `uint8_t pflash_prot_status_t::protsl`
- (4) `uint8_t pflash_prot_status_t::protsh`

12.2.2.2 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.

12.2.3 Macro Definition Documentation

12.2.3.1 #define FSL_FLASH_DRIVER_VERSION (MAKE_VERSION(3U, 1U, 2U))

Version 3.1.2.

12.2.3.2 #define FSL_FLASH_DRIVER_VERSION_ROM (MAKE_VERSION(3U, 0U, 0U))

Version 3.0.0.

12.2.4 Enumeration Type Documentation

12.2.4.1 enum flash_prot_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.

12.2.4.2 enum flash_property_tag_t

Enumerator

kFLASH_PropertyPflash0SectorSize Pflash sector size property.

kFLASH_PropertyPflash0TotalSize Pflash total size property.

kFLASH_PropertyPflash0BlockSize Pflash block size property.

kFLASH_PropertyPflash0BlockCount Pflash block count property.

kFLASH_PropertyPflash0BlockBaseAddr Pflash block base address property.
kFLASH_PropertyPflash0FacSupport Pflash fac support property.
kFLASH_PropertyPflash0AccessSegmentSize Pflash access segment size property.
kFLASH_PropertyPflash0AccessSegmentCount Pflash access segment count property.
kFLASH_PropertyPflash1SectorSize Pflash sector size property.
kFLASH_PropertyPflash1TotalSize Pflash total size property.
kFLASH_PropertyPflash1BlockSize Pflash block size property.
kFLASH_PropertyPflash1BlockCount Pflash block count property.
kFLASH_PropertyPflash1BlockBaseAddr Pflash block base address property.
kFLASH_PropertyPflash1FacSupport Pflash fac support property.
kFLASH_PropertyPflash1AccessSegmentSize Pflash access segment size property.
kFLASH_PropertyPflash1AccessSegmentCount Pflash access segment count property.
kFLASH_PropertyFlexRamBlockBaseAddr FlexRam block base address property.
kFLASH_PropertyFlexRamTotalSize FlexRam total size property.

12.2.5 Function Documentation

12.2.5.1 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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Partition-StatusUpdateFailure</i>	Failed to update the partition status.

12.2.5.2 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_FTFx_Success</i>	API was executed successfully; the appropriate number of flash sectors based on the desired start address and length were erased successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_AddressError</i>	The address is out of range.
<i>kStatus_FTFx_EraseKeyError</i>	The API erase key is invalid.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.2.5.3 status_t FLASH_EraseSectorNonBlocking (**flash_config_t * config, uint32_t start, uint32_t key**)

This function erases one flash sector size based on the start address, and it is executed asynchronously.

NOTE: This function can only erase one flash sector at a time, and the other commands can be executed after the previous command has been completed.

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>key</i>	The value used to validate all flash erase APIs.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_-AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_Address-Error</i>	The address is out of range.
<i>kStatus_FTFx_EraseKey-Error</i>	The API erase key is invalid.

12.2.5.4 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_FTFx_Success</i>	API was executed successfully; the all pflash and flexnvm were erased successfully, the swap and eeprom have been reset to unconfigured state.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_EraseKey-Error</i>	API erase key is invalid.

<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx-CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FTFx_Partition-StatusUpdateFailure</i>	Failed to update the partition status.

12.2.5.5 **status_t FLASH_Program (flash_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes)**

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the flash.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; the desired data were programmed successfully into flash based on desired start address and length.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx-AlignmentError</i>	Parameter is not aligned with the specified baseline.

<i>kStatus_FTFx_Address_Error</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_-CommandFailure</i>	Run-time error during the command execution.

12.2.5.6 `status_t FLASH_ProgramOnce (flash_config_t * config, uint32_t index, uint8_t * src, uint32_t lengthInBytes)`

This function Program the Program-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>src</i>	A pointer to the source buffer of data that is used to store data to be write.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; The index indicating the area of program once field was programmed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.

<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.2.5.7 status_t FLASH_ReadResource (*flash_config_t * config, uint32_t start, uint8_t * dst, uint32_t lengthInBytes, ftfx_read_resource_opt_t option*)

This function reads the flash memory with the desired location for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>dst</i>	A pointer to the destination buffer of data that is used to store data to be read.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be read. Must be word-aligned.
<i>option</i>	The resource option which indicates which area should be read back.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; the data have been read successfully from program flash IFR, data flash IFR space, and the Version ID field.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.

<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.2.5.8 status_t FLASH_ReadOnce (*flash_config_t * config, uint32_t index, uint8_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_FTFx_Success</i>	API was executed successfully; the data have been successfully read form Program flash0 IFR map and Program Once field based on index and length.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.2.5.9 status_t FLASH_VerifyErase (*flash_config_t * config, uint32_t start, uint32_t lengthInBytes, ftfx_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_FTFx_Success</i>	API was executed successfully; the specified FLASH region has been erased.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.2.5.10 status_t FLASH_VerifyEraseAll (flash_config_t * *config*, ftfx_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_FTFx_Success</i>	API was executed successfully; all program flash and flexnvm were in erased state.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.2.5.11 **status_t FLASH_VerifyProgram (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, const uint8_t * *expectedData*, ftfx_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_FTFx_Success</i>	API was executed successfully; the desired data have been successfully programmed into specified FLASH region.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.2.5.12 status_t FLASH_GetSecurityState (flash_config_t * *config*, ftfx_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_FTFx_Success</i>	API was executed successfully; the security state of flash was stored to state.
-----------------------------	---

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

12.2.5.13 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.

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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_-CommandFailure</i>	Run-time error during the command execution.

12.2.5.14 status_t **FLASH_IsProtected** (*flash_config_t * config, uint32_t start, uint32_t lengthInBytes, flash_prot_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_FTFx_Success</i>	API was executed successfully; the protection state of specified FLASH region was stored to protection_state.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FTFx_AddressError</i>	The address is out of range.

12.2.5.15 status_t FLASH_PflashSetProtection (*flash_config_t * config*, *pflash_prot_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_FTFx_Success</i>	API was executed successfully; the specified FLASH region is protected.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during command execution.

12.2.5.16 **status_t FLASH_PflashGetProtection (flash_config_t * *config*,
pflash_prot_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_FTFx_Success</i>	API was executed successfully; the Protection state was stored to protect-Status;
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.

12.2.5.17 status_t FLASH_GetProperty (flash_config_t * *config*, flash_property_tag_t *whichProperty*, uint32_t * *value*)

Parameters

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

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; the flash property was stored to value.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_Unknown-Property</i>	An unknown property tag.

12.2.5.18 status_t FLASH_GetCommandState (void)

This function is used to obtain the execution status of the previous command.

Return values

<i>kStatus_FTFx_Success</i>	The previous command is executed successfully.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_-CommandFailure</i>	Run-time error during the command execution.

12.3 Fftfx CACHE Driver

12.3.1 Overview

Data Structures

- struct `fftfx_prefetch_speculation_status_t`
FTFx prefetch speculation status. [More...](#)
- struct `fftfx_cache_config_t`
FTFx cache driver state information. [More...](#)

Enumerations

- enum `_fftfx_cache_ram_func_constants` { `kFTFx_CACHE_RamFuncMaxSizeInWords` = 16U }
Constants for execute-in-RAM flash function.

Functions

- `status_t FTFx_CACHE_Init (fftfx_cache_config_t *config)`
Initializes the global FTFx cache structure members.
- `status_t FTFx_CACHE_ClearCachePrefetchSpeculation (fftfx_cache_config_t *config, bool isPreProcess)`
Process the cache/prefetch/speculation to the flash.
- `status_t FTFx_CACHE_PflashSetPrefetchSpeculation (fftfx_prefetch_speculation_status_t *speculationStatus)`
Sets the PFlash prefetch speculation to the intended speculation status.
- `status_t FTFx_CACHE_PflashGetPrefetchSpeculation (fftfx_prefetch_speculation_status_t *speculationStatus)`
Gets the PFlash prefetch speculation status.

12.3.2 Data Structure Documentation

12.3.2.1 struct fftfx_prefetch_speculation_status_t

Data Fields

- `bool instructionOff`
Instruction speculation.
- `bool dataOff`
Data speculation.

Field Documentation

- (1) `bool fftfx_prefetch_speculation_status_t::instructionOff`

(2) `bool fftfx_prefetch_speculation_status_t::dataOff`

12.3.2.2 struct fftfx_cache_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

- `uint8_t flashMemoryIndex`
0 - primary flash; 1 - secondary flash
- `function_bit_operation_ptr_t bitOperFuncAddr`
An buffer point to the flash execute-in-RAM function.

Field Documentation

(1) `function_bit_operation_ptr_t fftfx_cache_config_t::bitOperFuncAddr`

12.3.3 Enumeration Type Documentation

12.3.3.1 enum _fftfx_cache_ram_func_constants

Enumerator

`kFTFx_CACHE_RamFuncMaxSizeInWords` The maximum size of execute-in-RAM function.

12.3.4 Function Documentation

12.3.4.1 status_t FTFx_CACHE_Init(fftfx_cache_config_t * config)

This function checks and initializes the Flash module for the other FTFx cache APIs.

Parameters

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

Return values

<code>kStatus_FTFx_Success</code>	API was executed successfully.
<code>kStatus_FTFx_Invalid-Argument</code>	An invalid argument is provided.

<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
---	---

12.3.4.2 status_t FTFx_CACHE_ClearCachePrefetchSpeculation (*ftfx_cache_config_t * config, bool isPreProcess*)

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>isPreProcess</i>	The possible option used to control flash cache/prefetch/speculation

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	Invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.

12.3.4.3 status_t FTFx_CACHE_PflashSetPrefetchSpeculation (*ftfx_prefetch_speculation_status_t * speculationStatus*)

Parameters

<i>speculation-Status</i>	The expected protect status to set to the PFlash protection register. Each bit is
---------------------------	---

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-SpeculationOption</i>	An invalid speculation option argument is provided.

12.3.4.4 status_t FTFx_CACHE_PflashGetPrefetchSpeculation (*ftfx_prefetch_speculation_status_t * speculationStatus*)

Parameters

<i>speculation- Status</i>	Speculation status returned by the PFlash IP.
--------------------------------	---

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
-----------------------------	--------------------------------

12.4 Ftftx FLEXNVM Driver

12.4.1 Overview

Data Structures

- struct `flexnvm_config_t`
Flexnvm driver state information. [More...](#)

Enumerations

- enum `flexnvm_property_tag_t` {

`kFLEXNVM_PropertyDflashSectorSize` = 0x00U,
`kFLEXNVM_PropertyDflashTotalSize` = 0x01U,
`kFLEXNVM_PropertyDflashBlockSize` = 0x02U,
`kFLEXNVM_PropertyDflashBlockCount` = 0x03U,
`kFLEXNVM_PropertyDflashBlockBaseAddr` = 0x04U,
`kFLEXNVM_PropertyAliasDflashBlockBaseAddr` = 0x05U,
`kFLEXNVM_PropertyFlexRamBlockBaseAddr` = 0x06U,
`kFLEXNVM_PropertyFlexRamTotalSize` = 0x07U,
`kFLEXNVM_PropertyEepromTotalSize` = 0x08U }

Enumeration for various flexnvm properties.

Functions

- `status_t FLEXNVM_EepromWrite (flexnvm_config_t *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)`
Programs the EEPROM with data at locations passed in through parameters.

Initialization

- `status_t FLEXNVM_Init (flexnvm_config_t *config)`
Initializes the global flash properties structure members.

Erasing

- `status_t FLEXNVM_DflashErase (flexnvm_config_t *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)`
Erases the Dflash sectors encompassed by parameters passed into function.
- `status_t FLEXNVM_EraseAll (flexnvm_config_t *config, uint32_t key)`
Erases entire flexnvm.

Programming

- **status_t FLEXNVM_DflashProgram (flexnvm_config_t *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)**
Programs flash with data at locations passed in through parameters.
- **status_t FLEXNVM_ProgramPartition (flexnvm_config_t *config, ftx_partition_flexram_load_opt_t option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode)**
Prepares the FlexNVM block for use as data flash, EEPROM backup, or a combination of both and initializes the FlexRAM.

Reading

- **status_t FLEXNVM_ReadResource (flexnvm_config_t *config, uint32_t start, uint8_t *dst, uint32_t lengthInBytes, ftx_read_resource_opt_t option)**
Reads the resource with data at locations passed in through parameters.

Verification

- **status_t FLEXNVM_DflashVerifyErase (flexnvm_config_t *config, uint32_t start, uint32_t lengthInBytes, ftx_margin_value_t margin)**
Verifies an erasure of the desired flash area at a specified margin level.
- **status_t FLEXNVM_VerifyEraseAll (flexnvm_config_t *config, ftx_margin_value_t margin)**
Verifies erasure of the entire flash at a specified margin level.
- **status_t FLEXNVM_DflashVerifyProgram (flexnvm_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, ftx_margin_value_t margin, uint32_t *failedAddress, uint32_t *failedData)**
Verifies programming of the desired flash area at a specified margin level.

Security

- **status_t FLEXNVM_GetSecurityState (flexnvm_config_t *config, ftx_security_state_t *state)**
Returns the security state via the pointer passed into the function.
- **status_t FLEXNVM_SecurityBypass (flexnvm_config_t *config, const uint8_t *backdoorKey)**
Allows users to bypass security with a backdoor key.

Flash Protection Utilities

- **status_t FLEXNVM_DflashSetProtection (flexnvm_config_t *config, uint8_t protectStatus)**
Sets the DFlash protection to the intended protection status.
- **status_t FLEXNVM_DflashGetProtection (flexnvm_config_t *config, uint8_t *protectStatus)**
Gets the DFlash protection status.
- **status_t FLEXNVM_EepromSetProtection (flexnvm_config_t *config, uint8_t protectStatus)**
Sets the EEPROM protection to the intended protection status.
- **status_t FLEXNVM_EepromGetProtection (flexnvm_config_t *config, uint8_t *protectStatus)**

Gets the EEPROM protection status.

Properties

- `status_t FLEXNVMGetProperty (flexnvm_config_t *config, flexnvm_property_tag_t whichProperty, uint32_t *value)`
Returns the desired flexnvm property.

12.4.2 Data Structure Documentation

12.4.2.1 struct flexnvm_config_t

An instance of this structure is allocated by the user of the Flexnvm driver and passed into each of the driver APIs.

12.4.3 Enumeration Type Documentation

12.4.3.1 enum flexnvm_property_tag_t

Enumerator

- `kFLEXNVM_PropertyDflashSectorSize` Dflash sector size property.
- `kFLEXNVM_PropertyDflashTotalSize` Dflash total size property.
- `kFLEXNVM_PropertyDflashBlockSize` Dflash block size property.
- `kFLEXNVM_PropertyDflashBlockCount` Dflash block count property.
- `kFLEXNVM_PropertyDflashBlockBaseAddr` Dflash block base address property.
- `kFLEXNVM_PropertyAliasDflashBlockBaseAddr` Dflash block base address Alias property.
- `kFLEXNVM_PropertyFlexRamBlockBaseAddr` FlexRam block base address property.
- `kFLEXNVM_PropertyFlexRamTotalSize` FlexRam total size property.
- `kFLEXNVM_PropertyEepromTotalSize` EEPROM total size property.

12.4.4 Function Documentation

12.4.4.1 status_t FLEXNVM_Init (`flexnvm_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

12.4.4.2 status_t FLEXNVM_DflashErase (*flexnvm_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_FTFx_Success</i>	API was executed successfully; the appropriate number of date flash sectors based on the desired start address and length were erased successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.

<i>kStatus_FTFx_AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_AddressError</i>	The address is out of range.
<i>kStatus_FTFx_EraseKeyError</i>	The API erase key is invalid.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.4.4.3 status_t FLEXNVM_EraseAll (*flexnvm_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_FTFx_Success</i>	API was executed successfully; the entire flexnvm has been erased successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_EraseKeyError</i>	API erase key is invalid.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.

<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FTFx_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

12.4.4.4 **status_t FLEXNVM_DflashProgram (flexnvm_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes)**

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the flash.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; the desired date have been successfully programed into specified date flash region.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.

<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx-CommandFailure</i>	Run-time error during the command execution.

12.4.4.5 status_t FLEXNVM_ProgramPartition (*flexnvm_config_t * config, ftfx_partition_flexram_load_opt_t option, uint32_t eepromDataSizeCode, uint32_t flexnvmPartitionCode*)

Parameters

<i>config</i>	Pointer to storage for the driver runtime state.
<i>option</i>	The option used to set FlexRAM load behavior during reset.
<i>eepromData-SizeCode</i>	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
<i>flexnvm-PartitionCode</i>	Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; the FlexNVM block for use as data flash, EEPROM backup, or a combination of both have been Prepared.
<i>kStatus_FTFx_Invalid-Argument</i>	Invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

<i>kStatus_FTFx_CommandFailure</i>	Run-time error during command execution.
------------------------------------	--

12.4.4.6 status_t FLEXNVM_ReadResource (*flexnvm_config_t * config*, *uint32_t start*, *uint8_t * dst*, *uint32_t lengthInBytes*, *ftfx_read_resource_opt_t option*)

This function reads the flash memory with the desired location for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>dst</i>	A pointer to the destination buffer of data that is used to store data to be read.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be read. Must be word-aligned.
<i>option</i>	The resource option which indicates which area should be read back.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; the data have been read successfully from program flash IFR, data flash IFR space, and the Version ID field
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_-AlignmentError</i>	Parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.
------------------------------------	--

12.4.4.7 status_t FLEXNVM_DflashVerifyErase (flexnvm_config_t * config, uint32_t start, uint32_t lengthInBytes, ftfx_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_FTFx_Success</i>	API was executed successfully; the specified data flash region is in erased state.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.
------------------------------------	--

12.4.4.8 status_t FLEXNVM_VerifyEraseAll (*flexnvm_config_t * config, ftfx_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_FTFx_Success</i>	API was executed successfully; the entire flexnvm region is in erased state.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.4.4.9 status_t FLEXNVM_DflashVerifyProgram (*flexnvm_config_t * config, uint32_t start, uint32_t lengthInBytes, const uint8_t * expectedData, ftfx_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_FTFx_Success</i>	API was executed successfully; the desired data have been programmed successfully into specified data flash region.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.4.4.10 status_t FLEXNVM_GetSecurityState (**flexnvm_config_t * config,** **ftfx_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_FTFx_Success</i>	API was executed successfully; the security state of flexnvm was stored to state.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.

12.4.4.11 **status_t FLEXNVM_SecurityBypass (flexnvm_config_t * *config*, const uint8_t * *backdoorKey*)**

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>backdoorKey</i>	A pointer to the user buffer containing the backdoor key.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.
------------------------------------	--

12.4.4.12 status_t FLEXNVM_EepromWrite (*flexnvm_config_t * config, uint32_t start, uint8_t * src, uint32_t lengthInBytes*)

This function programs the emulated EEPROM with the desired data for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<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_FTFx_Success</i>	API was executed successfully; the desired data have been successfully programmed into specified eeprom region.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_SetFlexramAsEepromError</i>	Failed to set flexram as eeprom.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_RecoverFlexramAsRamError</i>	Failed to recover the FlexRAM as RAM.

12.4.4.13 status_t FLEXNVM_DflashSetProtection (*flexnvm_config_t * config, uint8_t protectStatus*)

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the DFlash protection register. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully; the specified DFlash region is protected.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_CommandNotSupported</i>	Flash API is not supported.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during command execution.

12.4.4.14 status_t FLEXNVM_DflashGetProtection (*flexnvm_config_t * config, uint8_t * protectStatus*)

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total DFlash. The least significant bit corresponds to the lowest address area of the DFlash. The most significant bit corresponds to the highest address area of the DFlash, and so on. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.

<i>kStatus_FTFx_CommandNotSupported</i>	Flash API is not supported.
---	-----------------------------

12.4.4.15 status_t FLEXNVM_EepromSetProtection (**flexnvm_config_t * config, uint8_t protectStatus**)

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the EEPROM protection register. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of EEPROM, and so on. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_CommandNotSupported</i>	Flash API is not supported.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during command execution.

12.4.4.16 status_t FLEXNVM_EepromGetProtection (**flexnvm_config_t * config, uint8_t * protectStatus**)

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	DFlash Protect status returned by the PFlash IP. Each bit corresponds to the protection of the 1/8 of the total EEPROM. The least significant bit corresponds to the lowest address area of the EEPROM. The most significant bit corresponds to the highest address area of the EEPROM. There are two possible cases as below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_CommandNotSupported</i>	Flash API is not supported.

12.4.4.17 status_t FLEXNVMGetProperty (*flexnvm_config_t * config*, *flexnvm_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 flexnvm_property_tag_t
<i>value</i>	A pointer to the value returned for the desired flexnvm property.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_UnknownProperty</i>	An unknown property tag.

12.5 fffx feature

12.5.1 Overview

Modules

- fffx adapter

Macros

- #define **FTFx_DRIVER_HAS_FLASH1_SUPPORT** (0U)
Indicates whether the secondary flash is supported in the Flash driver.

FTFx configuration

- #define **FTFx_DRIVER_IS_FLASH_RESIDENT** 1U
Flash driver location.
- #define **FTFx_DRIVER_IS_EXPORTED** 0U
Flash Driver Export option.

Secondary flash configuration

- #define **FTFx_FLASH1_HAS_PROT_CONTROL** (0U)
Indicates whether the secondary flash has its own protection register in flash module.
- #define **FTFx_FLASH1_HAS_XACC_CONTROL** (0U)
Indicates whether the secondary flash has its own Execute-Only access register in flash module.

12.5.2 Macro Definition Documentation

12.5.2.1 #define FTFx_DRIVER_IS_FLASH_RESIDENT 1U

Used for the flash resident application.

12.5.2.2 #define FTFx_DRIVER_IS_EXPORTED 0U

Used for the MCUXpresso SDK application.

12.5.2.3 #define FTFx_FLASH1_HAS_PROT_CONTROL (0U)

12.5.2.4 #define FTFx_FLASH1_HAS_XACC_CONTROL (0U)

12.5.3 ftx adapter

12.6 ftx controller

12.6.1 Overview

Modules

- [ftfx utilities](#)

Data Structures

- struct [ftfx_spec_mem_t](#)
ftfx special memory access information. [More...](#)
- struct [ftfx_mem_desc_t](#)
Flash memory descriptor. [More...](#)
- struct [ftfx_ops_config_t](#)
Active FTFx information for the current operation. [More...](#)
- struct [ftfx_ifr_desc_t](#)
Flash IFR memory descriptor. [More...](#)
- struct [ftfx_config_t](#)
Flash driver state information. [More...](#)

Enumerations

- enum [ftfx_partition_flexram_load_opt_t](#) {

kFTFx_PartitionFlexramLoadOptLoadedWithValidEepromData,

kFTFx_PartitionFlexramLoadOptNotLoaded = 0x01U }

Enumeration for the FlexRAM load during reset option.
- enum [ftfx_read_resource_opt_t](#) {

kFTFx_ResourceOptionFlashIfr,

kFTFx_ResourceOptionVersionId = 0x01U }

Enumeration for the two possible options of flash read resource command.
- enum [ftfx_margin_value_t](#) {

kFTFx_MarginValueNormal,

kFTFx_MarginValueUser,

kFTFx_MarginValueFactory,

kFTFx_MarginValueInvalid }

Enumeration for supported FTFx margin levels.
- enum [ftfx_security_state_t](#) {

kFTFx_SecurityStateNotSecure = (int)0xc33cc33cu,

kFTFx_SecurityStateBackdoorEnabled = (int)0x5aa55aa5u,

kFTFx_SecurityStateBackdoorDisabled = (int)0x5ac33ca5u }

Enumeration for the three possible FTFx security states.
- enum [ftfx_flexram_func_opt_t](#) {

kFTFx_FlexramFuncOptAvailableAsRam = 0xFFU,

kFTFx_FlexramFuncOptAvailableForEeprom = 0x00U }

Enumeration for the two possilbe options of set FlexRAM function command.

- enum `_flash_acceleration_ram_property`
Enumeration for acceleration ram property.
- enum `ftfx_swap_state_t` {

`kFTFx_SwapStateUninitialized` = 0x00U,
`kFTFx_SwapStateReady` = 0x01U,
`kFTFx_SwapStateUpdate` = 0x02U,
`kFTFx_SwapStateUpdateErased` = 0x03U,
`kFTFx_SwapStateComplete` = 0x04U,
`kFTFx_SwapStateDisabled` = 0x05U }
- Enumeration for the possible flash Swap status.*
- enum `_ftfx_memory_type`
Enumeration for FTFx memory type.

FTFx status

- enum {

`kStatus_FTFx_Success` = MAKE_STATUS(kStatusGroupGeneric, 0),
`kStatus_FTFx_InvalidArgument` = MAKE_STATUS(kStatusGroupGeneric, 4),
`kStatus_FTFx_SizeError` = MAKE_STATUS(kStatusGroupFtxDriver, 0),
`kStatus_FTFx_AlignmentError`,
`kStatus_FTFx_AddressError` = MAKE_STATUS(kStatusGroupFtxDriver, 2),
`kStatus_FTFx_AccessError`,
`kStatus_FTFx_ProtectionViolation`,
`kStatus_FTFx_CommandFailure`,
`kStatus_FTFx_UnknownProperty` = MAKE_STATUS(kStatusGroupFtxDriver, 6),
`kStatus_FTFx_EraseKeyError` = MAKE_STATUS(kStatusGroupFtxDriver, 7),
`kStatus_FTFx_RegionExecuteOnly` = MAKE_STATUS(kStatusGroupFtxDriver, 8),
`kStatus_FTFx_ExecuteInRamFunctionNotReady`,
`kStatus_FTFx_PartitionStatusUpdateFailure`,
`kStatus_FTFx_SetFlexramAsEepromError`,
`kStatus_FTFx_RecoverFlexramAsRamError`,
`kStatus_FTFx_SetFlexramAsRamError` = MAKE_STATUS(kStatusGroupFtxDriver, 13),
`kStatus_FTFx_RecoverFlexramAsEepromError`,
`kStatus_FTFx_CommandNotSupported` = MAKE_STATUS(kStatusGroupFtxDriver, 15),
`kStatus_FTFx_SwapSystemNotInUninitialized`,
`kStatus_FTFx_SwapIndicatorAddressError`,
`kStatus_FTFx_ReadOnlyProperty` = MAKE_STATUS(kStatusGroupFtxDriver, 18),
`kStatus_FTFx_InvalidPropertyValue`,
`kStatus_FTFx_InvalidSpeculationOption`,
`kStatus_FTFx_CommandOperationInProgress` }
- FTFx driver status codes.*
- #define `kStatusGroupGeneric` 0
FTFx driver status group.
- #define `kStatusGroupFtxDriver` 1

FTFx API key

- enum `_ftfx_driver_api_keys` { `kFTFx_ApiEraseKey` = FOUR_CHAR_CODE('k', 'f', 'e', 'k') }
- Enumeration for FTFx driver API keys.*

Initialization

- void `FTFx_API_Init` (`ftfx_config_t` *config)
- Initializes the global flash properties structure members.*

Erasing

- `status_t FTFx_CMD_Erase` (`ftfx_config_t` *config, `uint32_t` start, `uint32_t` lengthInBytes, `uint32_t` key)

Erases the flash sectors encompassed by parameters passed into function.

- `status_t FTFx_CMD_EraseSectorNonBlocking` (`ftfx_config_t` *config, `uint32_t` start, `uint32_t` key)

Erases the flash sectors encompassed by parameters passed into function.

- `status_t FTFx_CMD_EraseAll` (`ftfx_config_t` *config, `uint32_t` key)

Erases entire flash.

- `status_t FTFx_CMD_EraseAllExecuteOnlySegments` (`ftfx_config_t` *config, `uint32_t` key)

Erases all program flash execute-only segments defined by the FXACC registers.

Programming

- `status_t FTFx_CMD_Program` (`ftfx_config_t` *config, `uint32_t` start, `const uint8_t` *src, `uint32_t` lengthInBytes)

Programs flash with data at locations passed in through parameters.

- `status_t FTFx_CMD_ProgramOnce` (`ftfx_config_t` *config, `uint32_t` index, `const uint8_t` *src, `uint32_t` lengthInBytes)

Programs Program Once Field through parameters.

Reading

- `status_t FTFx_CMD_ReadOnce` (`ftfx_config_t` *config, `uint32_t` index, `uint8_t` *dst, `uint32_t` lengthInBytes)

Reads the Program Once Field through parameters.

- `status_t FTFx_CMD_ReadResource` (`ftfx_config_t` *config, `uint32_t` start, `uint8_t` *dst, `uint32_t` lengthInBytes, `ftfx_read_resource_opt_t` option)

Reads the resource with data at locations passed in through parameters.

Verification

- `status_t FTFx_CMD_VerifyErase (ftfx_config_t *config, uint32_t start, uint32_t lengthInBytes, ftx_margin_value_t margin)`
Verifies an erasure of the desired flash area at a specified margin level.
- `status_t FTFx_CMD_VerifyEraseAll (ftfx_config_t *config, ftx_margin_value_t margin)`
Verifies erasure of the entire flash at a specified margin level.
- `status_t FTFx_CMD_VerifyEraseAllExecuteOnlySegments (ftfx_config_t *config, ftx_margin_value_t margin)`
Verifies whether the program flash execute-only segments have been erased to the specified read margin level.
- `status_t FTFx_CMD_VerifyProgram (ftfx_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, ftx_margin_value_t margin, uint32_t *failedAddress, uint32_t *failedData)`
Verifies programming of the desired flash area at a specified margin level.

Security

- `status_t FTFx_REG_GetSecurityState (ftfx_config_t *config, ftx_security_state_t *state)`
Returns the security state via the pointer passed into the function.
- `status_t FTFx_CMD_SecurityBypass (ftfx_config_t *config, const uint8_t *backdoorKey)`
Allows users to bypass security with a backdoor key.

12.6.2 Data Structure Documentation

12.6.2.1 struct ftx_spec_mem_t

Data Fields

- `uint32_t base`
Base address of flash special memory.
- `uint32_t size`
size of flash special memory.
- `uint32_t count`
flash special memory count.

Field Documentation

- (1) `uint32_t ftx_spec_mem_t::base`
- (2) `uint32_t ftx_spec_mem_t::size`
- (3) `uint32_t ftx_spec_mem_t::count`

12.6.2.2 struct ftfx_mem_desc_t

Data Fields

- `uint32_t blockBase`
A base address of the flash block.
- `uint32_t totalSize`
The size of the flash block.
- `uint32_t sectorSize`
The size in bytes of a sector of flash.
- `uint32_t blockCount`
A number of flash blocks.
- `uint8_t type`
Type of flash block.
- `uint8_t index`
Index of flash block.

Field Documentation

- (1) `uint8_t ftfx_mem_desc_t::type`
- (2) `uint8_t ftfx_mem_desc_t::index`
- (3) `uint32_t ftfx_mem_desc_t::totalSize`
- (4) `uint32_t ftfx_mem_desc_t::sectorSize`
- (5) `uint32_t ftfx_mem_desc_t::blockCount`

12.6.2.3 struct ftfx_ops_config_t

Data Fields

- `uint32_t convertedAddress`
A converted address for the current flash type.

Field Documentation

- (1) `uint32_t ftfx_ops_config_t::convertedAddress`

12.6.2.4 struct ftfx_ifr_desc_t

12.6.2.5 struct ftfx_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 flexramBlockBase`
The base address of the FlexRAM/acceleration RAM.
- `uint32_t flexramTotalSize`
The size of the FlexRAM/acceleration RAM.
- `uint16_t eepromTotalSize`
The size of EEPROM area which was partitioned from FlexRAM.
- `function_ptr_t runCmdFuncAddr`
An buffer point to the flash execute-in-RAM function.

Field Documentation

(1) `function_ptr_t ftfx_config_t::runCmdFuncAddr`

12.6.3 Macro Definition Documentation

12.6.3.1 #define kStatusGroupGeneric 0

12.6.4 Enumeration Type Documentation

12.6.4.1 anonymous enum

Enumerator

`kStatus_FTFx_Success` API is executed successfully.

`kStatus_FTFx_InvalidArgument` Invalid argument.

`kStatus_FTFx_SizeError` Error size.

`kStatus_FTFx_AlignmentError` Parameter is not aligned with the specified baseline.

`kStatus_FTFx_AddressError` Address is out of range.

`kStatus_FTFx_AccessError` Invalid instruction codes and out-of bound addresses.

`kStatus_FTFx_ProtectionViolation` The program/erase operation is requested to execute on protected areas.

`kStatus_FTFx_CommandFailure` Run-time error during command execution.

`kStatus_FTFx_UnknownProperty` Unknown property.

`kStatus_FTFx_EraseKeyError` API erase key is invalid.

`kStatus_FTFx_RegionExecuteOnly` The current region is execute-only.

`kStatus_FTFx_ExecuteInRamFunctionNotReady` Execute-in-RAM function is not available.

`kStatus_FTFx_PartitionStatusUpdateFailure` Failed to update partition status.

`kStatus_FTFx_SetFlexramAsEepromError` Failed to set FlexRAM as EEPROM.

`kStatus_FTFx_RecoverFlexramAsRamError` Failed to recover FlexRAM as RAM.

`kStatus_FTFx_SetFlexramAsRamError` Failed to set FlexRAM as RAM.

`kStatus_FTFx_RecoverFlexramAsEepromError` Failed to recover FlexRAM as EEPROM.

`kStatus_FTFx_CommandNotSupported` Flash API is not supported.

`kStatus_FTFx_SwapSystemNotInUninitialized` Swap system is not in an uninitialized state.

`kStatus_FTFx_SwapIndicatorAddressError` The swap indicator address is invalid.

`kStatus_FTFx_ReadOnlyProperty` The flash property is read-only.

kStatus_FTFx_InvalidPropertyValue The flash property value is out of range.

kStatus_FTFx_InvalidSpeculationOption The option of flash prefetch speculation is invalid.

kStatus_FTFx_CommandOperationInProgress The option of flash command is processing.

12.6.4.2 enum _ftfx_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

kFTFx_ApiEraseKey Key value used to validate all FTFx erase APIs.

12.6.4.3 enum ftfx_partition_flexram_load_opt_t

Enumerator

kFTFx_PartitionFlexramLoadOptLoadedWithValidEepromData FlexRAM is loaded with valid EEPROM data during reset sequence.

kFTFx_PartitionFlexramLoadOptNotLoaded FlexRAM is not loaded during reset sequence.

12.6.4.4 enum ftfx_read_resource_opt_t

Enumerator

kFTFx_ResourceOptionFlashIfr Select code for Program flash 0 IFR, Program flash swap 0 IFR, Data flash 0 IFR.

kFTFx_ResourceOptionVersionId Select code for the version ID.

12.6.4.5 enum ftfx_margin_value_t

Enumerator

kFTFx_MarginValueNormal Use the 'normal' read level for 1s.

kFTFx_MarginValueUser Apply the 'User' margin to the normal read-1 level.

kFTFx_MarginValueFactory Apply the 'Factory' margin to the normal read-1 level.

kFTFx_MarginValueInvalid Not real margin level, Used to determine the range of valid margin level.

12.6.4.6 enum ftfx_security_state_t

Enumerator

kFTFx_SecurityStateNotSecure Flash is not secure.

kFTFx_SecurityStateBackdoorEnabled Flash backdoor is enabled.

kFTFx_SecurityStateBackdoorDisabled Flash backdoor is disabled.

12.6.4.7 enum ftfx_flexram_func_opt_t

Enumerator

kFTFx_FlexramFuncOptAvailableAsRam An option used to make FlexRAM available as RAM.

kFTFx_FlexramFuncOptAvailableForEeprom An option used to make FlexRAM available for E-EPROM.

12.6.4.8 enum ftfx_swap_state_t

Enumerator

kFTFx_SwapStateUninitialized Flash Swap system is in an uninitialized state.

kFTFx_SwapStateReady Flash Swap system is in a ready state.

kFTFx_SwapStateUpdate Flash Swap system is in an update state.

kFTFx_SwapStateUpdateErased Flash Swap system is in an updateErased state.

kFTFx_SwapStateComplete Flash Swap system is in a complete state.

kFTFx_SwapStateDisabled Flash Swap system is in a disabled state.

12.6.5 Function Documentation

12.6.5.1 void FTFx_API_Init (*ftfx_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.
---------------	--

12.6.5.2 status_t FTFx_CMD_Erase (*ftfx_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_AddressError</i>	The address is out of range.
<i>kStatus_FTFx_EraseKeyError</i>	The API erase key is invalid.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.6.5.3 status_t FTFx_CMD_EraseSectorNonBlocking (*ftfx_config_t * config, uint32_t start, uint32_t key*)

This function erases one flash sector size based on the start address.

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>key</i>	The value used to validate all flash erase APIs.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_AddressError</i>	The address is out of range.
<i>kStatus_FTFx_EraseKeyError</i>	The API erase key is invalid.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.

12.6.5.4 status_t FTFx_CMD_EraseAll (*ftfx_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_EraseKeyError</i>	API erase key is invalid.

<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_-CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FTFx_Partition-StatusUpdateFailure</i>	Failed to update the partition status.

12.6.5.5 **status_t FTFx_CMD_EraseAllExecuteOnlySegments (ftfx_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_EraseKey-Error</i>	API erase key is invalid.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.
------------------------------------	--

12.6.5.6 **status_t FTFx_CMD_Program (ftfx_config_t * config, uint32_t start, const uint8_t * src, uint32_t lengthInBytes)**

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the flash.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.
------------------------------------	--

12.6.5.7 status_t FTFx_CMD_ProgramOnce (ftfx_config_t * config, uint32_t index, const uint8_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.6.5.8 status_t FTFx_CMD_ReadOnce (ftfx_config_t * config, uint32_t index, uint8_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_-CommandFailure</i>	Run-time error during the command execution.

12.6.5.9 status_t FTFx_CMD_ReadResource (*ftfx_config_t * config*, *uint32_t start*, *uint8_t * dst*, *uint32_t lengthInBytes*, *ftfx_read_resource_opt_t option*)

This function reads the flash memory with the desired location for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be programmed. Must be word-aligned.
<i>dst</i>	A pointer to the destination buffer of data that is used to store data to be read.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be read. Must be word-aligned.

<i>option</i>	The resource option which indicates which area should be read back.
---------------	---

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with the specified baseline.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.6.5.10 **status_t FTFx_CMD_VerifyErase (ftfx_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, ftfx_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.6.5.11 `status_t FTFx_CMD_VerifyEraseAll(ftfx_config_t * config, ftfx_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.

<i>kStatus_FTFx_Access_Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.6.5.12 status_t FTFx_CMD_VerifyEraseAllExecuteOnlySegments (*ftfx_config_t * config, ftfx_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access_Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.6.5.13 status_t FTFx_CMD_VerifyProgram (*ftfx_config_t * config, uint32_t start, uint32_t lengthInBytes, const uint8_t * expectedData, ftfx_margin_value_t margin, uint32_t * failedAddress, uint32_t * failedData*)

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be verified. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<i>expectedData</i>	A pointer to the expected data that is to be verified against.
<i>margin</i>	Read margin choice.
<i>failedAddress</i>	A pointer to the returned failing address.
<i>failedData</i>	A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FTFx_AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FTFx_AddressError</i>	Address is out of range.
<i>kStatus_FTFx_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_AccessError</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_CommandFailure</i>	Run-time error during the command execution.

12.6.5.14 **status_t FTFx_REG_GetSecurityState (ftfx_config_t * config, ftfx_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_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.

12.6.5.15 **status_t FTFx_CMD_SecurityBypass (*ftfx_config_t * config, const uint8_t * backdoorKey*)**

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>backdoorKey</i>	A pointer to the user buffer containing the backdoor key.

Return values

<i>kStatus_FTFx_Success</i>	API was executed successfully.
<i>kStatus_FTFx_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FTFx_ExecuteIn-RamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FTFx_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FTFx_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FTFx_-CommandFailure</i>	Run-time error during the command execution.

12.6.6 ftx utilities

12.6.6.1 Overview

Macros

- `#define MAKE_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))`
Constructs the version number for drivers.
- `#define MAKE_STATUS(group, code) (((group)*100) + (code)))`
Constructs a status code value from a group and a code number.
- `#define FOUR_CHAR_CODE(a, b, c, d) (((uint32_t)(d) << 24u) | ((uint32_t)(c) << 16u) | ((uint32_t)(b) << 8u) | ((uint32_t)(a)))`
Constructs the four character code for the Flash driver API key.
- `#define B1P4(b) (((uint32_t)(b)&0xFFU) << 24U)`
bytes2word utility.

Alignment macros

- `#define ALIGN_DOWN(x, a) (((uint32_t)(x)) & ~((uint32_t)(a)-1u))`
Alignment(down) utility.
- `#define ALIGN_UP(x, a) ALIGN_DOWN((uint32_t)(x) + (uint32_t)(a)-1u, a)`
Alignment(up) utility.

12.6.6.2 Macro Definition Documentation

12.6.6.2.1 `#define MAKE_VERSION(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))`

12.6.6.2.2 `#define MAKE_STATUS(group, code) (((group)*100) + (code)))`

12.6.6.2.3 `#define FOUR_CHAR_CODE(a, b, c, d) (((uint32_t)(d) << 24u) | ((uint32_t)(c) << 16u) | ((uint32_t)(b) << 8u) | ((uint32_t)(a)))`

12.6.6.2.4 `#define ALIGN_DOWN(x, a) (((uint32_t)(x)) & ~((uint32_t)(a)-1u))`

12.6.6.2.5 `#define ALIGN_UP(x, a) ALIGN_DOWN((uint32_t)(x) + (uint32_t)(a)-1u, a)`

12.6.6.2.6 `#define B1P4(b) (((uint32_t)(b)&0xFFU) << 24U)`

Chapter 13

FlexIO: FlexIO Driver

13.1 Overview

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

Modules

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

13.2 FlexIO Driver

13.2.1 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` {

```
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` {

kFLEXIO_ShifterTimerPolarityOnPositive = 0x0U,

kFLEXIO_ShifterTimerPolarityOnNegative = 0x1U }

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.

Variables

- FLEXIO_Type *const `s_flexioBases` []

Pointers to flexio bases for each instance.

- const `clock_ip_name_t s_flexioClocks []`
Pointers to flexio clocks for each instance.

Driver version

- #define `FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))`
FlexIO driver version.

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.
- uint32_t `FLEXIOGetInstance (FLEXIO_Type *base)`
Get instance number for FLEXIO module.

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.
- static uint32_t `FLEXIO_ReadPinInput (FLEXIO_Type *base)`
Reads the input data on each of the FlexIO pins.
- 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)`

- 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.
- 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.

13.2.2 Data Structure Documentation

13.2.2.1 `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.

Field Documentation

(1) **bool flexio_config_t::enableFastAccess**

13.2.2.2 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.

Field Documentation

(1) **uint32_t flexio_timer_config_t::triggerSelect**

(2) **flexio_timer_trigger_polarity_t flexio_timer_config_t::triggerPolarity**

(3) **flexio_timer_trigger_source_t flexio_timer_config_t::triggerSource**

- (4) `flexio_pin_config_t flexio_timer_config_t::pinConfig`
- (5) `uint32_t flexio_timer_config_t::pinSelect`
- (6) `flexio_pin_polarity_t flexio_timer_config_t::pinPolarity`
- (7) `flexio_timer_mode_t flexio_timer_config_t::timerMode`
- (8) `flexio_timer_output_t flexio_timer_config_t::timerOutput`
- (9) `flexio_timer_decrement_source_t flexio_timer_config_t::timerDecrement`
- (10) `flexio_timer_reset_condition_t flexio_timer_config_t::timerReset`
- (11) `flexio_timer_disable_condition_t flexio_timer_config_t::timerDisable`
- (12) `flexio_timer_enable_condition_t flexio_timer_config_t::timerEnable`
- (13) `flexio_timer_stop_bit_condition_t flexio_timer_config_t::timerStop`
- (14) `flexio_timer_start_bit_condition_t flexio_timer_config_t::timerStart`
- (15) `uint32_t flexio_timer_config_t::timerCompare`

13.2.2.3 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.

Field Documentation

- (1) `uint32_t flexio_shifter_config_t::timerSelect`

- (2) `flexio_shifter_timer_polarity_t flexio_shifter_config_t::timerPolarity`
- (3) `flexio_pin_config_t flexio_shifter_config_t::pinConfig`
- (4) `uint32_t flexio_shifter_config_t::pinSelect`
- (5) `flexio_pin_polarity_t flexio_shifter_config_t::pinPolarity`
- (6) `flexio_shifter_mode_t flexio_shifter_config_t::shifterMode`
- (7) `flexio_shifter_input_source_t flexio_shifter_config_t::inputSource`
- (8) `flexio_shifter_stop_bit_t flexio_shifter_config_t::shifterStop`
- (9) `flexio_shifter_start_bit_t flexio_shifter_config_t::shifterStart`

13.2.3 Macro Definition Documentation

13.2.3.1 #define FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))

13.2.3.2 #define FLEXIO_TIMER_TRIGGER_SEL_PININPUT(x) ((uint32_t)(x) << 1U)

13.2.4 Typedef Documentation

13.2.4.1 typedef void(* flexio_isr_t)(void *base, void *handle)

13.2.5 Enumeration Type Documentation

13.2.5.1 enum flexio_timer_trigger_polarity_t

Enumerator

kFLEXIO_TimerTriggerPolarityActiveHigh Active high.

kFLEXIO_TimerTriggerPolarityActiveLow Active low.

13.2.5.2 enum flexio_timer_trigger_source_t

Enumerator

kFLEXIO_TimerTriggerSourceExternal External trigger selected.

kFLEXIO_TimerTriggerSourceInternal Internal trigger selected.

13.2.5.3 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.

13.2.5.4 enum flexio_pin_polarity_t

Enumerator

kFLEXIO_PinActiveHigh Active high.

kFLEXIO_PinActiveLow Active low.

13.2.5.5 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.

13.2.5.6 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.

13.2.5.7 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.

13.2.5.8 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.

13.2.5.9 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.

13.2.5.10 enum flexio_timer_enable_condition_t

Enumerator

kFLEXIO_TimerEnabledAlways Timer always enabled.

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.

13.2.5.11 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.

13.2.5.12 enum flexio_timer_start_bit_condition_t

Enumerator

kFLEXIO_TimerStartBitDisabled Start bit disabled.

kFLEXIO_TimerStartBitEnabled Start bit enabled.

13.2.5.13 enum flexio_shifter_timer_polarity_t

Enumerator

kFLEXIO_ShifterTimerPolarityOnPositive Shift on positive edge of shift clock.

kFLEXIO_ShifterTimerPolarityOnNegative Shift on negative edge of shift clock.

13.2.5.14 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.

13.2.5.15 enum flexio_shifter_input_source_t

Enumerator

kFLEXIO_ShifterInputFromPin Shifter input from pin.

kFLEXIO_ShifterInputFromNextShifterOutput Shifter input from Shifter N+1.

13.2.5.16 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.

13.2.5.17 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.

13.2.5.18 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.

13.2.6 Function Documentation

13.2.6.1 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);
```

Parameters

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

13.2.6.2 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 <code>flexio_config_t</code> structure

13.2.6.3 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
-------------	--------------------------------

13.2.6.4 `uint32_t` FLEXIO.GetInstance (`FLEXIO_Type` * *base*)

Parameters

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

13.2.6.5 void FLEXIO_Reset (FLEXIO_Type * *base*)

Parameters

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

13.2.6.6 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.

13.2.6.7 static uint32_t FLEXIO_ReadPinInput (FLEXIO_Type * *base*) [inline], [static]

Parameters

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

Returns

FlexIO pin input data

13.2.6.8 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

13.2.6.9 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.

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

13.2.6.10 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 $((1 \ll \text{shifter index}0) | (1 \ll \text{shifter index}1))$

13.2.6.11 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 $(1 \ll \text{shifter index})$

Note

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

13.2.6.12 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 ($1 \ll$ shifter index)

Note

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

13.2.6.13 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 ($1 \ll$ shifter index)

Note

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

13.2.6.14 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 ($1 \ll$ timer index)

Note

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

**13.2.6.15 static void FLEXIO_DisableTimerStatusInterrufts (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 $((1 << \text{timer index}0) | (1 << \text{timer index}1))$

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

Parameters

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

Returns

Shifter status flags

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

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 $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

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

Parameters

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

Returns

Shifter error flags

13.2.6.19 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 << shifter index)

Note

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

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

Parameters

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

Returns

Timer status flags

13.2.6.21 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 ($1 << \text{timer index}$)

Note

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

13.2.6.22 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 $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

Parameters

<i>base</i>	FlexIO peripheral base address
<i>mask</i>	The shifter status mask which can be calculated by ($1 << \text{shifter index}$)
<i>enable</i>	True to enable, false to disable.

13.2.6.23 **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

13.2.6.24 **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.

13.2.6.25 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.

13.2.7 Variable Documentation

13.2.7.1 FLEXIO_Type* const s_flexioBases[]

13.2.7.2 const clock_ip_name_t s_flexioClocks[]

13.3 FlexIO I2C Master Driver

13.3.1 Overview

The MCUXpresso SDK provides a peripheral driver for I2C master function using Flexible I/O module of MCUXpresso SDK 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.

13.3.2 Typical use case

13.3.2.1 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.
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...](#)

Macros

- #define **I2C_RETRY_TIMES** 0U /* Define to zero means keep waiting until the flag is assert/deassert. */
Retry times for waiting flag.

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 {
 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),
 }

- ```

kStatus_FLEXIO_I2C_Timeout = MAKE_STATUS(kStatusGroup_FLEXIO_I2C, 3) }

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, 4, 0))

## Initialization and deinitialization

- status\_t FLEXIO\_I2C\_CheckForBusyBus (FLEXIO\_I2C\_Type \*base)
 

*Make sure the bus isn't already pulled down.*
- status\_t 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.

## 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, **uint16\_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.
- **status\_t 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.*

### 13.3.3 Data Structure Documentation

#### 13.3.3.1 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** [3]  
*Timer index used in FlexIO I2C.*
- **uint32\_t** **baudrate**  
*Master transfer baudrate, used to calculate delay time.*

##### Field Documentation

- (1) **FLEXIO\_Type\*** **FLEXIO\_I2C\_Type::flexioBase**
- (2) **uint8\_t** **FLEXIO\_I2C\_Type::SDAPinIndex**
- (3) **uint8\_t** **FLEXIO\_I2C\_Type::SCLPinIndex**
- (4) **uint8\_t** **FLEXIO\_I2C\_Type::shifterIndex[2]**
- (5) **uint8\_t** **FLEXIO\_I2C\_Type::timerIndex[3]**
- (6) **uint32\_t** **FLEXIO\_I2C\_Type::baudrate**

#### 13.3.3.2 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.*

### Field Documentation

- (1) `bool flexio_i2c_master_config_t::enableMaster`
- (2) `bool flexio_i2c_master_config_t::enableInDoze`
- (3) `bool flexio_i2c_master_config_t::enableInDebug`
- (4) `bool flexio_i2c_master_config_t::enableFastAccess`
- (5) `uint32_t flexio_i2c_master_config_t::baudRate_Bps`

### 13.3.3.3 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.*

### Field Documentation

- (1) `uint32_t flexio_i2c_master_transfer_t::flags`
- (2) `uint8_t flexio_i2c_master_transfer_t::slaveAddress`
- (3) `flexio_i2c_direction_t flexio_i2c_master_transfer_t::direction`
- (4) `uint32_t flexio_i2c_master_transfer_t::subaddress`  
Transferred MSB first.
- (5) `uint8_t flexio_i2c_master_transfer_t::subaddressSize`

- (6) `uint8_t volatile* flexio_i2c_master_transfer_t::data`
- (7) `volatile size_t flexio_i2c_master_transfer_t::dataSize`

### 13.3.3.4 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.*
- `bool needRestart`  
*Whether master needs to send re-start signal.*

#### Field Documentation

- (1) `flexio_i2c_master_transfer_t flexio_i2c_master_handle_t::transfer`
- (2) `size_t flexio_i2c_master_handle_t::transferSize`
- (3) `uint8_t flexio_i2c_master_handle_t::state`
- (4) `flexio_i2c_master_transfer_callback_t flexio_i2c_master_handle_t::completionCallback`

Callback function called at transfer event.

- (5) `void* flexio_i2c_master_handle_t::userData`
- (6) `bool flexio_i2c_master_handle_t::needRestart`

### 13.3.4 Macro Definition Documentation

- 13.3.4.1 `#define I2C_RETRY_TIMES 0U /* Define to zero means keep waiting until the flag is assert/deassert. */`

### 13.3.5 Typedef Documentation

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

### 13.3.6 Enumeration Type Documentation

#### 13.3.6.1 anonymous enum

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.

*kStatus\_FLEXIO\_I2C\_Timeout* Timeout polling status flags.

#### 13.3.6.2 `enum _flexio_i2c_master_interrupt`

Enumerator

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

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

#### 13.3.6.3 `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.

#### 13.3.6.4 `enum flexio_i2c_direction_t`

Enumerator

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

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

### 13.3.7 Function Documentation

#### 13.3.7.1 `status_t FLEXIO_I2C_CheckForBusyBus ( FLEXIO_I2C_Type * base )`

Check the FLEXIO pin status to see whether either of SDA and SCL pin is pulled down.

Parameters

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

Return values

|                                |  |
|--------------------------------|--|
| <i>kStatus_Success</i>         |  |
| <i>kStatus_FLEXIO_I2C_Busy</i> |  |

### 13.3.7.2 status\_t [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 <a href="#">FLEXIO_I2C_Type</a> structure.            |
| <i>masterConfig</i> | Pointer to <a href="#">flexio_i2c_master_config_t</a> structure. |
| <i>srcClock_Hz</i>  | FlexIO source clock in Hz.                                       |

Return values

|                                |                                                |
|--------------------------------|------------------------------------------------|
| <i>kStatus_Success</i>         | Initialization successful                      |
| <i>kStatus_InvalidArgument</i> | The source clock exceed upper range limitation |

### 13.3.7.3 void [FLEXIO\\_I2C\\_MasterDeinit](#) ( [FLEXIO\\_I2C\\_Type](#) \* *base* )

Calling this API Resets the FlexIO I2C master shifer and timer config, module can't work unless the [FLEXIO\\_I2C\\_MasterInit](#) is called.

Parameters

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

### 13.3.7.4 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);
```

Parameters

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

### 13.3.7.5 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 does not have any effect. |

### 13.3.7.6 [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.

### 13.3.7.7 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>• <a href="#">kFLEXIO_I2C_RxFullFlag</a></li> <li>• <a href="#">kFLEXIO_I2C_ReceiveNakFlag</a></li> </ul> |

### 13.3.7.8 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>• <a href="#">kFLEXIO_I2C_TransferCompleteInterruptEnable</a></li> </ul> |

### 13.3.7.9 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.                                     |

### 13.3.7.10 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 |
|--------------------|--------------------|

### 13.3.7.11 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.

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 <a href="#">flexio_i2c_direction_t</a> : <ul style="list-style-type: none"> <li>• kFLEXIO_I2C_Write: Transmit</li> <li>• kFLEXIO_I2C_Read: Receive</li> </ul> |

### 13.3.7.12 void FLEXIO\_I2C\_MasterStop ( FLEXIO\_I2C\_Type \* *base* )

Parameters

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

### 13.3.7.13 void FLEXIO\_I2C\_MasterRepeatedStart ( FLEXIO\_I2C\_Type \* *base* )

Parameters

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

### 13.3.7.14 void FLEXIO\_I2C\_MasterAbortStop ( FLEXIO\_I2C\_Type \* *base* )

Parameters

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

### 13.3.7.15 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. |

### 13.3.7.16 status\_t FLEXIO\_I2C\_MasterSetTransferCount ( [FLEXIO\\_I2C\\_Type](#) \* *base*, [uint16\\_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.         |

### 13.3.7.17 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.                                       |

### 13.3.7.18 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.

Parameters

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

Returns

data byte read.

### 13.3.7.19 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. |
| <i>kStatus_FLEXIO_I2C_-Timeout</i> | Timeout polling status flags.    |

### 13.3.7.20 status\_t 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 <a href="#">FLEXIO_I2C_Type</a> structure. |
| <i>rxBuff</i> | The buffer to store the received bytes.               |
| <i>rxSize</i> | The number of data bytes to be received.              |

Return values

|                                         |                               |
|-----------------------------------------|-------------------------------|
| <i>kStatus_Success</i>                  | Successfully read data.       |
| <i>kStatus_FLEXIO_I2C_-<br/>Timeout</i> | Timeout polling status flags. |

### 13.3.7.21 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 <a href="#">FLEXIO_I2C_Type</a> structure.              |
| <i>xfer</i> | pointer to <a href="#">flexio_i2c_master_transfer_t</a> structure. |

Returns

status of `status_t`.

### 13.3.7.22 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 <a href="#">FLEXIO_I2C_Type</a> structure.                                        |
| <i>handle</i>   | Pointer to <a href="#">flexio_i2c_master_handle_t</a> 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.                |
| <i>kStatus_OutOfRange</i> | The FlexIO type/handle/isr table out of range. |

**13.3.7.23 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\\_MasterTransferGetCount](#) 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. |

**13.3.7.24 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_NoTransferIn-Progress</i> | There is not a non-blocking transaction currently in progress. |
| <i>kStatus_Success</i>               | Successfully return the count.                                 |

### 13.3.7.25 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 <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 |

### 13.3.7.26 void FLEXIO\_I2C\_MasterTransferHandleIRQ ( [void](#) \* *i2cType*, [void](#) \* *i2cHandle* )

Parameters

|                  |                                                                   |
|------------------|-------------------------------------------------------------------|
| <i>i2cType</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure              |
| <i>i2cHandle</i> | Pointer to <a href="#">flexio_i2c_master_transfer_t</a> structure |

## 13.4 FlexIO SPI Driver

### 13.4.1 Overview

The MCUXpresso SDK provides a peripheral driver for an SPI function using the Flexible I/O module of MCUXpresso SDK 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 cannot 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.

### 13.4.2 Typical use case

#### 13.4.2.1 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 = 5000000U;

 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)
 {
 // ...
 }
}

```

### 13.4.2.2 FlexIO\_SPI Send/Receive in DMA way

```

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;
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;

/* Init DMAMUX. */
DMAMUX_Init(EXAMPLE_FLEXIO_SPI_DMAMUX_BASEADDR)

/* Init the DMA/EDMA module */
#if defined(FSL_FEATURE_SOC_DMA_COUNT) && FSL_FEATURE_SOC_DMA_COUNT > 0U
DMA_Init(EXAMPLE_FLEXIO_SPI_DMA_BASEADDR);
DMA_CreateHandle(&txHandle, EXAMPLE_FLEXIO_SPI_DMA_BASEADDR, FLEXIO_SPI_TX_DMA_CHANNEL);
DMA_CreateHandle(&rxHandle, EXAMPLE_FLEXIO_SPI_DMA_BASEADDR, FLEXIO_SPI_RX_DMA_CHANNEL);
#endif /* FSL_FEATURE_SOC_DMA_COUNT */

#if defined(FSL_FEATURE_SOC_EDMA_COUNT) && FSL_FEATURE_SOC_EDMA_COUNT > 0U
edma_config_t edmaConfig;

EDMA_GetDefaultConfig(&edmaConfig);
EDMA_Init(EXAMPLE_FLEXIO_SPI_DMA_BASEADDR, &edmaConfig);
EDMA_CreateHandle(&txHandle, EXAMPLE_FLEXIO_SPI_DMA_BASEADDR,
FLEXIO_SPI_TX_DMA_CHANNEL);
EDMA_CreateHandle(&rxHandle, EXAMPLE_FLEXIO_SPI_DMA_BASEADDR,
FLEXIO_SPI_RX_DMA_CHANNEL);
#endif /* FSL_FEATURE_SOC_EDMA_COUNT */

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. */
DMAMUX_SetSource(EXAMPLE_FLEXIO_SPI_DMAMUX_BASEADDR, FLEXIO_SPI_TX_DMA_CHANNEL, (
dma_request_source_t)dma_request_source_tx);
DMAMUX_SetSource(EXAMPLE_FLEXIO_SPI_DMAMUX_BASEADDR, FLEXIO_SPI_RX_DMA_CHANNEL, (
dma_request_source_t)dma_request_source_rx);
DMAMUX_EnableChannel(EXAMPLE_FLEXIO_SPI_DMAMUX_BASEADDR,
FLEXIO_SPI_TX_DMA_CHANNEL);
DMAMUX_EnableChannel(EXAMPLE_FLEXIO_SPI_DMAMUX_BASEADDR,
FLEXIO_SPI_RX_DMA_CHANNEL);

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 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** (0xFFFFFFFFU)  
*FlexIO SPI dummy transfer data, the data is sent while txData is NULL.*
- #define **SPI\_RETRY\_TIMES** 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/  
*Retry times for waiting flag.*
- #define **FLEXIO\_SPI\_XFER\_DATA\_FORMAT**(flag) ((flag) & (0x7U))  
*Get the transfer data format of width and bit order.*

## 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 {
   
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),
   
kStatus\_FLEXIO\_SPI\_Timeout }
   
*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,
   
kFLEXIO\_SPI\_32BitMode = 0x20U }
   
*FlexIO SPI data length mode options.*
- enum **\_flexio\_spi\_interrupt\_enable** {
   
kFLEXIO\_SPI\_TxEmptyInterruptEnable = 0x1U,
   
kFLEXIO\_SPI\_RxFullInterruptEnable = 0x2U }
   
*Define FlexIO SPI interrupt mask.*
- 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 = 0x0U,
   
kFLEXIO\_SPI\_8bitLsb = 0x1U,
   
kFLEXIO\_SPI\_16bitMsb = 0x2U,
   
kFLEXIO\_SPI\_16bitLsb = 0x3U,
   
kFLEXIO\_SPI\_32bitMsb = 0x4U,
   
kFLEXIO\_SPI\_32bitLsb = 0x5U,
   
kFLEXIO\_SPI\_csContinuous = 0x8U }
   
*Define FlexIO SPI transfer flags.*

## Driver version

- #define **FSL\_FLEXIO\_SPI\_DRIVER\_VERSION** (MAKE\_VERSION(2, 3, 0))  
*FlexIO SPI driver version.*

## 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)  
*Resets the FlexIO SPI timer and shifter config.*
- 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, uint32\_t data)  
*Writes one byte of data, which is sent using the MSB method.*
- static uint32\_t **FLEXIO\_SPI\_ReadData** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_shift\_direction\_t direction)  
*Reads 8 bit/16 bit data.*
- status\_t **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.*
- status\_t **FLEXIO\_SPI\_ReadBlocking** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_shift\_direction\_t direction, uint8\_t \*buffer, size\_t size)  
*Receives a buffer of bytes.*
- status\_t **FLEXIO\_SPI\_MasterTransferBlocking** (FLEXIO\_SPI\_Type \*base, flexio\_spi\_transfer\_t \*xfer)  
*Receives a buffer of bytes.*
- void **FLEXIO\_SPI\_FlushShifters** (FLEXIO\_SPI\_Type \*base)  
*Flush tx/rx shifters.*

## 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.*
- 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.*

### 13.4.3 Data Structure Documentation

#### 13.4.3.1 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.*

##### Field Documentation

(1) **FLEXIO\_Type\*** **FLEXIO\_SPI\_Type::flexioBase**

(2) **uint8\_t FLEXIO\_SPI\_Type::SDOPinIndex**

To set SDO pin in Hi-Z state, user needs to mux the pin as GPIO input and disable all pull up/down in application.

(3) **uint8\_t FLEXIO\_SPI\_Type::SDIPinIndex**

(4) **uint8\_t FLEXIO\_SPI\_Type::SCKPinIndex**

(5) **uint8\_t FLEXIO\_SPI\_Type::CSnPinIndex**

- (6) `uint8_t FLEXIO_SPI_Type::shifterIndex[2]`
- (7) `uint8_t FLEXIO_SPI_Type::timerIndex[2]`

### 13.4.3.2 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.*

#### Field Documentation

- (1) `bool flexio_spi_master_config_t::enableMaster`
- (2) `bool flexio_spi_master_config_t::enableInDoze`
- (3) `bool flexio_spi_master_config_t::enableInDebug`
- (4) `bool flexio_spi_master_config_t::enableFastAccess`
- (5) `uint32_t flexio_spi_master_config_t::baudRate_Bps`
- (6) `flexio_spi_clock_phase_t flexio_spi_master_config_t::phase`
- (7) `flexio_spi_data_bitcount_mode_t flexio_spi_master_config_t::dataMode`

### 13.4.3.3 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.*

### Field Documentation

- (1) **bool flexio\_spi\_slave\_config\_t::enableSlave**
- (2) **bool flexio\_spi\_slave\_config\_t::enableInDoze**
- (3) **bool flexio\_spi\_slave\_config\_t::enableInDebug**
- (4) **bool flexio\_spi\_slave\_config\_t::enableFastAccess**
- (5) **flexio\_spi\_clock\_phase\_t flexio\_spi\_slave\_config\_t::phase**
- (6) **flexio\_spi\_data\_bitcount\_mode\_t flexio\_spi\_slave\_config\_t::dataMode**

### 13.4.3.4 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.*

### Field Documentation

- (1) **uint8\_t\* flexio\_spi\_transfer\_t::txData**
- (2) **uint8\_t\* flexio\_spi\_transfer\_t::rxData**
- (3) **size\_t flexio\_spi\_transfer\_t::dataSize**
- (4) **uint8\_t flexio\_spi\_transfer\_t::flags**

### 13.4.3.5 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`  
*Send data remaining in bytes.*
- `volatile size_t rxRemainingBytes`  
*Receive data remaining in bytes.*
- `volatile uint32_t state`  
*FlexIO SPI internal state.*
- `uint8_t bytePerFrame`  
*SPI mode, 2bytes or 1byte in a frame.*
- `flexio_spi_shift_direction_t direction`  
*Shift direction.*
- `flexio_spi_master_transfer_callback_t callback`  
*FlexIO SPI callback.*
- `void * userData`  
*Callback parameter.*

## Field Documentation

- (1) `uint8_t* flexio_spi_master_handle_t::txData`
- (2) `uint8_t* flexio_spi_master_handle_t::rxData`
- (3) `size_t flexio_spi_master_handle_t::transferSize`
- (4) `volatile size_t flexio_spi_master_handle_t::txRemainingBytes`
- (5) `volatile size_t flexio_spi_master_handle_t::rxRemainingBytes`
- (6) `volatile uint32_t flexio_spi_master_handle_t::state`
- (7) `flexio_spi_shift_direction_t flexio_spi_master_handle_t::direction`
- (8) `flexio_spi_master_transfer_callback_t flexio_spi_master_handle_t::callback`
- (9) `void* flexio_spi_master_handle_t::userData`

### 13.4.4 Macro Definition Documentation

**13.4.4.1 #define FSL\_FLEXIO\_SPI\_DRIVER\_VERSION (MAKE\_VERSION(2, 3, 0))**

**13.4.4.2 #define FLEXIO\_SPI\_DUMMYDATA (0xFFFFFFFFU)**

**13.4.4.3 #define SPI\_RETRY\_TIMES 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/**

**13.4.4.4 #define FLEXIO\_SPI\_XFER\_DATA\_FORMAT( *flag* ) ((*flag*) & (0x7U))**

### 13.4.5 Typedef Documentation

**13.4.5.1 `typedef flexio_spi_master_handle_t flexio_spi_slave_handle_t`**

### 13.4.6 Enumeration Type Documentation

#### 13.4.6.1 anonymous enum

Enumerator

*kStatus\_FLEXIO\_SPI\_Busy* FlexIO SPI is busy.

*kStatus\_FLEXIO\_SPI\_Idle* SPI is idle.

*kStatus\_FLEXIO\_SPI\_Error* FlexIO SPI error.

*kStatus\_FLEXIO\_SPI\_Timeout* FlexIO SPI timeout polling status flags.

#### 13.4.6.2 `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.

#### 13.4.6.3 `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.

#### 13.4.6.4 `enum flexio_spi_data_bitcount_mode_t`

Enumerator

*kFLEXIO\_SPI\_8BitMode* 8-bit data transmission mode.

*kFLEXIO\_SPI\_16BitMode* 16-bit data transmission mode.

*kFLEXIO\_SPI\_32BitMode* 32-bit data transmission mode.

### 13.4.6.5 enum \_flexio\_spi\_interrupt\_enable

Enumerator

**kFLEXIO\_SPI\_TxEmptyInterruptEnable** Transmit buffer empty interrupt enable.

**kFLEXIO\_SPI\_RxFullInterruptEnable** Receive buffer full interrupt enable.

### 13.4.6.6 enum \_flexio\_spi\_status\_flags

Enumerator

**kFLEXIO\_SPI\_TxBufferEmptyFlag** Transmit buffer empty flag.

**kFLEXIO\_SPI\_RxBufferFullFlag** Receive buffer full flag.

### 13.4.6.7 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.

### 13.4.6.8 enum \_flexio\_spi\_transfer\_flags

Note

Use kFLEXIO\_SPI\_csContinuous and one of the other flags to OR together to form the transfer flag.

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.

**kFLEXIO\_SPI\_32bitMsb** FlexIO SPI 32-bit MSB first.

**kFLEXIO\_SPI\_32bitLsb** FlexIO SPI 32-bit LSB first.

**kFLEXIO\_SPI\_csContinuous** Enable the CS signal continuous mode.

## 13.4.7 Function Documentation

### 13.4.7.1 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

1.FlexIO SPI master only support CPOL = 0, which means clock inactive low. 2.For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI master communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by  $2 \times 2 = 4$ . If FlexIO SPI master communicates with FlexIO SPI slave, the maximum baud rate is FlexIO clock frequency divided by  $(1.5 + 2.5) \times 2 = 8$ .

## 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

|                     |                                                                   |
|---------------------|-------------------------------------------------------------------|
| <i>base</i>         | Pointer to the <code>FLEXIO_SPI_Type</code> structure.            |
| <i>masterConfig</i> | Pointer to the <code>flexio_spi_master_config_t</code> structure. |
| <i>srcClock_Hz</i>  | FlexIO source clock in Hz.                                        |

**13.4.7.2 void FLEXIO\_SPI\_MasterDeinit ( FLEXIO\_SPI\_Type \* *base* )**

## Parameters

|             |                                               |
|-------------|-----------------------------------------------|
| <i>base</i> | Pointer to the <code>FLEXIO_SPI_Type</code> . |
|-------------|-----------------------------------------------|

**13.4.7.3 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. |
|---------------------|----------------------------------------------------------------------|

#### 13.4.7.4 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

1.Only one timer is needed in the FlexIO SPI slave. As a result, the second timer index is ignored. 2.- FlexIO SPI slave only support CPOL = 0, which means clock inactive low. 3.For FlexIO SPI master, the input valid time is 1.5 clock cycles, for slave the output valid time is 2.5 clock cycles. So if FlexIO SPI slave communicates with other spi IPs, the maximum baud rate is FlexIO clock frequency divided by  $3*2=6$ . If FlexIO SPI slave communicates with FlexIO SPI master, the maximum baud rate is FlexIO clock frequency divided by  $(1.5+2.5)*2=8$ . 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. |

#### 13.4.7.5 void FLEXIO\_SPI\_SlaveDeinit ( [FLEXIO\\_SPI\\_Type](#) \* *base* )

Parameters

|             |                                                  |
|-------------|--------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> . |
|-------------|--------------------------------------------------|

#### 13.4.7.6 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. |
|--------------------|---------------------------------------------------------------------|

#### 13.4.7.7 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](#)

#### 13.4.7.8 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>• <a href="#">kFLEXIO_SPI_TxEmptyFlag</a></li> <li>• <a href="#">kFLEXIO_SPI_RxEmptyFlag</a></li> </ul> |

```
13.4.7.9 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> |

#### 13.4.7.10 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> |

#### 13.4.7.11 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.

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.           |

#### 13.4.7.12 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.

#### 13.4.7.13 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.

#### 13.4.7.14 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 does not have any effect.  |

#### 13.4.7.15 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.                         |

#### **13.4.7.16 static void FLEXIO\_SPI\_WriteData ( FLEXIO\_SPI\_Type \* *base*, flexio\_spi\_shift\_direction\_t *direction*, 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 the <a href="#">FLEXIO_SPI_Type</a> structure. |
| <i>direction</i> | Shift direction of MSB first or LSB first.                |
| <i>data</i>      | 8/16/32 bit data.                                         |

#### **13.4.7.17 static uint32\_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.

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.

#### **13.4.7.18 status\_t 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.                         |

## Return values

|                                   |                                         |
|-----------------------------------|-----------------------------------------|
| <i>kStatus_Success</i>            | Successfully create the handle.         |
| <i>kStatus_FLEXIO_SPI_Timeout</i> | The transfer timed out and was aborted. |

**13.4.7.19 status\_t 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.                |

## Return values

|                                   |                                         |
|-----------------------------------|-----------------------------------------|
| <i>kStatus_Success</i>            | Successfully create the handle.         |
| <i>kStatus_FLEXIO_SPI_Timeout</i> | The transfer timed out and was aborted. |

**13.4.7.20 status\_t 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 <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 create the handle.         |
| <i>kStatus_FLEXIO_SPI_Timeout</i> | The transfer timed out and was aborted. |

**13.4.7.21 void FLEXIO\_SPI\_FlushShifters ( [FLEXIO\\_SPI\\_Type](#) \* *base* )**

## Parameters

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

**13.4.7.22 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 <a href="#">FLEXIO_SPI_Type</a> structure.                                        |
| <i>handle</i>   | Pointer to the <a href="#">flexio_spi_master_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. |
|---------------------------|------------------------------------------------|

### 13.4.7.23 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.

Parameters

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

### 13.4.7.24 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 <a href="#">flexio_spi_master_handle_t</a> structure to store the transfer state. |

### 13.4.7.25 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 <a href="#">flexio_spi_master_handle_t</a> 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. |

### 13.4.7.26 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 flexio_spi_master_handle_t structure to store the transfer state. |

### 13.4.7.27 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 flexio_spi_slave_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. |

### 13.4.7.28 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 flexio_spi_slave_handle_t 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. |

#### 13.4.7.29 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. |

#### 13.4.7.30 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> | <i>count</i> is Invalid.       |
| <i>kStatus_Success</i>         | Successfully return the count. |

#### 13.4.7.31 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 <code>flexio_spi_slave_handle_t</code> structure to store the transfer state. |

## 13.4.8 FlexIO eDMA SPI Driver

### 13.4.8.1 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.*

#### Driver version

- #define `FSL_FLEXIO_SPI_EDMA_DRIVER_VERSION` (`MAKE_VERSION(2, 3, 0)`)  
*FlexIO SPI EDMA driver version.*

#### 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 number of bytes transferred so far using FlexIO SPI master eDMA.*
- 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 number of bytes transferred so far using FlexIO SPI slave eDMA.*

### 13.4.8.2 Data Structure Documentation

#### 13.4.8.2.1 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.*

##### Field Documentation

- (1) **size\_t flexio\_spi\_master\_edma\_handle\_t::transferSize**
- (2) **uint8\_t flexio\_spi\_master\_edma\_handle\_t::nbytes**

### 13.4.8.3 Macro Definition Documentation

#### 13.4.8.3.1 #define FSL\_FLEXIO\_SPI\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 3, 0))

### 13.4.8.4 Typedef Documentation

#### 13.4.8.4.1 `typedef flexio_spi_master_edma_handle_t flexio_spi_slave_edma_handle_t`

#### 13.4.8.5 Function Documentation

##### 13.4.8.5.1 `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. |

##### 13.4.8.5.2 `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 flexio_spi_master_edma_handle_t 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. |

**13.4.8.5.3 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.                       |

**13.4.8.5.4 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. |

**13.4.8.5.5 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.

## 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.                                   |

**13.4.8.5.6 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. |

**13.4.8.5.7 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. |

#### **13.4.8.5.8 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. |

## 13.5 FlexIO UART Driver

### 13.5.1 Overview

The MCUXpresso SDK 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.

### 13.5.2 Typical use case

#### 13.5.2.1 FlexIO UART send/receive using a polling method

```
uint8_t ch;
```

```

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);
}

```

### 13.5.2.2 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)
{
}

// ...
}

```

### 13.5.2.3 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)
{

```

```

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)
 {
 }
 }
}

// ...
}

```

### 13.5.2.4 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;
 }

 /* Init DMAMUX. */
 DMAMUX_Init(EXAMPLE_FLEXIO_UART_DMAMUX_BASEADDR);

 /* Init the DMA/EDMA module */
#if defined(FSL_FEATURE_SOC_DMA_COUNT) && FSL_FEATURE_SOC_DMA_COUNT > 0U
 DMA_Init(EXAMPLE_FLEXIO_UART_DMA_BASEADDR);
 DMA_CreateHandle(&g_uartTxDmaHandle, EXAMPLE_FLEXIO_UART_DMA_BASEADDR, FLEXIO_UART_TX_DMA_CHANNEL);
 DMA_CreateHandle(&g_uartRxDmaHandle, EXAMPLE_FLEXIO_UART_DMA_BASEADDR, FLEXIO_UART_RX_DMA_CHANNEL);
#endif /* FSL_FEATURE_SOC_DMA_COUNT */

#if defined(FSL_FEATURE_SOC_EDMA_COUNT) && FSL_FEATURE_SOC_EDMA_COUNT > 0U
 edma_config_t edmaConfig;

 EDMA_GetDefaultConfig(&edmaConfig);
 EDMA_Init(EXAMPLE_FLEXIO_UART_DMA_BASEADDR, &edmaConfig);

```

```

 EDMA_CreateHandle(&g_uartTxDmaHandle, EXAMPLE_FLEXIO_UART_DMA_BASEADDR,
FLEXIO_UART_TX_DMA_CHANNEL);
 EDMA_CreateHandle(&g_uartRxDmaHandle, EXAMPLE_FLEXIO_UART_DMA_BASEADDR,
FLEXIO_UART_RX_DMA_CHANNEL);
#endif /* FSL_FEATURE_SOC_EDMA_COUNT */

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. */
DMAMUX_SetSource(EXAMPLE_FLEXIO_UART_DMAMUX_BASEADDR, FLEXIO_UART_TX_DMA_CHANNEL, (
dma_request_source_t)dma_request_source_tx);
DMAMUX_SetSource(EXAMPLE_FLEXIO_UART_DMAMUX_BASEADDR, FLEXIO_UART_RX_DMA_CHANNEL, (
dma_request_source_t)dma_request_source_rx);
DMAMUX_EnableChannel(EXAMPLE_FLEXIO_UART_DMAMUX_BASEADDR,
FLEXIO_UART_TX_DMA_CHANNEL);
DMAMUX_EnableChannel(EXAMPLE_FLEXIO_UART_DMAMUX_BASEADDR,
FLEXIO_UART_RX_DMA_CHANNEL);

FLEXIO_UART_TransferCreateHandleDMA(&uartDev, &g_uartHandle, FLEXIO_UART_UserCallback, NULL, &
g_uartTxDmaHandle, &g_uartRxDmaHandle);

// 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 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...](#)*

## Macros

- #define `UART_RETRY_TIMES` 0U /\* Defining to zero means to keep waiting for the flag until it is assert/deassert. \*/  
*Retry times for waiting flag.*

## 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 {
   
`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),
 `kStatus_FLEXIO_UART_Timeout` = MAKE\_STATUS(kStatusGroup\_FLEXIO\_UART, 7),
 `kStatus_FLEXIO_UART_BaudrateNotSupport` }
   
*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, 4, 0))  
*FlexIO UART driver version.*

## 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)  
*Resets the FlexIO UART shifter and timer config.*
- **void FLEXIO\_UART\_GetDefaultConfig** (**flexio\_uart\_config\_t** \*userConfig)  
*Gets the default configuration to configure the FlexIO UART.*

## 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.*
- **status\_t FLEXIO\_UART\_WriteBlocking** (**FLEXIO\_UART\_Type** \*base, const uint8\_t \*txData, size\_t txSize)  
*Sends a buffer of data bytes.*
- **status\_t 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.*

### 13.5.3 Data Structure Documentation

#### 13.5.3.1 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.*

##### Field Documentation

- (1) `FLEXIO_Type* FLEXIO_UART_Type::flexioBase`
- (2) `uint8_t FLEXIO_UART_Type::TxPinIndex`
- (3) `uint8_t FLEXIO_UART_Type::RxPinIndex`
- (4) `uint8_t FLEXIO_UART_Type::shifterIndex[2]`
- (5) `uint8_t FLEXIO_UART_Type::timerIndex[2]`

#### 13.5.3.2 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*

## Field Documentation

- (1) **bool flexio\_uart\_config\_t::enableUart**
- (2) **bool flexio\_uart\_config\_t::enableFastAccess**
- (3) **uint32\_t flexio\_uart\_config\_t::baudRate\_Bps**

### 13.5.3.3 struct flexio\_uart\_transfer\_t

#### Data Fields

- **size\_t dataSize**  
*Transfer size.*
- **uint8\_t \* data**  
*The buffer of data to be transfer.*
- **uint8\_t \* rxData**  
*The buffer to receive data.*
- **const uint8\_t \* txData**  
*The buffer of data to be sent.*

## Field Documentation

- (1) **uint8\_t\* flexio\_uart\_transfer\_t::data**
- (2) **uint8\_t\* flexio\_uart\_transfer\_t::rxData**
- (3) **const uint8\_t\* flexio\_uart\_transfer\_t::txData**

### 13.5.3.4 struct \_flexio\_uart\_handle

#### Data Fields

- **const 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.*

## Field Documentation

- (1) const uint8\_t\* volatile **flexio\_uart\_handle\_t::txData**
- (2) volatile size\_t **flexio\_uart\_handle\_t::txDataSize**
- (3) uint8\_t\* volatile **flexio\_uart\_handle\_t::rxData**
- (4) volatile size\_t **flexio\_uart\_handle\_t::rxDataSize**
- (5) size\_t **flexio\_uart\_handle\_t::txDataSizeAll**
- (6) size\_t **flexio\_uart\_handle\_t::rxDataSizeAll**
- (7) uint8\_t\* **flexio\_uart\_handle\_t::rxRingBuffer**
- (8) size\_t **flexio\_uart\_handle\_t::rxRingBufferSize**
- (9) volatile uint16\_t **flexio\_uart\_handle\_t::rxRingBufferHead**
- (10) volatile uint16\_t **flexio\_uart\_handle\_t::rxRingBufferTail**
- (11) **flexio\_uart\_transfer\_callback\_t flexio\_uart\_handle\_t::callback**
- (12) void\* **flexio\_uart\_handle\_t::userData**
- (13) volatile uint8\_t **flexio\_uart\_handle\_t::txState**

## 13.5.4 Macro Definition Documentation

**13.5.4.1 #define FSL\_FLEXIO\_UART\_DRIVER\_VERSION (MAKE\_VERSION(2, 4, 0))**

**13.5.4.2 #define UART\_RETRY\_TIMES 0U /\* Defining to zero means to keep waiting for the flag until it is assert/deassert. \*/**

## 13.5.5 Typedef Documentation

**13.5.5.1 `typedef void(* flexio_uart_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, status_t status, void *userData)`**

## 13.5.6 Enumeration Type Documentation

### 13.5.6.1 anonymous enum

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.  
*kStatus\_FLEXIO\_UART\_Timeout* UART times out.  
*kStatus\_FLEXIO\_UART\_BaudrateNotSupport* Baudrate is not supported in current clock source.

### 13.5.6.2 `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

### 13.5.6.3 `enum _flexio_uart_interrupt_enable`

Enumerator

*kFLEXIO\_UART\_TxDataRegEmptyInterruptEnable* Transmit buffer empty interrupt enable.  
*kFLEXIO\_UART\_RxDataRegFullInterruptEnable* Receive buffer full interrupt enable.

### 13.5.6.4 `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.

### 13.5.7 Function Documentation

#### 13.5.7.1 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,
 .enableInDebug = true,
 .enableFastAccess = false,
 .baudRate_Bps = 115200U,
 .bitCountPerChar = 8
};
FLEXIO_UART_Init(base, &config, srcClock_Hz);
```

Parameters

|                    |                                                                |
|--------------------|----------------------------------------------------------------|
| <i>base</i>        | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.     |
| <i>userConfig</i>  | Pointer to the <a href="#">flexio_uart_config_t</a> structure. |
| <i>srcClock_Hz</i> | FlexIO source clock in Hz.                                     |

Return values

|                                               |                                                               |
|-----------------------------------------------|---------------------------------------------------------------|
| <i>kStatus_Success</i>                        | Configuration success.                                        |
| <i>kStatus_FLEXIO_UART-BaudrateNotSupport</i> | Baudrate is not supported for current clock source frequency. |

#### 13.5.7.2 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 <a href="#">FLEXIO_UART_Type</a> structure |
|-------------|-------------------------------------------------------|

### 13.5.7.3 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 <a href="#">flexio_uart_config_t</a> structure. |
|-------------------|----------------------------------------------------------------|

### 13.5.7.4 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.

### 13.5.7.5 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> |

### 13.5.7.6 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.                                          |

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

This function disables the FlexIO UART interrupt.

Parameters

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

### 13.5.7.8 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.

### 13.5.7.9 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.

**13.5.7.10 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.                          |

### 13.5.7.11 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.

Parameters

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

### 13.5.7.12 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 does not have any effect.   |

### 13.5.7.13 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. |
|---------------|-------------------------|

### 13.5.7.14 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.                    |

### 13.5.7.15 status\_t 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.                          |

Return values

|                                    |                                         |
|------------------------------------|-----------------------------------------|
| <i>kStatus_FLEXIO_UART_Timeout</i> | Transmission timed out and was aborted. |
| <i>kStatus_Success</i>             | Successfully wrote all data.            |

### 13.5.7.16 status\_t 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.                   |

Return values

|                                    |                                         |
|------------------------------------|-----------------------------------------|
| <i>kStatus_FLEXIO_UART_Timeout</i> | Transmission timed out and was aborted. |
| <i>kStatus_Success</i>             | Successfully received all data.         |

**13.5.7.17 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*.

Parameters

|                 |                                                                                            |
|-----------------|--------------------------------------------------------------------------------------------|
| <i>base</i>     | to <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>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. |

**13.5.7.18 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 <code>FLEXIO_UART_Type</code> structure.                                      |
| <i>handle</i>         | Pointer to the <code>flexio_uart_handle_t</code> 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.                                                                     |

### 13.5.7.19 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 <code>FLEXIO_UART_Type</code> structure.                                 |
| <i>handle</i> | Pointer to the <code>flexio_uart_handle_t</code> structure to store the transfer state. |

### 13.5.7.20 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 <code>flexio_uart_handle_t</code> 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. |

### 13.5.7.21 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

|               |                                                                                         |
|---------------|-----------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                              |
| <i>handle</i> | Pointer to the <code>flexio_uart_handle_t</code> structure to store the transfer state. |

### 13.5.7.22 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 <code>flexio_uart_handle_t</code> structure to store the transfer state. |
| <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.                    |

**13.5.7.23 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 <a href="#">flexio_uart_handle_t</a> 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.                |

**13.5.7.24 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 <a href="#">flexio_uart_handle_t</a> structure to store the transfer state. |

**13.5.7.25 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_NoTransferIn-Progress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>               | Successfully return the count.                    |

### 13.5.7.26 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 flexio_uart_handle_t structure to store the transfer state. |

## 13.5.8 FlexIO eDMA UART Driver

### 13.5.8.1 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.*

#### Driver version

- #define `FSL_FLEXIO_UART_EDMA_DRIVER_VERSION(MAKE_VERSION(2, 4, 1))`  
*FlexIO UART EDMA driver version.*

#### 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.*

### 13.5.8.2 Data Structure Documentation

#### 13.5.8.2.1 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.*

##### Field Documentation

- (1) **flexio\_uart\_edma\_transfer\_callback\_t flexio\_uart\_edma\_handle\_t::callback**
- (2) **void\* flexio\_uart\_edma\_handle\_t::userData**
- (3) **size\_t flexio\_uart\_edma\_handle\_t::txDataSizeAll**
- (4) **size\_t flexio\_uart\_edma\_handle\_t::rxDataSizeAll**
- (5) **edma\_handle\_t\* flexio\_uart\_edma\_handle\_t::txEdmaHandle**
- (6) **edma\_handle\_t\* flexio\_uart\_edma\_handle\_t::rxEdmaHandle**
- (7) **uint8\_t flexio\_uart\_edma\_handle\_t::nbytes**
- (8) **volatile uint8\_t flexio\_uart\_edma\_handle\_t::txState**

### 13.5.8.3 Macro Definition Documentation

#### 13.5.8.3.1 #define FSL\_FLEXIO\_UART\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 4, 1))

### 13.5.8.4 Typedef Documentation

**13.5.8.4.1** `typedef void(* flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_edma_handle_t *handle, status_t status, void *userData)`

### 13.5.8.5 Function Documentation

**13.5.8.5.1** `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 <code>flexio_uart_edma_handle_t</code> 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. |

**13.5.8.5.2** `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. |

### 13.5.8.5.3 status\_t FLEXIO\_UART\_TransferReceiveEDMA ( ***base***, ***handle***, ***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 <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. |

### 13.5.8.5.4 void FLEXIO\_UART\_TransferAbortSendEDMA ( ***base***, ***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 |

### 13.5.8.5.5 void FLEXIO\_UART\_TransferAbortReceiveEDMA ( ***base***, ***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 <code>flexio_uart_edma_handle_t</code> structure |

### **13.5.8.5.6 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.                    |

### **13.5.8.5.7 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. |
|------------------------|--------------------------------|

# Chapter 14

## FTM: FlexTimer Driver

### 14.1 Overview

The MCUXpresso SDK provides a driver for the FlexTimer Module (FTM) of MCUXpresso SDK devices.

### 14.2 Function groups

The FTM driver supports the generation of PWM signals, input capture, dual edge capture, output compare, and quadrature decoder modes. The driver also supports configuring each of the FTM fault inputs.

#### 14.2.1 Initialization and deinitialization

The function [FTM\\_Init\(\)](#) initializes the FTM with specified configurations. The function [FTM\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the FTM for the requested register update mode for registers with buffers. It also sets up the FTM's fault operation mode and FTM behavior in the BDM mode.

The function [FTM\\_Deinit\(\)](#) disables the FTM counter and turns off the module clock.

#### 14.2.2 PWM Operations

The function [FTM\\_SetupPwm\(\)](#) sets up FTM channels for the PWM output. The function sets up the PWM signal properties for multiple channels. Each channel has its own duty cycle and level-mode specified. However, the same PWM period and PWM mode is applied to all channels requesting the PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle).

The function [FTM\\_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular FTM channel.

The function [FTM\\_UpdateChnlEdgeLevelSelect\(\)](#) updates the level select bits of a particular FTM channel. This can be used to disable the PWM output when making changes to the PWM signal.

#### 14.2.3 Input capture operations

The function [FTM\\_SetupInputCapture\(\)](#) sets up an FTM channel for the input capture. The user can specify the capture edge and a filter value to be used when processing the input signal.

The function [FTM\\_SetupDualEdgeCapture\(\)](#) can be used to measure the pulse width of a signal. A channel pair is used during capture with the input signal coming through a channel n. The user can specify whether to use one-shot or continuous capture, the capture edge for each channel, and any filter value to be used when processing the input signal.

#### 14.2.4 Output compare operations

The function [FTM\\_SetupOutputCompare\(\)](#) sets up an FTM channel for the output comparison. The user can specify the channel output on a successful comparison and a comparison value.

#### 14.2.5 Quad decode

The function [FTM\\_SetupQuadDecode\(\)](#) sets up FTM channels 0 and 1 for quad decoding. The user can specify the quad decoding mode, polarity, and filter properties for each input signal.

#### 14.2.6 Fault operation

The function [FTM\\_SetupFault\(\)](#) sets up the properties for each fault. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

### 14.3 Register Update

Some of the FTM registers have buffers. The driver supports various methods to update these registers with the content of the register buffer. The registers can be updated using the PWM synchronized loading or an intermediate point loading. The update mechanism for register with buffers can be specified through the following fields available in the configuration structure. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/fmMultiple PWM synchronization update modes can be used by providing an OR'ed list of options available in the enumeration [ftm\\_pwm\\_sync\\_method\\_t](#) to the pwmSyncMode field.

When using an intermediate reload points, the PWM synchronization is not required. Multiple reload points can be used by providing an OR'ed list of options available in the enumeration [ftm\\_reload\\_point\\_t](#) to the reloadPoints field.

The driver initialization function sets up the appropriate bits in the FTM module based on the register update options selected.

If software PWM synchronization is used, the below function can be used to initiate a software trigger. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/fm

### 14.4 Typical use case

#### 14.4.1 PWM output

Output a PWM signal on two FTM channels with different duty cycles. Periodically update the PWM signal duty cycle. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ftm

### Data Structures

- struct `ftm_chnl_pwm_signal_param_t`  
*Options to configure a FTM channel's PWM signal.* [More...](#)
- struct `ftm_chnl_pwm_config_param_t`  
*Options to configure a FTM channel using precise setting.* [More...](#)
- struct `ftm_dual_edge_capture_param_t`  
*FlexTimer dual edge capture parameters.* [More...](#)
- struct `ftm_phase_params_t`  
*FlexTimer quadrature decode phase parameters.* [More...](#)
- struct `ftm_fault_param_t`  
*Structure is used to hold the parameters to configure a FTM fault.* [More...](#)
- struct `ftm_config_t`  
*FTM configuration structure.* [More...](#)

### Enumerations

- enum `ftm_chnl_t` {
   
kFTM\_Chnl\_0 = 0U,
   
kFTM\_Chnl\_1,
   
kFTM\_Chnl\_2,
   
kFTM\_Chnl\_3,
   
kFTM\_Chnl\_4,
   
kFTM\_Chnl\_5,
   
kFTM\_Chnl\_6,
   
kFTM\_Chnl\_7 }
   
*List of FTM channels.*
- enum `ftm_fault_input_t` {
   
kFTM\_Fault\_0 = 0U,
   
kFTM\_Fault\_1,
   
kFTM\_Fault\_2,
   
kFTM\_Fault\_3 }
   
*List of FTM faults.*
- enum `ftm_pwm_mode_t` {
   
kFTM\_EdgeAlignedPwm = 0U,
   
kFTM\_CenterAlignedPwm,
   
kFTM\_EdgeAlignedCombinedPwm,
   
kFTM\_CenterAlignedCombinedPwm,
   
kFTM\_AsymmetricalCombinedPwm }
   
*FTM PWM operation modes.*
- enum `ftm_pwm_level_select_t` {

```
kFTM_NoPwmSignal = 0U,
kFTM_LowTrue,
kFTM_HighTrue }
```

*FTM PWM output pulse mode: high-true, low-true or no output.*

- enum `ftm_output_compare_mode_t` {
   
kFTM\_NoOutputSignal = (1U << FTM\_CnSC\_MSA\_SHIFT),
   
kFTM\_ToggleOnMatch = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (1U << FTM\_CnSC\_ELSA\_S-HIFT)),
   
kFTM\_ClearOnMatch = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (2U << FTM\_CnSC\_ELSA\_SHIFT)),
   
kFTM\_SetOnMatch = ((1U << FTM\_CnSC\_MSA\_SHIFT) | (3U << FTM\_CnSC\_ELSA\_SHIFT)) }

*FlexTimer output compare mode.*

- enum `ftm_input_capture_edge_t` {
   
kFTM\_RisingEdge = (1U << FTM\_CnSC\_ELSA\_SHIFT),
   
kFTM\_FallingEdge = (2U << FTM\_CnSC\_ELSA\_SHIFT),
   
kFTM\_RiseAndFallEdge = (3U << FTM\_CnSC\_ELSA\_SHIFT) }

*FlexTimer input capture edge.*

- enum `ftm_dual_edge_capture_mode_t` {
   
kFTM\_OneShot = 0U,
   
kFTM\_Continuous = (1U << FTM\_CnSC\_MSA\_SHIFT) }

*FlexTimer dual edge capture modes.*

- enum `ftm_quad_decode_mode_t` {
   
kFTM\_QuadPhaseEncode = 0U,
   
kFTM\_QuadCountAndDir }
  
- enum `ftm_phase_polarity_t` {
   
kFTM\_QuadPhaseNormal = 0U,
   
kFTM\_QuadPhaseInvert }

*FlexTimer quadrature phase polarities.*

- enum `ftm_deadtime_prescale_t` {
   
kFTM\_Deadtime\_Prescale\_1 = 1U,
   
kFTM\_Deadtime\_Prescale\_4,
   
kFTM\_Deadtime\_Prescale\_16 }

*FlexTimer pre-scaler factor for the dead time insertion.*

- enum `ftm_clock_source_t` {
   
kFTM\_SystemClock = 1U,
   
kFTM\_FixedClock,
   
kFTM\_ExternalClock }

*FlexTimer clock source selection.*

- enum `ftm_clock_prescale_t` {

```
kFTM_Prescale_Divide_1 = 0U,
kFTM_Prescale_Divide_2,
kFTM_Prescale_Divide_4,
kFTM_Prescale_Divide_8,
kFTM_Prescale_Divide_16,
kFTM_Prescale_Divide_32,
kFTM_Prescale_Divide_64,
kFTM_Prescale_Divide_128 }
```

*FlexTimer pre-scaler factor selection for the clock source.*

- enum `ftm_bdm_mode_t` {
   
kFTM\_BdmMode\_0 = 0U,
   
kFTM\_BdmMode\_1,
   
kFTM\_BdmMode\_2,
   
kFTM\_BdmMode\_3 }

*Options for the FlexTimer behaviour in BDM Mode.*

- enum `ftm_fault_mode_t` {
   
kFTM\_Fault\_Disable = 0U,
   
kFTM\_Fault\_EvenChnls,
   
kFTM\_Fault\_AllChnlsMan,
   
kFTM\_Fault\_AllChnlsAuto }

*Options for the FTM fault control mode.*

- enum `ftm_external_trigger_t` {
   
kFTM\_Chnl0Trigger = (1U << 4),
   
kFTM\_Chnl1Trigger = (1U << 5),
   
kFTM\_Chnl2Trigger = (1U << 0),
   
kFTM\_Chnl3Trigger = (1U << 1),
   
kFTM\_Chnl4Trigger = (1U << 2),
   
kFTM\_Chnl5Trigger = (1U << 3),
   
kFTM\_Chnl6Trigger,
   
kFTM\_Chnl7Trigger,
   
kFTM\_InitTrigger = (1U << 6),
   
kFTM\_ReloadInitTrigger = (1U << 7) }

*FTM external trigger options.*

- enum `ftm_pwm_sync_method_t` {
   
kFTM\_SoftwareTrigger = FTM\_SYNC\_SWSYNC\_MASK,
   
kFTM\_HardwareTrigger\_0 = FTM\_SYNC\_TRIG0\_MASK,
   
kFTM\_HardwareTrigger\_1 = FTM\_SYNC\_TRIG1\_MASK,
   
kFTM\_HardwareTrigger\_2 = FTM\_SYNC\_TRIG2\_MASK }

*FlexTimer PWM sync options to update registers with buffer.*

- enum `ftm_reload_point_t` {

```

kFTM_Chnl0Match = (1U << 0),
kFTM_Chnl1Match = (1U << 1),
kFTM_Chnl2Match = (1U << 2),
kFTM_Chnl3Match = (1U << 3),
kFTM_Chnl4Match = (1U << 4),
kFTM_Chnl5Match = (1U << 5),
kFTM_Chnl6Match = (1U << 6),
kFTM_Chnl7Match = (1U << 7),
kFTM_CntMax = (1U << 8),
kFTM_CntMin = (1U << 9),
kFTM_HalfCycMatch = (1U << 10) }

```

*FTM options available as loading point for register reload.*

- enum `ftm_interrupt_enable_t` {

```

kFTM_Chnl0InterruptEnable = (1U << 0),
kFTM_Chnl1InterruptEnable = (1U << 1),
kFTM_Chnl2InterruptEnable = (1U << 2),
kFTM_Chnl3InterruptEnable = (1U << 3),
kFTM_Chnl4InterruptEnable = (1U << 4),
kFTM_Chnl5InterruptEnable = (1U << 5),
kFTM_Chnl6InterruptEnable = (1U << 6),
kFTM_Chnl7InterruptEnable = (1U << 7),
kFTM_FaultInterruptEnable = (1U << 8),
kFTM_TimeOverflowInterruptEnable = (1U << 9),
kFTM_ReloadInterruptEnable = (1U << 10) }

```

*List of FTM interrupts.*

- enum `ftm_status_flags_t` {

```

kFTM_Chnl0Flag = (1U << 0),
kFTM_Chnl1Flag = (1U << 1),
kFTM_Chnl2Flag = (1U << 2),
kFTM_Chnl3Flag = (1U << 3),
kFTM_Chnl4Flag = (1U << 4),
kFTM_Chnl5Flag = (1U << 5),
kFTM_Chnl6Flag = (1U << 6),
kFTM_Chnl7Flag = (1U << 7),
kFTM_FaultFlag = (1U << 8),
kFTM_TimeOverflowFlag = (1U << 9),
kFTM_ChnlTriggerFlag = (1U << 10),
kFTM_ReloadFlag = (1U << 11) }

```

*List of FTM flags.*

## Functions

- void `FTM_SetupFaultInput` (FTM\_Type \*base, `ftm_fault_input_t` faultNumber, const `ftm_fault_param_t` \*faultParams)
 

*Sets up the working of the FTM fault inputs protection.*
- static void `FTM_SetGlobalTimeBaseOutputEnable` (FTM\_Type \*base, bool enable)

- Enables or disables the FTM global time base signal generation to other FTMs.
- static void **FTM\_SetOutputMask** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool mask)  
Sets the FTM peripheral timer channel output mask.
- static void **FTM\_SetPwmOutputEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool value)  
Allows users to enable an output on an FTM channel.
- static void **FTM\_SetSoftwareTrigger** (FTM\_Type \*base, bool enable)  
Enables or disables the FTM software trigger for PWM synchronization.
- static void **FTM\_SetWriteProtection** (FTM\_Type \*base, bool enable)  
Enables or disables the FTM write protection.
- static void **FTM\_EnableDmaTransfer** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, bool enable)  
Enable DMA transfer or not.

## Driver version

- #define **FSL\_FTM\_DRIVER\_VERSION** (MAKE\_VERSION(2, 5, 0))  
FTM driver version 2.5.0.

## Initialization and deinitialization

- status\_t **FTM\_Init** (FTM\_Type \*base, const **ftm\_config\_t** \*config)  
Ungates the FTM clock and configures the peripheral for basic operation.
- void **FTM\_Deinit** (FTM\_Type \*base)  
Gates the FTM clock.
- void **FTM\_GetDefaultConfig** (**ftm\_config\_t** \*config)  
Fills in the FTM configuration structure with the default settings.
- static **ftm\_clock\_prescale\_t** **FTM\_CalculateCounterClkDiv** (FTM\_Type \*base, uint32\_t counterPeriod\_Hz, uint32\_t srcClock\_Hz)  
brief Calculates the counter clock prescaler.

## Channel mode operations

- status\_t **FTM\_SetupPwm** (FTM\_Type \*base, const **ftm\_chnl\_pwm\_signal\_param\_t** \*chnlParams, uint8\_t numOfChnls, **ftm\_pwm\_mode\_t** mode, uint32\_t pwmFreq\_Hz, uint32\_t srcClock\_Hz)  
Configures the PWM signal parameters.
- status\_t **FTM\_UpdatePwmDutyCycle** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_pwm\_mode\_t** currentPwmMode, uint8\_t dutyCyclePercent)  
Updates the duty cycle of an active PWM signal.
- void **FTM\_UpdateChnlEdgeLevelSelect** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, uint8\_t level)  
Updates the edge level selection for a channel.
- status\_t **FTM\_SetupPwmMode** (FTM\_Type \*base, const **ftm\_chnl\_pwm\_config\_param\_t** \*chnlParams, uint8\_t numOfChnls, **ftm\_pwm\_mode\_t** mode)  
Configures the PWM mode parameters.
- void **FTM\_SetupInputCapture** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_input\_capture\_edge\_t** captureMode, uint32\_t filterValue)  
Enables capturing an input signal on the channel using the function parameters.
- void **FTM\_SetupOutputCompare** (FTM\_Type \*base, **ftm\_chnl\_t** chnlNumber, **ftm\_output\_compare\_mode\_t** compareMode, uint32\_t compareValue)  
Configures the FTM to generate timed pulses.
- void **FTM\_SetupDualEdgeCapture** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, const **ftm\_dual\_edge\_capture\_param\_t** \*edgeParam, uint32\_t filterValue)

*Configures the dual edge capture mode of the FTM.*

## Interrupt Interface

- void [FTM\\_EnableInterrupts](#) (FTM\_Type \*base, uint32\_t mask)  
*Enables the selected FTM interrupts.*
- void [FTM\\_DisableInterrupts](#) (FTM\_Type \*base, uint32\_t mask)  
*Disables the selected FTM interrupts.*
- uint32\_t [FTM\\_GetEnabledInterrupts](#) (FTM\_Type \*base)  
*Gets the enabled FTM interrupts.*

## Status Interface

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

## Read and write the timer period

- static void [FTM\\_SetTimerPeriod](#) (FTM\_Type \*base, uint32\_t ticks)  
*Sets the timer period in units of ticks.*
- static uint32\_t [FTM\\_GetCurrentTimerCount](#) (FTM\_Type \*base)  
*Reads the current timer counting value.*
- static uint32\_t [FTM\\_GetInputCaptureValue](#) (FTM\_Type \*base, [ftm\\_chnl\\_t](#) chnlNumber)  
*Reads the captured value.*

## Timer Start and Stop

- static void [FTM\\_StartTimer](#) (FTM\_Type \*base, [ftm\\_clock\\_source\\_t](#) clockSource)  
*Starts the FTM counter.*
- static void [FTM\\_StopTimer](#) (FTM\_Type \*base)  
*Stops the FTM counter.*

## Software output control

- static void [FTM\\_SetSoftwareCtrlEnable](#) (FTM\_Type \*base, [ftm\\_chnl\\_t](#) chnlNumber, bool value)  
*Enables or disables the channel software output control.*
- static void [FTM\\_SetSoftwareCtrlVal](#) (FTM\_Type \*base, [ftm\\_chnl\\_t](#) chnlNumber, bool value)  
*Sets the channel software output control value.*

## Channel pair operations

- static void [FTM\\_SetFaultControlEnable](#) (FTM\_Type \*base, [ftm\\_chnl\\_t](#) chnlPairNumber, bool value)  
*This function enables/disables the fault control in a channel pair.*
- static void [FTM\\_SetDeadTimeEnable](#) (FTM\_Type \*base, [ftm\\_chnl\\_t](#) chnlPairNumber, bool value)  
*This function enables/disables the dead time insertion in a channel pair.*
- static void [FTM\\_SetComplementaryEnable](#) (FTM\_Type \*base, [ftm\\_chnl\\_t](#) chnlPairNumber, bool value)

- This function enables/disables complementary mode in a channel pair.
- static void **FTM\_SetInvertEnable** (FTM\_Type \*base, **ftm\_chnl\_t** chnlPairNumber, bool value)
 

This function enables/disables inverting control in a channel pair.

## Quad Decoder

- void **FTM\_SetupQuadDecode** (FTM\_Type \*base, const **ftm\_phase\_params\_t** \*phaseAParams, const **ftm\_phase\_params\_t** \*phaseBParams, **ftm\_quad\_decode\_mode\_t** quadMode)
 

Configures the parameters and activates the quadrature decoder mode.
- static void **FTM\_SetQuadDecoderModuloValue** (FTM\_Type \*base, uint32\_t startValue, uint32\_t overValue)
 

Sets the modulo values for Quad Decoder.
- static uint32\_t **FTM\_GetQuadDecoderCounterValue** (FTM\_Type \*base)
 

Gets the current Quad Decoder counter value.
- static void **FTM\_ClearQuadDecoderCounterValue** (FTM\_Type \*base)
 

Clears the current Quad Decoder counter value.

## 14.5 Data Structure Documentation

### 14.5.1 struct **ftm\_chnl\_pwm\_signal\_param\_t**

#### Data Fields

- **ftm\_chnl\_t chnlNumber**  
*The channel/channel pair number.*
- **ftm\_pwm\_level\_select\_t level**  
*PWM output active level select.*
- **uint8\_t dutyCyclePercent**  
*PWM pulse width, value should be between 0 to 100 0 = inactive signal(0% duty cycle)...*
- **uint8\_t firstEdgeDelayPercent**  
*Used only in kFTM\_AsymmetricalCombinedPwm mode to generate an asymmetrical PWM.*
- **bool enableComplementary**  
*Used only in combined PWM mode.*
- **bool enableDeadtime**  
*Used only in combined PWM mode with enable complementary.*

#### Field Documentation

##### (1) **ftm\_chnl\_t ftm\_chnl\_pwm\_signal\_param\_t::chnlNumber**

In combined mode, this represents the channel pair number.

##### (2) **ftm\_pwm\_level\_select\_t ftm\_chnl\_pwm\_signal\_param\_t::level**

##### (3) **uint8\_t ftm\_chnl\_pwm\_signal\_param\_t::dutyCyclePercent**

100 = always active signal (100% duty cycle).

**(4) uint8\_t ftm\_chnl\_pwm\_signal\_param\_t::firstEdgeDelayPercent**

Specifies the delay to the first edge in a PWM period. If unsure leave as 0; Should be specified as a percentage of the PWM period

**(5) bool ftm\_chnl\_pwm\_signal\_param\_t::enableComplementary**

true: The combined channels output complementary signals; false: The combined channels output same signals;

**(6) bool ftm\_chnl\_pwm\_signal\_param\_t::enableDeadtime**

true: The deadtime insertion in this pair of channels is enabled; false: The deadtime insertion in this pair of channels is disabled.

**14.5.2 struct ftm\_chnl\_pwm\_config\_param\_t****Data Fields**

- [ftm\\_chnl\\_t chnlNumber](#)  
*The channel/channel pair number.*
- [ftm\\_pwm\\_level\\_select\\_t level](#)  
*PWM output active level select.*
- [uint16\\_t dutyValue](#)  
*PWM pulse width, the uint of this value is timer ticks.*
- [uint16\\_t firstEdgeValue](#)  
*Used only in kFTM\_AsymmetricalCombinedPwm mode to generate an asymmetrical PWM.*
- [bool enableComplementary](#)  
*Used only in combined PWM mode.*
- [bool enableDeadtime](#)  
*Used only in combined PWM mode with enable complementary.*

**Field Documentation****(1) ftm\_chnl\_t ftm\_chnl\_pwm\_config\_param\_t::chnlNumber**

In combined mode, this represents the channel pair number.

**(2) ftm\_pwm\_level\_select\_t ftm\_chnl\_pwm\_config\_param\_t::level****(3) uint16\_t ftm\_chnl\_pwm\_config\_param\_t::dutyValue****(4) uint16\_t ftm\_chnl\_pwm\_config\_param\_t::firstEdgeValue**

Specifies the delay to the first edge in a PWM period. If unsure leave as 0, uint of this value is timer ticks.

**(5) bool ftm\_chnl\_pwm\_config\_param\_t::enableComplementary**

true: The combined channels output complementary signals; false: The combined channels output same signals;

**(6) bool ftm\_chnl\_pwm\_config\_param\_t::enableDeadtime**

true: The deadtime insertion in this pair of channels is enabled; false: The deadtime insertion in this pair of channels is disabled.

**14.5.3 struct ftm\_dual\_edge\_capture\_param\_t****Data Fields**

- `ftm_dual_edge_capture_mode_t mode`  
*Dual Edge Capture mode.*
- `ftm_input_capture_edge_t currChanEdgeMode`  
*Input capture edge select for channel n.*
- `ftm_input_capture_edge_t nextChanEdgeMode`  
*Input capture edge select for channel n+1.*

**14.5.4 struct ftm\_phase\_params\_t****Data Fields**

- `bool enablePhaseFilter`  
*True: enable phase filter; false: disable filter.*
- `uint32_t phaseFilterVal`  
*Filter value, used only if phase filter is enabled.*
- `ftm_phase_polarity_t phasePolarity`  
*Phase polarity.*

**14.5.5 struct ftm\_fault\_param\_t****Data Fields**

- `bool enableFaultInput`  
*True: Fault input is enabled; false: Fault input is disabled.*
- `bool faultLevel`  
*True: Fault polarity is active low; in other words, '0' indicates a fault; False: Fault polarity is active high.*
- `bool useFaultFilter`  
*True: Use the filtered fault signal; False: Use the direct path from fault input.*

### 14.5.6 struct `ftm_config_t`

This structure holds the configuration settings for the FTM peripheral. To initialize this structure to reasonable defaults, call the [FTM\\_GetDefaultConfig\(\)](#) function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

### Data Fields

- **`ftm_clock_prescale_t prescale`**  
*FTM clock prescale value.*
- **`ftm_bdm_mode_t bdmMode`**  
*FTM behavior in BDM mode.*
- **`uint32_t pwmSyncMode`**  
*Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration [ftm\\_pwm\\_sync\\_method\\_t](#).*
- **`uint32_t reloadPoints`**  
*FTM reload points; When using this, the PWM synchronization is not required.*
- **`ftm_fault_mode_t faultMode`**  
*FTM fault control mode.*
- **`uint8_t faultFilterValue`**  
*Fault input filter value.*
- **`ftm_deadtime_prescale_t deadTimePrescale`**  
*The dead time prescalar value.*
- **`uint32_t deadTimeValue`**  
*The dead time value deadTimeValue's available range is 0-1023 when register has DTVALEX, otherwise its available range is 0-63.*
- **`uint32_t extTriggers`**  
*External triggers to enable.*
- **`uint8_t chnlInitState`**  
*Defines the initialization value of the channels in OUTINT register.*
- **`uint8_t chnlPolarity`**  
*Defines the output polarity of the channels in POL register.*
- **`bool useGlobalTimeBase`**  
*True: Use of an external global time base is enabled; False: disabled.*

### Field Documentation

(1) **`uint32_t ftm_config_t::pwmSyncMode`**

(2) **`uint32_t ftm_config_t::reloadPoints`**

Multiple reload points can be used by providing an OR'ed list of options available in enumeration [ftm\\_reload\\_point\\_t](#).

(3) **`uint32_t ftm_config_t::deadTimeValue`**

**(4) uint32\_t ftm\_config\_t::extTriggers**

Multiple trigger sources can be enabled by providing an OR'ed list of options available in enumeration [ftm\\_external\\_trigger\\_t](#).

**14.6 Macro Definition Documentation****14.6.1 #define FSL\_FTM\_DRIVER\_VERSION (MAKE\_VERSION(2, 5, 0))****14.7 Enumeration Type Documentation****14.7.1 enum ftm\_chnl\_t**

Note

Actual number of available channels is SoC dependent

Enumerator

- kFTM\_Chnl\_0*** FTM channel number 0.
- kFTM\_Chnl\_1*** FTM channel number 1.
- kFTM\_Chnl\_2*** FTM channel number 2.
- kFTM\_Chnl\_3*** FTM channel number 3.
- kFTM\_Chnl\_4*** FTM channel number 4.
- kFTM\_Chnl\_5*** FTM channel number 5.
- kFTM\_Chnl\_6*** FTM channel number 6.
- kFTM\_Chnl\_7*** FTM channel number 7.

**14.7.2 enum ftm\_fault\_input\_t**

Enumerator

- kFTM\_Fault\_0*** FTM fault 0 input pin.
- kFTM\_Fault\_1*** FTM fault 1 input pin.
- kFTM\_Fault\_2*** FTM fault 2 input pin.
- kFTM\_Fault\_3*** FTM fault 3 input pin.

**14.7.3 enum ftm\_pwm\_mode\_t**

Enumerator

- kFTM\_EdgeAlignedPwm*** Edge-aligned PWM.
- kFTM\_CenterAlignedPwm*** Center-aligned PWM.
- kFTM\_EdgeAlignedCombinedPwm*** Edge-aligned combined PWM.

*kFTM\_CenterAlignedCombinedPwm* Center-aligned combined PWM.

*kFTM\_AsymmetricalCombinedPwm* Asymmetrical combined PWM.

#### 14.7.4 enum ftm\_pwm\_level\_select\_t

Enumerator

*kFTM\_NoPwmSignal* No PWM output on pin.

*kFTM\_LowTrue* Low true pulses.

*kFTM\_HighTrue* High true pulses.

#### 14.7.5 enum ftm\_output\_compare\_mode\_t

Enumerator

*kFTM\_NoOutputSignal* No channel output when counter reaches CnV.

*kFTM\_ToggleOnMatch* Toggle output.

*kFTM\_ClearOnMatch* Clear output.

*kFTM\_SetOnMatch* Set output.

#### 14.7.6 enum ftm\_input\_capture\_edge\_t

Enumerator

*kFTM\_RisingEdge* Capture on rising edge only.

*kFTM\_FallingEdge* Capture on falling edge only.

*kFTM\_RiseAndFallEdge* Capture on rising or falling edge.

#### 14.7.7 enum ftm\_dual\_edge\_capture\_mode\_t

Enumerator

*kFTM\_OneShot* One-shot capture mode.

*kFTM\_Continuous* Continuous capture mode.

#### 14.7.8 enum ftm\_quad\_decode\_mode\_t

Enumerator

*kFTM\_QuadPhaseEncode* Phase A and Phase B encoding mode.

*kFTM\_QuadCountAndDir* Count and direction encoding mode.

**14.7.9 enum ftm\_phase\_polarity\_t**

Enumerator

*kFTM\_QuadPhaseNormal* Phase input signal is not inverted.*kFTM\_QuadPhaseInvert* Phase input signal is inverted.**14.7.10 enum ftm\_deadtime\_prescale\_t**

Enumerator

*kFTM\_Deadtime\_Prescale\_1* Divide by 1.*kFTM\_Deadtime\_Prescale\_4* Divide by 4.*kFTM\_Deadtime\_Prescale\_16* Divide by 16.**14.7.11 enum ftm\_clock\_source\_t**

Enumerator

*kFTM\_SystemClock* System clock selected.*kFTM\_FixedClock* Fixed frequency clock.*kFTM\_ExternalClock* External clock.**14.7.12 enum ftm\_clock\_prescale\_t**

Enumerator

*kFTM\_Prescale\_Divide\_1* Divide by 1.*kFTM\_Prescale\_Divide\_2* Divide by 2.*kFTM\_Prescale\_Divide\_4* Divide by 4.*kFTM\_Prescale\_Divide\_8* Divide by 8.*kFTM\_Prescale\_Divide\_16* Divide by 16.*kFTM\_Prescale\_Divide\_32* Divide by 32.*kFTM\_Prescale\_Divide\_64* Divide by 64.*kFTM\_Prescale\_Divide\_128* Divide by 128.**14.7.13 enum ftm\_bdm\_mode\_t**

Enumerator

*kFTM\_BdmMode\_0* FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.

***kFTM\_BdmMode\_1*** FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers.

***kFTM\_BdmMode\_2*** FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.

***kFTM\_BdmMode\_3*** FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode.

#### 14.7.14 enum ftm\_fault\_mode\_t

Enumerator

***kFTM\_Fault\_Disable*** Fault control is disabled for all channels.

***kFTM\_Fault\_EvenChnls*** Enabled for even channels only(0,2,4,6) with manual fault clearing.

***kFTM\_Fault\_AllChnlsMan*** Enabled for all channels with manual fault clearing.

***kFTM\_Fault\_AllChnlsAuto*** Enabled for all channels with automatic fault clearing.

#### 14.7.15 enum ftm\_external\_trigger\_t

Note

Actual available external trigger sources are SoC-specific

Enumerator

***kFTM\_Chnl0Trigger*** Generate trigger when counter equals chnl 0 CnV reg.

***kFTM\_Chnl1Trigger*** Generate trigger when counter equals chnl 1 CnV reg.

***kFTM\_Chnl2Trigger*** Generate trigger when counter equals chnl 2 CnV reg.

***kFTM\_Chnl3Trigger*** Generate trigger when counter equals chnl 3 CnV reg.

***kFTM\_Chnl4Trigger*** Generate trigger when counter equals chnl 4 CnV reg.

***kFTM\_Chnl5Trigger*** Generate trigger when counter equals chnl 5 CnV reg.

***kFTM\_Chnl6Trigger*** Available on certain SoC's, generate trigger when counter equals chnl 6 CnV reg.

***kFTM\_Chnl7Trigger*** Available on certain SoC's, generate trigger when counter equals chnl 7 CnV reg.

***kFTM\_InitTrigger*** Generate Trigger when counter is updated with CNTIN.

***kFTM\_ReloadInitTrigger*** Available on certain SoC's, trigger on reload point.

#### 14.7.16 enum ftm\_pwm\_sync\_method\_t

Enumerator

***kFTM\_SoftwareTrigger*** Software triggers PWM sync.

***kFTM\_HardwareTrigger\_0*** Hardware trigger 0 causes PWM sync.

***kFTM\_HardwareTrigger\_1*** Hardware trigger 1 causes PWM sync.

***kFTM\_HardwareTrigger\_2*** Hardware trigger 2 causes PWM sync.

#### 14.7.17 enum ftm\_reload\_point\_t

Note

Actual available reload points are SoC-specific

Enumerator

***kFTM\_Chnl0Match*** Channel 0 match included as a reload point.

***kFTM\_Chnl1Match*** Channel 1 match included as a reload point.

***kFTM\_Chnl2Match*** Channel 2 match included as a reload point.

***kFTM\_Chnl3Match*** Channel 3 match included as a reload point.

***kFTM\_Chnl4Match*** Channel 4 match included as a reload point.

***kFTM\_Chnl5Match*** Channel 5 match included as a reload point.

***kFTM\_Chnl6Match*** Channel 6 match included as a reload point.

***kFTM\_Chnl7Match*** Channel 7 match included as a reload point.

***kFTM\_CntMax*** Use in up-down count mode only, reload when counter reaches the maximum value.

***kFTM\_CntMin*** Use in up-down count mode only, reload when counter reaches the minimum value.

***kFTM\_HalfCycMatch*** Available on certain SoC's, half cycle match reload point.

#### 14.7.18 enum ftm\_interrupt\_enable\_t

Note

Actual available interrupts are SoC-specific

Enumerator

***kFTM\_Chnl0InterruptEnable*** Channel 0 interrupt.

***kFTM\_Chnl1InterruptEnable*** Channel 1 interrupt.

***kFTM\_Chnl2InterruptEnable*** Channel 2 interrupt.

***kFTM\_Chnl3InterruptEnable*** Channel 3 interrupt.

***kFTM\_Chnl4InterruptEnable*** Channel 4 interrupt.

***kFTM\_Chnl5InterruptEnable*** Channel 5 interrupt.

***kFTM\_Chnl6InterruptEnable*** Channel 6 interrupt.

***kFTM\_Chnl7InterruptEnable*** Channel 7 interrupt.

***kFTM\_FaultInterruptEnable*** Fault interrupt.

***kFTM\_TimeOverflowInterruptEnable*** Time overflow interrupt.

***kFTM\_ReloadInterruptEnable*** Reload interrupt; Available only on certain SoC's.

### 14.7.19 enum ftm\_status\_flags\_t

Note

Actual available flags are SoC-specific

Enumerator

|                              |                                               |
|------------------------------|-----------------------------------------------|
| <i>kFTM_Chnl0Flag</i>        | Channel 0 Flag.                               |
| <i>kFTM_Chnl1Flag</i>        | Channel 1 Flag.                               |
| <i>kFTM_Chnl2Flag</i>        | Channel 2 Flag.                               |
| <i>kFTM_Chnl3Flag</i>        | Channel 3 Flag.                               |
| <i>kFTM_Chnl4Flag</i>        | Channel 4 Flag.                               |
| <i>kFTM_Chnl5Flag</i>        | Channel 5 Flag.                               |
| <i>kFTM_Chnl6Flag</i>        | Channel 6 Flag.                               |
| <i>kFTM_Chnl7Flag</i>        | Channel 7 Flag.                               |
| <i>kFTM_FaultFlag</i>        | Fault Flag.                                   |
| <i>kFTM_TimeOverflowFlag</i> | Time overflow Flag.                           |
| <i>kFTM_ChnlTriggerFlag</i>  | Channel trigger Flag.                         |
| <i>kFTM_ReloadFlag</i>       | Reload Flag; Available only on certain SoC's. |

## 14.8 Function Documentation

### 14.8.1 status\_t FTM\_Init ( FTM\_Type \* *base*, const ftm\_config\_t \* *config* )

Note

This API should be called at the beginning of the application which is using the FTM driver. If the FTM instance has only TPM features, please use the TPM driver.

Parameters

|               |                                              |
|---------------|----------------------------------------------|
| <i>base</i>   | FTM peripheral base address                  |
| <i>config</i> | Pointer to the user configuration structure. |

Returns

*kStatus\_Success* indicates success; Else indicates failure.

### 14.8.2 void FTM\_Deinit ( FTM\_Type \* *base* )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

### 14.8.3 void FTM\_GetDefaultConfig ( *ftm\_config\_t* \* *config* )

The default values are:

```
* config->prescale = kFTM_Prescale_Divide_1;
* config->bdmMode = kFTM_BdmMode_0;
* config->pwmSyncMode = kFTM_SoftwareTrigger;
* config->reloadPoints = 0;
* config->faultMode = kFTM_Fault_Disable;
* config->faultFilterValue = 0;
* config->deadTimePrescale = kFTM_Deadtime_Prescale_1;
* config->deadTimeValue = 0;
* config->extTriggers = 0;
* config->chnlInitState = 0;
* config->chnlPolarity = 0;
* config->useGlobalTimeBase = false;
*
```

Parameters

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

### 14.8.4 static *ftm\_clock\_prescale\_t* FTM\_CalculateCounterClkDiv ( *FTM\_Type* \* *base*, *uint32\_t* *counterPeriod\_Hz*, *uint32\_t* *srcClock\_Hz* ) [inline], [static]

This function calculates the values for SC[PS] bit.

param *base* FTM peripheral base address param *counterPeriod\_Hz* The desired frequency in Hz which corresponding to the time when the counter reaches the mod value param *srcClock\_Hz* FTM counter clock in Hz

return Calculated clock prescaler value, see [ftm\\_clock\\_prescale\\_t](#).

### 14.8.5 *status\_t* FTM\_SetupPwm ( *FTM\_Type* \* *base*, *const ftm\_chnl\_pwm\_signal\_param\_t* \* *chnlParams*, *uint8\_t* *numOfChnls*, *ftm\_pwm\_mode\_t* *mode*, *uint32\_t* *pwmFreq\_Hz*, *uint32\_t* *srcClock\_Hz* )

Call this function to configure the PWM signal period, mode, duty cycle, and edge. Use this function to configure all FTM channels that are used to output a PWM signal.

Parameters

|                    |                                                                                     |
|--------------------|-------------------------------------------------------------------------------------|
| <i>base</i>        | FTM peripheral base address                                                         |
| <i>chnlParams</i>  | Array of PWM channel parameters to configure the channel(s)                         |
| <i>numOfChnls</i>  | Number of channels to configure; This should be the size of the array passed in     |
| <i>mode</i>        | PWM operation mode, options available in enumeration <a href="#">ftm_pwm_mode_t</a> |
| <i>pwmFreq_Hz</i>  | PWM signal frequency in Hz                                                          |
| <i>srcClock_Hz</i> | FTM counter clock in Hz                                                             |

Returns

kStatus\_Success if the PWM setup was successful kStatus\_Error on failure

#### 14.8.6 status\_t **FTM\_UpdatePwmDutycycle** ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlNumber*, **ftm\_pwm\_mode\_t** *currentPwmMode*, **uint8\_t** *dutyCyclePercent* )

Parameters

|                         |                                                                                                                                   |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>             | FTM peripheral base address                                                                                                       |
| <i>chnlNumber</i>       | The channel/channel pair number. In combined mode, this represents the channel pair number                                        |
| <i>currentPwmMode</i>   | The current PWM mode set during PWM setup                                                                                         |
| <i>dutyCyclePercent</i> | New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle) |

Returns

kStatus\_Success if the PWM update was successful kStatus\_Error on failure

#### 14.8.7 void **FTM\_UpdateChnlEdgeLevelSelect** ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlNumber*, **uint8\_t** *level* )

Parameters

|                   |                                                                                                                                                   |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | FTM peripheral base address                                                                                                                       |
| <i>chnlNumber</i> | The channel number                                                                                                                                |
| <i>level</i>      | The level to be set to the ELSnB:ELSnA field; Valid values are 00, 01, 10, 11. See the Kinetis SoC reference manual for details about this field. |

#### 14.8.8 **status\_t FTM\_SetupPwmMode ( FTM\_Type \* *base*, const ftm\_chnl\_pwm\_config\_param\_t \* *chnlParams*, uint8\_t *numOfChnls*, ftm\_pwm\_mode\_t *mode* )**

Call this function to configure the PWM signal mode, duty cycle in ticks, and edge. Use this function to configure all FTM channels that are used to output a PWM signal. Please note that: This API is similar with [FTM\\_SetupPwm\(\)](#) API, but will not set the timer period, and this API will set channel match value in timer ticks, not period percent.

Parameters

|                   |                                                                                     |
|-------------------|-------------------------------------------------------------------------------------|
| <i>base</i>       | FTM peripheral base address                                                         |
| <i>chnlParams</i> | Array of PWM channel parameters to configure the channel(s)                         |
| <i>numOfChnls</i> | Number of channels to configure; This should be the size of the array passed in     |
| <i>mode</i>       | PWM operation mode, options available in enumeration <a href="#">ftm_pwm_mode_t</a> |

Returns

kStatus\_Success if the PWM setup was successful kStatus\_Error on failure

#### 14.8.9 **void FTM\_SetupInputCapture ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, ftm\_input\_capture\_edge\_t *captureMode*, uint32\_t *filterValue* )**

When the edge specified in the captureMode argument occurs on the channel, the FTM counter is captured into the CnV register. The user has to read the CnV register separately to get this value. The filter function is disabled if the filterVal argument passed in is 0. The filter function is available only for channels 0, 1, 2, 3.

Parameters

|                    |                                                                             |
|--------------------|-----------------------------------------------------------------------------|
| <i>base</i>        | FTM peripheral base address                                                 |
| <i>chnlNumber</i>  | The channel number                                                          |
| <i>captureMode</i> | Specifies which edge to capture                                             |
| <i>filterValue</i> | Filter value, specify 0 to disable filter. Available only for channels 0-3. |

**14.8.10 void FTM\_SetupOutputCompare ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, ftm\_output\_compare\_mode\_t *compareMode*, uint32\_t *compareValue* )**

When the FTM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

|                     |                                                                        |
|---------------------|------------------------------------------------------------------------|
| <i>base</i>         | FTM peripheral base address                                            |
| <i>chnlNumber</i>   | The channel number                                                     |
| <i>compareMode</i>  | Action to take on the channel output when the compare condition is met |
| <i>compareValue</i> | Value to be programmed in the CnV register.                            |

**14.8.11 void FTM\_SetupDualEdgeCapture ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, const ftm\_dual\_edge\_capture\_param\_t \* *edgeParam*, uint32\_t *filterValue* )**

This function sets up the dual edge capture mode on a channel pair. The capture edge for the channel pair and the capture mode (one-shot or continuous) is specified in the parameter argument. The filter function is disabled if the filterVal argument passed is zero. The filter function is available only on channels 0 and 2. The user has to read the channel CnV registers separately to get the capture values.

Parameters

|                        |                                                     |
|------------------------|-----------------------------------------------------|
| <i>base</i>            | FTM peripheral base address                         |
| <i>chnlPair-Number</i> | The FTM channel pair number; options are 0, 1, 2, 3 |

|                    |                                                                                     |
|--------------------|-------------------------------------------------------------------------------------|
| <i>edgeParam</i>   | Sets up the dual edge capture function                                              |
| <i>filterValue</i> | Filter value, specify 0 to disable filter. Available only for channel pair 0 and 1. |

#### 14.8.12 void FTM\_SetupFaultInput ( **FTM\_Type** \* *base*, **ftm\_fault\_input\_t** *faultNumber*, **const ftm\_fault\_param\_t** \* *faultParams* )

FTM can have up to 4 fault inputs. This function sets up fault parameters, fault level, and input filter.

Parameters

|                    |                                          |
|--------------------|------------------------------------------|
| <i>base</i>        | FTM peripheral base address              |
| <i>faultNumber</i> | FTM fault to configure.                  |
| <i>faultParams</i> | Parameters passed in to set up the fault |

#### 14.8.13 void FTM\_EnableInterrupts ( **FTM\_Type** \* *base*, **uint32\_t** *mask* )

Parameters

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | FTM peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">ftm_interrupt_enable_t</a> |

#### 14.8.14 void FTM\_DisableInterrupts ( **FTM\_Type** \* *base*, **uint32\_t** *mask* )

Parameters

|             |                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | FTM peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">ftm_interrupt_enable_t</a> |

#### 14.8.15 **uint32\_t** FTM\_GetEnabledInterrupts ( **FTM\_Type** \* *base* )

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [ftm\\_interrupt\\_enable\\_t](#)

#### 14.8.16 `uint32_t FTM_GetStatusFlags ( FTM_Type * base )`

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

Returns

The status flags. This is the logical OR of members of the enumeration [ftm\\_status\\_flags\\_t](#)

#### 14.8.17 `void FTM_ClearStatusFlags ( FTM_Type * base, uint32_t mask )`

Parameters

|             |                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | FTM peripheral base address                                                                                      |
| <i>mask</i> | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">ftm_status_flags_t</a> |

#### 14.8.18 `static void FTM_SetTimerPeriod ( FTM_Type * base, uint32_t ticks ) [inline], [static]`

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note

1. This API allows the user to use the FTM module as a timer. Do not mix usage of this API with FTM's PWM setup API's.
2. Call the utility macros provided in the fsl\_common.h to convert usec or msec to ticks.

Parameters

|              |                                                                            |
|--------------|----------------------------------------------------------------------------|
| <i>base</i>  | FTM peripheral base address                                                |
| <i>ticks</i> | A timer period in units of ticks, which should be equal or greater than 1. |

#### 14.8.19 static uint32\_t FTM\_GetCurrentTimerCount ( **FTM\_Type** \* *base* ) [**inline**], [**static**]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note

Call the utility macros provided in the fsl\_common.h to convert ticks to usec or msec.

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

Returns

The current counter value in ticks

#### 14.8.20 static uint32\_t FTM\_GetInputCaptureValue ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlNumber* ) [**inline**], [**static**]

This function returns the captured value of a FTM channel configured in input capture or dual edge capture mode.

Note

Call the utility macros provided in the fsl\_common.h to convert ticks to usec or msec.

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

|                   |                    |
|-------------------|--------------------|
| <i>chnlNumber</i> | Channel to be read |
|-------------------|--------------------|

Returns

The captured FTM counter value of the input modes.

#### 14.8.21 static void FTM\_StartTimer ( **FTM\_Type** \* *base*, **ftm\_clock\_source\_t** *clockSource* ) [inline], [static]

Parameters

|                    |                                                                              |
|--------------------|------------------------------------------------------------------------------|
| <i>base</i>        | FTM peripheral base address                                                  |
| <i>clockSource</i> | FTM clock source; After the clock source is set, the counter starts running. |

#### 14.8.22 static void FTM\_StopTimer ( **FTM\_Type** \* *base* ) [inline], [static]

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

#### 14.8.23 static void FTM\_SetSoftwareCtrlEnable ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlNumber*, **bool** *value* ) [inline], [static]

Parameters

|                   |                                                                                                                            |
|-------------------|----------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | FTM peripheral base address                                                                                                |
| <i>chnlNumber</i> | Channel to be enabled or disabled                                                                                          |
| <i>value</i>      | true: channel output is affected by software output control false: channel output is unaffected by software output control |

#### 14.8.24 static void FTM\_SetSoftwareCtrlVal ( **FTM\_Type** \* *base*, **ftm\_chnl\_t** *chnlNumber*, **bool** *value* ) [inline], [static]

Parameters

|                   |                               |
|-------------------|-------------------------------|
| <i>base</i>       | FTM peripheral base address.  |
| <i>chnlNumber</i> | Channel to be configured      |
| <i>value</i>      | true to set 1, false to set 0 |

**14.8.25 static void FTM\_SetGlobalTimeBaseOutputEnable ( FTM\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

|               |                                  |
|---------------|----------------------------------|
| <i>base</i>   | FTM peripheral base address      |
| <i>enable</i> | true to enable, false to disable |

**14.8.26 static void FTM\_SetOutputMask ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, bool *mask* ) [inline], [static]**

Parameters

|                   |                                                                        |
|-------------------|------------------------------------------------------------------------|
| <i>base</i>       | FTM peripheral base address                                            |
| <i>chnlNumber</i> | Channel to be configured                                               |
| <i>mask</i>       | true: masked, channel is forced to its inactive state; false: unmasked |

**14.8.27 static void FTM\_SetPwmOutputEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, bool *value* ) [inline], [static]**

To enable the PWM channel output call this function with val=true. For input mode, call this function with val=false.

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | FTM peripheral base address |
|-------------|-----------------------------|

|                   |                                                                    |
|-------------------|--------------------------------------------------------------------|
| <i>chnlNumber</i> | Channel to be configured                                           |
| <i>value</i>      | true: enable output; false: output is disabled, used in input mode |

**14.8.28 static void FTM\_SetFaultControlEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

|                        |                                                                           |
|------------------------|---------------------------------------------------------------------------|
| <i>base</i>            | FTM peripheral base address                                               |
| <i>chnlPair-Number</i> | The FTM channel pair number; options are 0, 1, 2, 3                       |
| <i>value</i>           | true: Enable fault control for this channel pair; false: No fault control |

**14.8.29 static void FTM\_SetDeadTimeEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

|                        |                                                                           |
|------------------------|---------------------------------------------------------------------------|
| <i>base</i>            | FTM peripheral base address                                               |
| <i>chnlPair-Number</i> | The FTM channel pair number; options are 0, 1, 2, 3                       |
| <i>value</i>           | true: Insert dead time in this channel pair; false: No dead time inserted |

**14.8.30 static void FTM\_SetComplementaryEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

|                        |                                                     |
|------------------------|-----------------------------------------------------|
| <i>base</i>            | FTM peripheral base address                         |
| <i>chnlPair-Number</i> | The FTM channel pair number; options are 0, 1, 2, 3 |

|              |                                                                    |
|--------------|--------------------------------------------------------------------|
| <i>value</i> | true: enable complementary mode; false: disable complementary mode |
|--------------|--------------------------------------------------------------------|

**14.8.31 static void FTM\_SetInvertEnable ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlPairNumber*, bool *value* ) [inline], [static]**

Parameters

|                        |                                                     |
|------------------------|-----------------------------------------------------|
| <i>base</i>            | FTM peripheral base address                         |
| <i>chnlPair-Number</i> | The FTM channel pair number; options are 0, 1, 2, 3 |
| <i>value</i>           | true: enable inverting; false: disable inverting    |

**14.8.32 void FTM\_SetupQuadDecode ( FTM\_Type \* *base*, const ftm\_phase\_params\_t \* *phaseAParams*, const ftm\_phase\_params\_t \* *phaseBParams*, ftm\_quad\_decode\_mode\_t *quadMode* )**

Parameters

|                     |                                                       |
|---------------------|-------------------------------------------------------|
| <i>base</i>         | FTM peripheral base address                           |
| <i>phaseAParams</i> | Phase A configuration parameters                      |
| <i>phaseBParams</i> | Phase B configuration parameters                      |
| <i>quadMode</i>     | Selects encoding mode used in quadrature decoder mode |

**14.8.33 static void FTM\_SetQuadDecoderModuloValue ( FTM\_Type \* *base*, uint32\_t *startValue*, uint32\_t *overValue* ) [inline], [static]**

The modulo values configure the minimum and maximum values that the Quad decoder counter can reach. After the counter goes over, the counter value goes to the other side and decrease/increase again.

Parameters

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

|                   |                                                |
|-------------------|------------------------------------------------|
| <i>startValue</i> | The low limit value for Quad Decoder counter.  |
| <i>overValue</i>  | The high limit value for Quad Decoder counter. |

**14.8.34 static uint32\_t FTM\_GetQuadDecoderCounterValue ( FTM\_Type \* *base* )  
[inline], [static]**

Parameters

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

Returns

Current quad Decoder counter value.

**14.8.35 static void FTM\_ClearQuadDecoderCounterValue ( FTM\_Type \* *base* )  
[inline], [static]**

The counter is set as the initial value.

Parameters

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

**14.8.36 static void FTM\_SetSoftwareTrigger ( FTM\_Type \* *base*, bool *enable* )  
[inline], [static]**

Parameters

|               |                                                                             |
|---------------|-----------------------------------------------------------------------------|
| <i>base</i>   | FTM peripheral base address                                                 |
| <i>enable</i> | true: software trigger is selected, false: software trigger is not selected |

**14.8.37 static void FTM\_SetWriteProtection ( FTM\_Type \* *base*, bool *enable* )  
[inline], [static]**

Parameters

|               |                                                                        |
|---------------|------------------------------------------------------------------------|
| <i>base</i>   | FTM peripheral base address                                            |
| <i>enable</i> | true: Write-protection is enabled, false: Write-protection is disabled |

#### 14.8.38 static void FTM\_EnableDmaTransfer ( FTM\_Type \* *base*, ftm\_chnl\_t *chnlNumber*, bool *enable* ) [inline], [static]

Note: CHnIE bit needs to be set when calling this API. The channel DMA transfer request is generated and the channel interrupt is not generated if (CHnF = 1) when DMA and CHnIE bits are set.

Parameters

|                   |                                  |
|-------------------|----------------------------------|
| <i>base</i>       | FTM peripheral base address.     |
| <i>chnlNumber</i> | Channel to be configured         |
| <i>enable</i>     | true to enable, false to disable |

# Chapter 15

## GPIO: General-Purpose Input/Output Driver

### 15.1 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\(2, 6, 0\)\)](#)  
*GPIO driver version.*

### 15.2 Data Structure Documentation

#### 15.2.1 [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.*

## 15.3 Macro Definition Documentation

15.3.1 `#define FSL_GPIO_DRIVER_VERSION (MAKE_VERSION(2, 6, 0))`

## 15.4 Enumeration Type Documentation

15.4.1 `enum gpio_pin_direction_t`

Enumerator

*kGPIO\_DigitalInput* Set current pin as digital input.

*kGPIO\_DigitalOutput* Set current pin as digital output.

## 15.5 GPIO Driver

### 15.5.1 Overview

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

### 15.5.2 Typical use case

#### 15.5.2.1 Output Operation

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/gpio

#### 15.5.2.2 Input Operation

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/gpio

## 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\\_PinWrite](#) (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\\_PortSet](#) (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 1.*
- static void [GPIO\\_PortClear](#) (GPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple GPIO pins to the logic 0.*
- static void [GPIO\\_PortToggle](#) (GPIO\_Type \*base, uint32\_t mask)  
*Reverses the current output logic of the multiple GPIO pins.*

## GPIO Input Operations

- static uint32\_t [GPIO\\_PinRead](#) (GPIO\_Type \*base, uint32\_t pin)  
*Reads the current input value of the GPIO port.*

## GPIO Interrupt

- uint32\_t [GPIO\\_PortGetInterruptFlags](#) (GPIO\_Type \*base)  
*Reads the GPIO port interrupt status flag.*

- void [GPIO\\_PortClearInterruptFlags](#) (GPIO\_Type \*base, uint32\_t mask)  
*Clears multiple GPIO pin interrupt status flags.*

### 15.5.3 Function Documentation

#### 15.5.3.1 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                                 |

#### 15.5.3.2 static void [GPIO\\_PinWrite](#) ( **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> |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 15.5.3.3 static void GPIO\_PortSet ( **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                                          |

### 15.5.3.4 static void GPIO\_PortClear ( **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                                          |

### 15.5.3.5 static void GPIO\_PortToggle ( **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                                          |

### 15.5.3.6 static **uint32\_t** GPIO\_PinRead ( **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 <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> |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

#### 15.5.3.7 `uint32_t GPIO_PortGetInterruptFlags ( 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. |
|------------|-------------------------------------------------------------------------------------------------------------|

#### 15.5.3.8 `void GPIO_PortClearInterruptFlags ( 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                                          |

## 15.6 FGPIO Driver

### 15.6.1 Overview

This section 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.

### 15.6.2 Typical use case

#### 15.6.2.1 Output Operation

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/gpio

#### 15.6.2.2 Input Operation

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/gpio

## FGPIO Configuration

- void [FGPIO\\_PinInit](#) (FGPIO\_Type \*base, uint32\_t pin, const [gpio\\_pin\\_config\\_t](#) \*config)  
*Initializes a FGPIO pin used by the board.*

## FGPIO Output Operations

- static void [FGPIO\\_PinWrite](#) (FGPIO\_Type \*base, uint32\_t pin, uint8\_t output)  
*Sets the output level of the multiple FGPIO pins to the logic 1 or 0.*
- static void [FGPIO\\_PortSet](#) (FGPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple FGPIO pins to the logic 1.*
- static void [FGPIO\\_PortClear](#) (FGPIO\_Type \*base, uint32\_t mask)  
*Sets the output level of the multiple FGPIO pins to the logic 0.*
- static void [FGPIO\\_PortToggle](#) (FGPIO\_Type \*base, uint32\_t mask)  
*Reverses the current output logic of the multiple FGPIO pins.*

## FGPIO Input Operations

- static uint32\_t [FGPIO\\_PinRead](#) (FGPIO\_Type \*base, uint32\_t pin)  
*Reads the current input value of the FGPIO port.*

## GPIO Interrupt

- `uint32_t GPIO_PortGetInterruptFlags (GPIO_Type *base)`  
*Reads the GPIO port interrupt status flag.*
- `void GPIO_PortClearInterruptFlags (GPIO_Type *base, uint32_t mask)`  
*Clears the multiple GPIO pin interrupt status flag.*

### 15.6.3 Function Documentation

#### 15.6.3.1 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

|                     |                                                                |
|---------------------|----------------------------------------------------------------|
| <code>base</code>   | GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.) |
| <code>pin</code>    | GPIO port pin number                                           |
| <code>config</code> | GPIO pin configuration pointer                                 |

#### 15.6.3.2 static void GPIO\_PinWrite ( `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> |

### 15.6.3.3 static void GPIO\_PortSet ( **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                                          |

### 15.6.3.4 static void GPIO\_PortClear ( **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                                          |

### 15.6.3.5 static void GPIO\_PortToggle ( **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                                          |

### 15.6.3.6 static **uint32\_t** GPIO\_PinRead ( **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 <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> |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 15.6.3.7 `uint32_t GPIO_PortGetInterruptFlags ( 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. |
|------------|--------------------------------------------------------------------------------------------------------------|

### 15.6.3.8 `void GPIO_PortClearInterruptFlags ( 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 16

## LPI2C: Low Power Inter-Integrated Circuit Driver

### 16.1 Overview

#### Modules

- LPI2C CMSIS Driver
- LPI2C FreeRTOS Driver
- LPI2C Master DMA Driver
- LPI2C Master Driver
- LPI2C Slave Driver

#### Macros

- `#define I2C_RETRY_TIMES 0U /* Define to zero means keep waiting until the flag is assert/deassert. */`  
*Retry times for waiting flag.*

#### Enumerations

- `enum {  
 kStatus_LPI2C_Busy = MAKE_STATUS(kStatusGroup_LPI2C, 0),  
 kStatus_LPI2C_Idle = MAKE_STATUS(kStatusGroup_LPI2C, 1),  
 kStatus_LPI2C_Nak = MAKE_STATUS(kStatusGroup_LPI2C, 2),  
 kStatus_LPI2C_FifoError = MAKE_STATUS(kStatusGroup_LPI2C, 3),  
 kStatus_LPI2C_BitError = MAKE_STATUS(kStatusGroup_LPI2C, 4),  
 kStatus_LPI2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_LPI2C, 5),  
 kStatus_LPI2C_PinLowTimeout,  
 kStatus_LPI2C_NoTransferInProgress,  
 kStatus_LPI2C_DmaRequestFail = MAKE_STATUS(kStatusGroup_LPI2C, 8),  
 kStatus_LPI2C_Timeout = MAKE_STATUS(kStatusGroup_LPI2C, 9) }  
LPI2C status return codes.`

#### Driver version

- `#define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))`  
*LPI2C driver version.*

### 16.2 Macro Definition Documentation

#### 16.2.1 `#define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))`

**16.2.2 #define I2C\_RETRY\_TIMES 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/**

## 16.3 Enumeration Type Documentation

### 16.3.1 anonymous enum

Enumerator

*kStatus\_LPI2C\_Busy* The master is already performing a transfer.

*kStatus\_LPI2C\_Idle* The slave driver is idle.

*kStatus\_LPI2C\_Nak* The slave device sent a NAK in response to a byte.

*kStatus\_LPI2C\_FifoError* FIFO under run or overrun.

*kStatus\_LPI2C\_BitError* Transferred bit was not seen on the bus.

*kStatus\_LPI2C\_ArbitrationLost* Arbitration lost error.

*kStatus\_LPI2C\_PinLowTimeout* SCL or SDA were held low longer than the timeout.

*kStatus\_LPI2C\_NoTransferInProgress* Attempt to abort a transfer when one is not in progress.

*kStatus\_LPI2C\_DmaRequestFail* DMA request failed.

*kStatus\_LPI2C\_Timeout* Timeout polling status flags.

## 16.4 LPI2C Master Driver

### 16.4.1 Overview

#### Data Structures

- struct `lpi2c_master_config_t`  
*Structure with settings to initialize the LPI2C master module. [More...](#)*
- struct `lpi2c_data_match_config_t`  
*LPI2C master data match configuration structure. [More...](#)*
- struct `lpi2c_master_transfer_t`  
*Non-blocking transfer descriptor structure. [More...](#)*
- struct `lpi2c_master_handle_t`  
*Driver handle for master non-blocking APIs. [More...](#)*

#### Typedefs

- typedef void(\* `lpi2c_master_transfer_callback_t` )(LPI2C\_Type \*base, `lpi2c_master_handle_t` \*handle, `status_t` completionStatus, void \*userData)  
*Master completion callback function pointer type.*
- typedef void(\* `lpi2c_master_isr_t` )(LPI2C\_Type \*base, void \*handle)  
*Typedef for master interrupt handler, used internally for LPI2C master interrupt and EDMA transactional APIs.*

#### Enumerations

- enum `_lpi2c_master_flags` {
   
`kLPI2C_MasterTxReadyFlag` = LPI2C\_MSR\_TDF\_MASK,  
`kLPI2C_MasterRxReadyFlag` = LPI2C\_MSR\_RDF\_MASK,  
`kLPI2C_MasterEndOfPacketFlag` = LPI2C\_MSR\_EPF\_MASK,  
`kLPI2C_MasterStopDetectFlag` = LPI2C\_MSR\_SDF\_MASK,  
`kLPI2C_MasterNackDetectFlag` = LPI2C\_MSR\_NDF\_MASK,  
`kLPI2C_MasterArbitrationLostFlag` = LPI2C\_MSR\_ALF\_MASK,  
`kLPI2C_MasterFifoErrFlag` = LPI2C\_MSR\_FEF\_MASK,  
`kLPI2C_MasterPinLowTimeoutFlag` = LPI2C\_MSR\_PLTF\_MASK,  
`kLPI2C_MasterDataMatchFlag` = LPI2C\_MSR\_DMF\_MASK,  
`kLPI2C_MasterBusyFlag` = LPI2C\_MSR\_MBF\_MASK,  
`kLPI2C_MasterBusBusyFlag` = LPI2C\_MSR\_BBF\_MASK,  
`kLPI2C_MasterClearFlags`,  
`kLPI2C_MasterIrqFlags`,  
`kLPI2C_MasterErrorFlags` }
   
*LPI2C master peripheral flags.*
- enum `lpi2c_direction_t` {
   
`kLPI2C_Write` = 0U,  
`kLPI2C_Read` = 1U }

- *Direction of master and slave transfers.*
- enum `lpi2c_master_pin_config_t` {
   
  `kLPI2C_2PinOpenDrain` = 0x0U,
   
  `kLPI2C_2PinOutputOnly` = 0x1U,
   
  `kLPI2C_2PinPushPull` = 0x2U,
   
  `kLPI2C_4PinPushPull` = 0x3U,
   
  `kLPI2C_2PinOpenDrainWithSeparateSlave`,
   
  `kLPI2C_2PinOutputOnlyWithSeparateSlave`,
   
  `kLPI2C_2PinPushPullWithSeparateSlave`,
   
  `kLPI2C_4PinPushPullWithInvertedOutput` = 0x7U }
- LPI2C pin configuration.*
- enum `lpi2c_host_request_source_t` {
   
  `kLPI2C_HostRequestExternalPin` = 0x0U,
   
  `kLPI2C_HostRequestInputTrigger` = 0x1U }
- LPI2C master host request selection.*
- enum `lpi2c_host_request_polarity_t` {
   
  `kLPI2C_HostRequestPinActiveLow` = 0x0U,
   
  `kLPI2C_HostRequestPinActiveHigh` = 0x1U }
- LPI2C master host request pin polarity configuration.*
- enum `lpi2c_data_match_config_mode_t` {
   
  `kLPI2C_MatchDisabled` = 0x0U,
   
  `kLPI2C_1stWordEqualsM0OrM1` = 0x2U,
   
  `kLPI2C_AnyWordEqualsM0OrM1` = 0x3U,
   
  `kLPI2C_1stWordEqualsM0And2ndWordEqualsM1`,
   
  `kLPI2C_AnyWordEqualsM0AndNextWordEqualsM1`,
   
  `kLPI2C_1stWordAndM1EqualsM0AndM1`,
   
  `kLPI2C_AnyWordAndM1EqualsM0AndM1` }
- LPI2C master data match configuration modes.*
- enum `_lpi2c_master_transfer_flags` {
   
  `kLPI2C_TransferDefaultFlag` = 0x00U,
   
  `kLPI2C_TransferNoStartFlag` = 0x01U,
   
  `kLPI2C_TransferRepeatedStartFlag` = 0x02U,
   
  `kLPI2C_TransferNoStopFlag` = 0x04U }
- Transfer option flags.*

## Initialization and deinitialization

- void `LPI2C_MasterGetDefaultConfig` (`lpi2c_master_config_t` \*`masterConfig`)
   
    *Provides a default configuration for the LPI2C master peripheral.*
- void `LPI2C_MasterInit` (`LPI2C_Type` \*`base`, const `lpi2c_master_config_t` \*`masterConfig`, `uint32_t` `sourceClock_Hz`)
   
    *Initializes the LPI2C master peripheral.*
- void `LPI2C_MasterDeinit` (`LPI2C_Type` \*`base`)
   
    *Deinitializes the LPI2C master peripheral.*
- void `LPI2C_MasterConfigureDataMatch` (`LPI2C_Type` \*`base`, const `lpi2c_data_match_config_t` \*`matchConfig`)

*Configures LPI2C master data match feature.*

- **status\_t LPI2C\_MasterCheckAndClearError** (LPI2C\_Type \*base, uint32\_t status)
- **status\_t LPI2C\_CheckForBusyBus** (LPI2C\_Type \*base)
- static void **LPI2C\_MasterReset** (LPI2C\_Type \*base)

*Performs a software reset.*

- static void **LPI2C\_MasterEnable** (LPI2C\_Type \*base, bool enable)
- Enables or disables the LPI2C module as master.*

## Status

- static uint32\_t **LPI2C\_MasterGetStatusFlags** (LPI2C\_Type \*base)  
*Gets the LPI2C master status flags.*
- static void **LPI2C\_MasterClearStatusFlags** (LPI2C\_Type \*base, uint32\_t statusMask)  
*Clears the LPI2C master status flag state.*

## Interrupts

- static void **LPI2C\_MasterEnableInterrupts** (LPI2C\_Type \*base, uint32\_t interruptMask)  
*Enables the LPI2C master interrupt requests.*
- static void **LPI2C\_MasterDisableInterrupts** (LPI2C\_Type \*base, uint32\_t interruptMask)  
*Disables the LPI2C master interrupt requests.*
- static uint32\_t **LPI2C\_MasterGetEnabledInterrupts** (LPI2C\_Type \*base)  
*Returns the set of currently enabled LPI2C master interrupt requests.*

## DMA control

- static void **LPI2C\_MasterEnableDMA** (LPI2C\_Type \*base, bool enableTx, bool enableRx)  
*Enables or disables LPI2C master DMA requests.*
- static uint32\_t **LPI2C\_MasterGetTxFifoAddress** (LPI2C\_Type \*base)  
*Gets LPI2C master transmit data register address for DMA transfer.*
- static uint32\_t **LPI2C\_MasterGetRxFifoAddress** (LPI2C\_Type \*base)  
*Gets LPI2C master receive data register address for DMA transfer.*

## FIFO control

- static void **LPI2C\_MasterSetWatermarks** (LPI2C\_Type \*base, size\_t txWords, size\_t rxWords)  
*Sets the watermarks for LPI2C master FIFOs.*
- static void **LPI2C\_MasterGetFifoCounts** (LPI2C\_Type \*base, size\_t \*rxCount, size\_t \*txCount)  
*Gets the current number of words in the LPI2C master FIFOs.*

## Bus operations

- void **LPI2C\_MasterSetBaudRate** (LPI2C\_Type \*base, uint32\_t sourceClock\_Hz, uint32\_t baudRate\_Hz)

- Sets the I2C bus frequency for master transactions.
- static bool [LPI2C\\_MasterGetBusIdleState](#) (LPI2C\_Type \*base)  
Returns whether the bus is idle.
- [status\\_t LPI2C\\_MasterStart](#) (LPI2C\_Type \*base, uint8\_t address, [lpi2c\\_direction\\_t](#) dir)  
Sends a START signal and slave address on the I2C bus.
- static [status\\_t LPI2C\\_MasterRepeatedStart](#) (LPI2C\_Type \*base, uint8\_t address, [lpi2c\\_direction\\_t](#) dir)  
Sends a repeated START signal and slave address on the I2C bus.
- [status\\_t LPI2C\\_MasterSend](#) (LPI2C\_Type \*base, void \*txBuff, size\_t txSize)  
Performs a polling send transfer on the I2C bus.
- [status\\_t LPI2C\\_MasterReceive](#) (LPI2C\_Type \*base, void \*rxBuff, size\_t rxSize)  
Performs a polling receive transfer on the I2C bus.
- [status\\_t LPI2C\\_MasterStop](#) (LPI2C\_Type \*base)  
Sends a STOP signal on the I2C bus.
- [status\\_t LPI2C\\_MasterTransferBlocking](#) (LPI2C\_Type \*base, [lpi2c\\_master\\_transfer\\_t](#) \*transfer)  
Performs a master polling transfer on the I2C bus.

## Non-blocking

- void [LPI2C\\_MasterTransferCreateHandle](#) (LPI2C\_Type \*base, [lpi2c\\_master\\_handle\\_t](#) \*handle, [lpi2c\\_master\\_transfer\\_callback\\_t](#) callback, void \*userData)  
Creates a new handle for the LPI2C master non-blocking APIs.
- [status\\_t LPI2C\\_MasterTransferNonBlocking](#) (LPI2C\_Type \*base, [lpi2c\\_master\\_handle\\_t](#) \*handle, [lpi2c\\_master\\_transfer\\_t](#) \*transfer)  
Performs a non-blocking transaction on the I2C bus.
- [status\\_t LPI2C\\_MasterTransferGetCount](#) (LPI2C\_Type \*base, [lpi2c\\_master\\_handle\\_t](#) \*handle, size\_t \*count)  
Returns number of bytes transferred so far.
- void [LPI2C\\_MasterTransferAbort](#) (LPI2C\_Type \*base, [lpi2c\\_master\\_handle\\_t](#) \*handle)  
Terminates a non-blocking LPI2C master transmission early.

## IRQ handler

- void [LPI2C\\_MasterTransferHandleIRQ](#) (LPI2C\_Type \*base, void \*lpi2cMasterHandle)  
Reusable routine to handle master interrupts.

### 16.4.2 Data Structure Documentation

#### 16.4.2.1 struct lpi2c\_master\_config\_t

This structure holds configuration settings for the LPI2C peripheral. To initialize this structure to reasonable defaults, call the [LPI2C\\_MasterGetDefaultConfig\(\)](#) function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

## Data Fields

- bool `enableMaster`  
*Whether to enable master mode.*
- bool `enableDoze`  
*Whether master is enabled in doze mode.*
- bool `debugEnable`  
*Enable transfers to continue when halted in debug mode.*
- bool `ignoreAck`  
*Whether to ignore ACK/NACK.*
- `lpi2c_master_pin_config_t pinConfig`  
*The pin configuration option.*
- `uint32_t baudRate_Hz`  
*Desired baud rate in Hertz.*
- `uint32_t busIdleTimeout_ns`  
*Bus idle timeout in nanoseconds.*
- `uint32_t pinLowTimeout_ns`  
*Pin low timeout in nanoseconds.*
- `uint8_t sdaGlitchFilterWidth_ns`  
*Width in nanoseconds of glitch filter on SDA pin.*
- `uint8_t sclGlitchFilterWidth_ns`  
*Width in nanoseconds of glitch filter on SCL pin.*
- struct {
  - bool `enable`  
*Enable host request.*
  - `lpi2c_host_request_source_t source`  
*Host request source.*
  - `lpi2c_host_request_polarity_t polarity`  
*Host request pin polarity.*
} `hostRequest`  
*Host request options.*

## Field Documentation

- (1) `bool lpi2c_master_config_t::enableMaster`
- (2) `bool lpi2c_master_config_t::enableDoze`
- (3) `bool lpi2c_master_config_t::debugEnable`
- (4) `bool lpi2c_master_config_t::ignoreAck`
- (5) `lpi2c_master_pin_config_t lpi2c_master_config_t::pinConfig`
- (6) `uint32_t lpi2c_master_config_t::baudRate_Hz`
- (7) `uint32_t lpi2c_master_config_t::busIdleTimeout_ns`

Set to 0 to disable.

(8) `uint32_t lpi2c_master_config_t::pinLowTimeout_ns`

Set to 0 to disable.

(9) `uint8_t lpi2c_master_config_t::sdaGlitchFilterWidth_ns`

Set to 0 to disable.

(10) `uint8_t lpi2c_master_config_t::sclGlitchFilterWidth_ns`

Set to 0 to disable.

(11) `bool lpi2c_master_config_t::enable`

(12) `lpi2c_host_request_source_t lpi2c_master_config_t::source`

(13) `lpi2c_host_request_polarity_t lpi2c_master_config_t::polarity`

(14) `struct { ... } lpi2c_master_config_t::hostRequest`

#### 16.4.2.2 struct lpi2c\_data\_match\_config\_t

##### Data Fields

- `lpi2c_data_match_config_mode_t matchMode`  
*Data match configuration setting.*
- `bool rxDataMatchOnly`  
*When set to true, received data is ignored until a successful match.*
- `uint32_t match0`  
*Match value 0.*
- `uint32_t match1`  
*Match value 1.*

##### Field Documentation

(1) `lpi2c_data_match_config_mode_t lpi2c_data_match_config_t::matchMode`

(2) `bool lpi2c_data_match_config_t::rxDataMatchOnly`

(3) `uint32_t lpi2c_data_match_config_t::match0`

(4) `uint32_t lpi2c_data_match_config_t::match1`

#### 16.4.2.3 struct \_lpi2c\_master\_transfer

This structure is used to pass transaction parameters to the [LPI2C\\_MasterTransferNonBlocking\(\)](#) API.

##### Data Fields

- `uint32_t flags`

- **uint16\_t slaveAddress**  
*The 7-bit slave address.*
- **lpi2c\_direction\_t direction**  
*Either `kLPI2C_Read` or `kLPI2C_Write`.*
- **uint32\_t subaddress**  
*Sub address.*
- **size\_t subaddressSize**  
*Length of sub address to send in bytes.*
- **void \* data**  
*Pointer to data to transfer.*
- **size\_t dataSize**  
*Number of bytes to transfer.*

## Field Documentation

(1) **uint32\_t lpi2c\_master\_transfer\_t::flags**

See enumeration `_lpi2c_master_transfer_flags` for available options. Set to 0 or `kLPI2C_TransferDefaultFlag` for normal transfers.

(2) **uint16\_t lpi2c\_master\_transfer\_t::slaveAddress**

(3) **lpi2c\_direction\_t lpi2c\_master\_transfer\_t::direction**

(4) **uint32\_t lpi2c\_master\_transfer\_t::subaddress**

Transferred MSB first.

(5) **size\_t lpi2c\_master\_transfer\_t::subaddressSize**

Maximum size is 4 bytes.

(6) **void\* lpi2c\_master\_transfer\_t::data**

(7) **size\_t lpi2c\_master\_transfer\_t::dataSize**

### 16.4.2.4 struct \_lpi2c\_master\_handle

Note

The contents of this structure are private and subject to change.

## Data Fields

- **uint8\_t state**  
*Transfer state machine current state.*
- **uint16\_t remainingBytes**  
*Remaining byte count in current state.*
- **uint8\_t \* buf**

- *Buffer pointer for current state.*
- `uint16_t commandBuffer[6]`  
*LPI2C command sequence.*
- `lpi2c_master_transfer_t transfer`  
*Copy of the current transfer info.*
- `lpi2c_master_transfer_callback_t completionCallback`  
*Callback function pointer.*
- `void *userData`  
*Application data passed to callback.*

## Field Documentation

- (1) `uint8_t lpi2c_master_handle_t::state`
- (2) `uint16_t lpi2c_master_handle_t::remainingBytes`
- (3) `uint8_t* lpi2c_master_handle_t::buf`
- (4) `uint16_t lpi2c_master_handle_t::commandBuffer[6]`

When all 6 command words are used: Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word]

- (5) `lpi2c_master_transfer_t lpi2c_master_handle_t::transfer`
- (6) `lpi2c_master_transfer_callback_t lpi2c_master_handle_t::completionCallback`
- (7) `void* lpi2c_master_handle_t::userData`

## 16.4.3 Typedef Documentation

### 16.4.3.1 `typedef void(* lpi2c_master_transfer_callback_t)(LPI2C_Type *base, lpi2c_master_handle_t *handle, status_t completionStatus, void *userData)`

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to [LPI2C\\_MasterTransferCreateHandle\(\)](#).

Parameters

|                                |                                                                                |
|--------------------------------|--------------------------------------------------------------------------------|
| <code>base</code>              | The LPI2C peripheral base address.                                             |
| <code>completion-Status</code> | Either kStatus_Success or an error code describing how the transfer completed. |

|                 |                                                            |
|-----------------|------------------------------------------------------------|
| <i>userData</i> | Arbitrary pointer-sized value passed from the application. |
|-----------------|------------------------------------------------------------|

## 16.4.4 Enumeration Type Documentation

### 16.4.4.1 enum \_lpi2c\_master\_flags

The following status register flags can be cleared:

- [kLPI2C\\_MasterEndOfPacketFlag](#)
- [kLPI2C\\_MasterStopDetectFlag](#)
- [kLPI2C\\_MasterNackDetectFlag](#)
- [kLPI2C\\_MasterArbitrationLostFlag](#)
- [kLPI2C\\_MasterFifoErrFlag](#)
- [kLPI2C\\_MasterPinLowTimeoutFlag](#)
- [kLPI2C\\_MasterDataMatchFlag](#)

All flags except [kLPI2C\\_MasterBusyFlag](#) and [kLPI2C\\_MasterBusBusyFlag](#) can be enabled as interrupts.

Note

These enums are meant to be OR'd together to form a bit mask.

Enumerator

- kLPI2C\_MasterTxReadyFlag* Transmit data flag.
- kLPI2C\_MasterRxReadyFlag* Receive data flag.
- kLPI2C\_MasterEndOfPacketFlag* End Packet flag.
- kLPI2C\_MasterStopDetectFlag* Stop detect flag.
- kLPI2C\_MasterNackDetectFlag* NACK detect flag.
- kLPI2C\_MasterArbitrationLostFlag* Arbitration lost flag.
- kLPI2C\_MasterFifoErrFlag* FIFO error flag.
- kLPI2C\_MasterPinLowTimeoutFlag* Pin low timeout flag.
- kLPI2C\_MasterDataMatchFlag* Data match flag.
- kLPI2C\_MasterBusyFlag* Master busy flag.
- kLPI2C\_MasterBusBusyFlag* Bus busy flag.
- kLPI2C\_MasterClearFlags* All flags which are cleared by the driver upon starting a transfer.
- kLPI2C\_MasterIrqFlags* IRQ sources enabled by the non-blocking transactional API.
- kLPI2C\_MasterErrorFlags* Errors to check for.

### 16.4.4.2 enum lpi2c\_direction\_t

Enumerator

- kLPI2C\_Write* Master transmit.
- kLPI2C\_Read* Master receive.

#### 16.4.4.3 enum lpi2c\_master\_pin\_config\_t

Enumerator

***kLPI2C\_2PinOpenDrain*** LPI2C Configured for 2-pin open drain mode.

***kLPI2C\_2PinOutputOnly*** LPI2C Configured for 2-pin output only mode (ultra-fast mode)

***kLPI2C\_2PinPushPull*** LPI2C Configured for 2-pin push-pull mode.

***kLPI2C\_4PinPushPull*** LPI2C Configured for 4-pin push-pull mode.

***kLPI2C\_2PinOpenDrainWithSeparateSlave*** LPI2C Configured for 2-pin open drain mode with separate LPI2C slave.

***kLPI2C\_2PinOutputOnlyWithSeparateSlave*** LPI2C Configured for 2-pin output only mode(ultra-fast mode) with separate LPI2C slave.

***kLPI2C\_2PinPushPullWithSeparateSlave*** LPI2C Configured for 2-pin push-pull mode with separate LPI2C slave.

***kLPI2C\_4PinPushPullWithInvertedOutput*** LPI2C Configured for 4-pin push-pull mode(inverted outputs)

#### 16.4.4.4 enum lpi2c\_host\_request\_source\_t

Enumerator

***kLPI2C\_HostRequestExternalPin*** Select the LPI2C\_HREQ pin as the host request input.

***kLPI2C\_HostRequestInputTrigger*** Select the input trigger as the host request input.

#### 16.4.4.5 enum lpi2c\_host\_request\_polarity\_t

Enumerator

***kLPI2C\_HostRequestPinActiveLow*** Configure the LPI2C\_HREQ pin active low.

***kLPI2C\_HostRequestPinActiveHigh*** Configure the LPI2C\_HREQ pin active high.

#### 16.4.4.6 enum lpi2c\_data\_match\_config\_mode\_t

Enumerator

***kLPI2C\_MatchDisabled*** LPI2C Match Disabled.

***kLPI2C\_1stWordEqualsM0OrM1*** LPI2C Match Enabled and 1st data word equals MATCH0 OR MATCH1.

***kLPI2C\_AnyWordEqualsM0OrM1*** LPI2C Match Enabled and any data word equals MATCH0 OR MATCH1.

***kLPI2C\_1stWordEqualsM0And2ndWordEqualsM1*** LPI2C Match Enabled and 1st data word equals MATCH0, 2nd data equals MATCH1.

***kLPI2C\_AnyWordEqualsM0AndNextWordEqualsM1*** LPI2C Match Enabled and any data word equals MATCH0, next data equals MATCH1.

***kLPI2C\_1stWordAndM1EqualsM0AndM1*** LPI2C Match Enabled and 1st data word and MATCH0 equals MATCH0 and MATCH1.

***kLPI2C\_AnyWordAndM1EqualsM0AndM1*** LPI2C Match Enabled and any data word and MATCH0 equals MATCH0 and MATCH1.

#### 16.4.4.7 enum \_lpi2c\_master\_transfer\_flags

Note

These enumerations are intended to be OR'd together to form a bit mask of options for the `_lpi2c_master_transfer::flags` field.

Enumerator

***kLPI2C\_TransferDefaultFlag*** Transfer starts with a start signal, stops with a stop signal.

***kLPI2C\_TransferNoStartFlag*** Don't send a start condition, address, and sub address.

***kLPI2C\_TransferRepeatedStartFlag*** Send a repeated start condition.

***kLPI2C\_TransferNoStopFlag*** Don't send a stop condition.

#### 16.4.5 Function Documentation

##### 16.4.5.1 void LPI2C\_MasterGetDefaultConfig ( `lpi2c_master_config_t * masterConfig` )

This function provides the following default configuration for the LPI2C master peripheral:

```
* masterConfig->enableMaster = true;
* masterConfig->debugEnable = false;
* masterConfig->ignoreAck = false;
* masterConfig->pinConfig = kLPI2C_2PinOpenDrain;
* masterConfig->baudRate_Hz = 100000U;
* masterConfig->busIdleTimeout_ns = 0;
* masterConfig->pinLowTimeout_ns = 0;
* masterConfig->sdaGlitchFilterWidth_ns = 0;
* masterConfig->sclGlitchFilterWidth_ns = 0;
* masterConfig->hostRequest.enable = false;
* masterConfig->hostRequest.source = kLPI2C_HostRequestExternalPin;
* masterConfig->hostRequest.polarity = kLPI2C_HostRequestPinActiveHigh;
*
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with [LPI2C\\_MasterInit\(\)](#).

Parameters

|     |                     |                                                                                                            |
|-----|---------------------|------------------------------------------------------------------------------------------------------------|
| out | <i>masterConfig</i> | User provided configuration structure for default values. Refer to <a href="#">lpi2c_master_config_t</a> . |
|-----|---------------------|------------------------------------------------------------------------------------------------------------|

#### 16.4.5.2 void LPI2C\_MasterInit ( LPI2C\_Type \* *base*, const lpi2c\_master\_config\_t \* *masterConfig*, uint32\_t *sourceClock\_Hz* )

This function enables the peripheral clock and initializes the LPI2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

|                       |                                                                                                                                            |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>           | The LPI2C peripheral base address.                                                                                                         |
| <i>masterConfig</i>   | User provided peripheral configuration. Use <a href="#">LPI2C_MasterGetDefaultConfig()</a> to get a set of defaults that you can override. |
| <i>sourceClock_Hz</i> | Frequency in Hertz of the LPI2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.            |

#### 16.4.5.3 void LPI2C\_MasterDeinit ( LPI2C\_Type \* *base* )

This function disables the LPI2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

#### 16.4.5.4 void LPI2C\_MasterConfigureDataMatch ( LPI2C\_Type \* *base*, const lpi2c\_data\_match\_config\_t \* *matchConfig* )

Parameters

|                    |                                      |
|--------------------|--------------------------------------|
| <i>base</i>        | The LPI2C peripheral base address.   |
| <i>matchConfig</i> | Settings for the data match feature. |

#### 16.4.5.5 static void LPI2C\_MasterReset ( LPI2C\_Type \* *base* ) [inline], [static]

Restores the LPI2C master peripheral to reset conditions.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

#### 16.4.5.6 static void LPI2C\_MasterEnable( LPI2C\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

|               |                                                                        |
|---------------|------------------------------------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.                                     |
| <i>enable</i> | Pass true to enable or false to disable the specified LPI2C as master. |

#### 16.4.5.7 static uint32\_t LPI2C\_MasterGetStatusFlags( LPI2C\_Type \* *base* ) [inline], [static]

A bit mask with the state of all LPI2C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

See Also

[\\_lpi2c\\_master\\_flags](#)

#### 16.4.5.8 static void LPI2C\_MasterClearStatusFlags( LPI2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared:

- [kLPI2C\\_MasterEndOfPacketFlag](#)
- [kLPI2C\\_MasterStopDetectFlag](#)
- [kLPI2C\\_MasterNackDetectFlag](#)
- [kLPI2C\\_MasterArbitrationLostFlag](#)

- `kLPI2C_MasterFifoErrFlag`
- `kLPI2C_MasterPinLowTimeoutFlag`
- `kLPI2C_MasterDataMatchFlag`

Attempts to clear other flags has no effect.

Parameters

|                   |                                                                                                                                                                                                                                    |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | The LPI2C peripheral base address.                                                                                                                                                                                                 |
| <i>statusMask</i> | A bitmask of status flags that are to be cleared. The mask is composed of <code>_lpi2c_master_flags</code> enumerators OR'd together. You may pass the result of a previous call to <a href="#">LPI2C_MasterGetStatusFlags()</a> . |

See Also

[\\_lpi2c\\_master\\_flags](#).

#### 16.4.5.9 static void LPI2C\_MasterEnableInterrupts ( `LPI2C_Type * base`, `uint32_t interruptMask` ) [inline], [static]

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be enabled as interrupts.

Parameters

|                      |                                                                                                                                                    |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>          | The LPI2C peripheral base address.                                                                                                                 |
| <i>interruptMask</i> | Bit mask of interrupts to enable. See <code>_lpi2c_master_flags</code> for the set of constants that should be OR'd together to form the bit mask. |

#### 16.4.5.10 static void LPI2C\_MasterDisableInterrupts ( `LPI2C_Type * base`, `uint32_t interruptMask` ) [inline], [static]

All flags except `kLPI2C_MasterBusyFlag` and `kLPI2C_MasterBusBusyFlag` can be disabled as interrupts.

Parameters

|                      |                                                                                                                                                     |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>          | The LPI2C peripheral base address.                                                                                                                  |
| <i>interruptMask</i> | Bit mask of interrupts to disable. See <code>_lpi2c_master_flags</code> for the set of constants that should be OR'd together to form the bit mask. |

#### 16.4.5.11 static uint32\_t LPI2C\_MasterGetEnabledInterrupts ( `LPI2C_Type * base` ) [inline], [static]

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Returns

A bitmask composed of \_lpi2c\_master\_flags enumerators OR'd together to indicate the set of enabled interrupts.

#### 16.4.5.12 static void LPI2C\_MasterEnableDMA ( LPI2C\_Type \* *base*, bool *enableTx*, bool *enableRx* ) [inline], [static]

Parameters

|                 |                                                                                |
|-----------------|--------------------------------------------------------------------------------|
| <i>base</i>     | The LPI2C peripheral base address.                                             |
| <i>enableTx</i> | Enable flag for transmit DMA request. Pass true for enable, false for disable. |
| <i>enableRx</i> | Enable flag for receive DMA request. Pass true for enable, false for disable.  |

#### 16.4.5.13 static uint32\_t LPI2C\_MasterGetTxFifoAddress ( LPI2C\_Type \* *base* ) [inline], [static]

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Returns

The LPI2C Master Transmit Data Register address.

#### 16.4.5.14 static uint32\_t LPI2C\_MasterGetRxFifoAddress ( LPI2C\_Type \* *base* ) [inline], [static]

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Returns

The LPI2C Master Receive Data Register address.

#### 16.4.5.15 static void LPI2C\_MasterSetWatermarks ( *LPI2C\_Type* \* *base*, *size\_t* *txWords*, *size\_t* *rxWords* ) [inline], [static]

Parameters

|                |                                                                                                                                                                                                                                                             |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | The LPI2C peripheral base address.                                                                                                                                                                                                                          |
| <i>txWords</i> | Transmit FIFO watermark value in words. The <a href="#">kLPI2C_MasterTxReadyFlag</a> flag is set whenever the number of words in the transmit FIFO is equal or less than <i>txWords</i> . Writing a value equal or greater than the FIFO size is truncated. |
| <i>rxWords</i> | Receive FIFO watermark value in words. The <a href="#">kLPI2C_MasterRxReadyFlag</a> flag is set whenever the number of words in the receive FIFO is greater than <i>rxWords</i> . Writing a value equal or greater than the FIFO size is truncated.         |

#### 16.4.5.16 static void LPI2C\_MasterGetFifoCounts ( *LPI2C\_Type* \* *base*, *size\_t* \* *rxCount*, *size\_t* \* *txCount* ) [inline], [static]

Parameters

|     |                |                                                                                                                              |
|-----|----------------|------------------------------------------------------------------------------------------------------------------------------|
|     | <i>base</i>    | The LPI2C peripheral base address.                                                                                           |
| out | <i>txCount</i> | Pointer through which the current number of words in the transmit FIFO is returned. Pass NULL if this value is not required. |
| out | <i>rxCount</i> | Pointer through which the current number of words in the receive FIFO is returned. Pass NULL if this value is not required.  |

#### 16.4.5.17 void LPI2C\_MasterSetBaudRate ( *LPI2C\_Type* \* *base*, *uint32\_t* *sourceClock\_Hz*, *uint32\_t* *baudRate\_Hz* )

The LPI2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

## Note

Please note that the second parameter is the clock frequency of LPI2C module, the third parameter means user configured bus baudrate, this implementation is different from other I2C drivers which use baudrate configuration as second parameter and source clock frequency as third parameter.

Parameters

|                       |                                            |
|-----------------------|--------------------------------------------|
| <i>base</i>           | The LPI2C peripheral base address.         |
| <i>sourceClock_Hz</i> | LPI2C functional clock frequency in Hertz. |
| <i>baudRate_Hz</i>    | Requested bus frequency in Hertz.          |

#### 16.4.5.18 static bool LPI2C\_MasterGetBusIdleState ( LPI2C\_Type \* *base* ) [inline], [static]

Requires the master mode to be enabled.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Return values

|              |              |
|--------------|--------------|
| <i>true</i>  | Bus is busy. |
| <i>false</i> | Bus is idle. |

#### 16.4.5.19 status\_t LPI2C\_MasterStart ( LPI2C\_Type \* *base*, uint8\_t *address*, lpi2c\_direction\_t *dir* )

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

|                |                                                                                                                                                                                     |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | The LPI2C peripheral base address.                                                                                                                                                  |
| <i>address</i> | 7-bit slave device address, in bits [6:0].                                                                                                                                          |
| <i>dir</i>     | Master transfer direction, either <a href="#">kLPI2C_Read</a> or <a href="#">kLPI2C_Write</a> . This parameter is used to set the R/w bit (bit 0) in the transmitted slave address. |

Return values

|                           |                                                                           |
|---------------------------|---------------------------------------------------------------------------|
| <i>kStatus_Success</i>    | START signal and address were successfully enqueued in the transmit FIFO. |
| <i>kStatus_LPI2C_Busy</i> | Another master is currently utilizing the bus.                            |

#### 16.4.5.20 static status\_t LPI2C\_MasterRepeatedStart ( LPI2C\_Type \* *base*, uint8\_t *address*, lpi2c\_direction\_t *dir* ) [inline], [static]

This function is used to send a Repeated START signal when a transfer is already in progress. Like [LPI2C\\_MasterStart\(\)](#), it also sends the specified 7-bit address.

Note

This function exists primarily to maintain compatible APIs between LPI2C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

|                |                                                                                                                                                                                     |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>    | The LPI2C peripheral base address.                                                                                                                                                  |
| <i>address</i> | 7-bit slave device address, in bits [6:0].                                                                                                                                          |
| <i>dir</i>     | Master transfer direction, either <a href="#">kLPI2C_Read</a> or <a href="#">kLPI2C_Write</a> . This parameter is used to set the R/w bit (bit 0) in the transmitted slave address. |

Return values

|                           |                                                                                    |
|---------------------------|------------------------------------------------------------------------------------|
| <i>kStatus_Success</i>    | Repeated START signal and address were successfully enqueued in the transmit FIFO. |
| <i>kStatus_LPI2C_Busy</i> | Another master is currently utilizing the bus.                                     |

#### 16.4.5.21 status\_t LPI2C\_MasterSend ( LPI2C\_Type \* *base*, void \* *txBuff*, size\_t *txSize* )

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns [kStatus\\_LPI2C\\_Nak](#).

Parameters

|               |                                                    |
|---------------|----------------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.                 |
| <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>               | Data was sent successfully.                        |
| <i>kStatus_LPI2C_Busy</i>            | Another master is currently utilizing the bus.     |
| <i>kStatus_LPI2C_Nak</i>             | The slave device sent a NAK in response to a byte. |
| <i>kStatus_LPI2C_FifoError</i>       | FIFO under run or over run.                        |
| <i>kStatus_LPI2C_ArbitrationLost</i> | Arbitration lost error.                            |
| <i>kStatus_LPI2C_PinLowTimeout</i>   | SCL or SDA were held low longer than the timeout.  |

#### 16.4.5.22 status\_t LPI2C\_MasterReceive ( LPI2C\_Type \* *base*, void \* *rxBuff*, size\_t *rxSize* )

Parameters

|               |                                                    |
|---------------|----------------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.                 |
| <i>rxBuff</i> | The pointer to the data to be transferred.         |
| <i>rxSize</i> | The length in bytes of the data to be transferred. |

Return values

|                                      |                                                    |
|--------------------------------------|----------------------------------------------------|
| <i>kStatus_Success</i>               | Data was received successfully.                    |
| <i>kStatus_LPI2C_Busy</i>            | Another master is currently utilizing the bus.     |
| <i>kStatus_LPI2C_Nak</i>             | The slave device sent a NAK in response to a byte. |
| <i>kStatus_LPI2C_FifoError</i>       | FIFO under run or overrun.                         |
| <i>kStatus_LPI2C_ArbitrationLost</i> | Arbitration lost error.                            |
| <i>kStatus_LPI2C_PinLowTimeout</i>   | SCL or SDA were held low longer than the timeout.  |

#### 16.4.5.23 status\_t LPI2C\_MasterStop ( LPI2C\_Type \* *base* )

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Return values

|                                      |                                                                                  |
|--------------------------------------|----------------------------------------------------------------------------------|
| <i>kStatus_Success</i>               | The STOP signal was successfully sent on the bus and the transaction terminated. |
| <i>kStatus_LPI2C_Busy</i>            | Another master is currently utilizing the bus.                                   |
| <i>kStatus_LPI2C_Nak</i>             | The slave device sent a NAK in response to a byte.                               |
| <i>kStatus_LPI2C_FifoError</i>       | FIFO under run or overrun.                                                       |
| <i>kStatus_LPI2C_ArbitrationLost</i> | Arbitration lost error.                                                          |
| <i>kStatus_LPI2C_PinLowTimeout</i>   | SCL or SDA were held low longer than the timeout.                                |

#### 16.4.5.24 `status_t LPI2C_MasterTransferBlocking ( LPI2C_Type * base, Ipi2c_master_transfer_t * transfer )`

Note

The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

|                 |                                    |
|-----------------|------------------------------------|
| <i>base</i>     | The LPI2C peripheral base address. |
| <i>transfer</i> | Pointer to the transfer structure. |

Return values

|                                      |                                                    |
|--------------------------------------|----------------------------------------------------|
| <i>kStatus_Success</i>               | Data was received successfully.                    |
| <i>kStatus_LPI2C_Busy</i>            | Another master is currently utilizing the bus.     |
| <i>kStatus_LPI2C_Nak</i>             | The slave device sent a NAK in response to a byte. |
| <i>kStatus_LPI2C_FifoError</i>       | FIFO under run or overrun.                         |
| <i>kStatus_LPI2C_ArbitrationLost</i> | Arbitration lost error.                            |
| <i>kStatus_LPI2C_PinLowTimeout</i>   | SCL or SDA were held low longer than the timeout.  |

**16.4.5.25 void LPI2C\_MasterTransferCreateHandle ( *LPI2C\_Type \* base*,  
*lpi2c\_master\_handle\_t \* handle*, *lpi2c\_master\_transfer\_callback\_t callback*,  
*void \* userData* )**

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the [LPI2C\\_MasterTransferAbort\(\)](#) API shall be called.

Note

The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

|     |                 |                                                              |
|-----|-----------------|--------------------------------------------------------------|
|     | <i>base</i>     | The LPI2C peripheral base address.                           |
| out | <i>handle</i>   | Pointer to the LPI2C master driver handle.                   |
|     | <i>callback</i> | User provided pointer to the asynchronous callback function. |
|     | <i>userData</i> | User provided pointer to the application callback data.      |

**16.4.5.26 status\_t LPI2C\_MasterTransferNonBlocking ( *LPI2C\_Type \* base*,  
*lpi2c\_master\_handle\_t \* handle*, *lpi2c\_master\_transfer\_t \* transfer* )**

Parameters

|                 |                                            |
|-----------------|--------------------------------------------|
| <i>base</i>     | The LPI2C peripheral base address.         |
| <i>handle</i>   | Pointer to the LPI2C master driver handle. |
| <i>transfer</i> | The pointer to the transfer descriptor.    |

Return values

|                           |                                                                                                             |
|---------------------------|-------------------------------------------------------------------------------------------------------------|
| <i>kStatus_Success</i>    | The transaction was started successfully.                                                                   |
| <i>kStatus_LPI2C_Busy</i> | Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress. |

**16.4.5.27 status\_t LPI2C\_MasterTransferGetCount ( *LPI2C\_Type \* base*,  
*lpi2c\_master\_handle\_t \* handle*, *size\_t \* count* )**

Parameters

|     |               |                                                                     |
|-----|---------------|---------------------------------------------------------------------|
|     | <i>base</i>   | The LPI2C peripheral base address.                                  |
|     | <i>handle</i> | Pointer to the LPI2C master driver handle.                          |
| out | <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction. |

Return values

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

#### 16.4.5.28 void LPI2C\_MasterTransferAbort ( LPI2C\_Type \* *base*, Ipi2c\_master\_handle\_t \* *handle* )

Note

It is not safe to call this function from an IRQ handler that has a higher priority than the LPI2C peripheral's IRQ priority.

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.         |
| <i>handle</i> | Pointer to the LPI2C master driver handle. |

Return values

|                           |                                                                |
|---------------------------|----------------------------------------------------------------|
| <i>kStatus_Success</i>    | A transaction was successfully aborted.                        |
| <i>kStatus_LPI2C_Idle</i> | There is not a non-blocking transaction currently in progress. |

#### 16.4.5.29 void LPI2C\_MasterTransferHandleIRQ ( LPI2C\_Type \* *base*, void \* *Ipi2cMasterHandle* )

Note

This function does not need to be called unless you are reimplementing the nonblocking API's interrupt handler routines to add special functionality.

## Parameters

|                          |                                            |
|--------------------------|--------------------------------------------|
| <i>base</i>              | The LPI2C peripheral base address.         |
| <i>lpi2cMasterHandle</i> | Pointer to the LPI2C master driver handle. |

## 16.5 LPI2C Slave Driver

### 16.5.1 Overview

#### Data Structures

- struct `lpi2c_slave_config_t`  
*Structure with settings to initialize the LPI2C slave module. [More...](#)*
- struct `lpi2c_slave_transfer_t`  
*LPI2C slave transfer structure. [More...](#)*
- struct `lpi2c_slave_handle_t`  
*LPI2C slave handle structure. [More...](#)*

#### Typedefs

- typedef void(\* `lpi2c_slave_transfer_callback_t`)`(LPI2C_Type *base, lpi2c_slave_transfer_t *transfer, void *userData)`  
*Slave event callback function pointer type.*

#### Enumerations

- enum `_lpi2c_slave_flags` {
   
`kLPI2C_SlaveTxReadyFlag` = LPI2C\_SSR\_TDF\_MASK,  
`kLPI2C_SlaveRxReadyFlag` = LPI2C\_SSR\_RDF\_MASK,  
`kLPI2C_SlaveAddressValidFlag` = LPI2C\_SSR\_AVF\_MASK,  
`kLPI2C_SlaveTransmitAckFlag` = LPI2C\_SSR\_TAF\_MASK,  
`kLPI2C_SlaveRepeatedStartDetectFlag` = LPI2C\_SSR\_RSF\_MASK,  
`kLPI2C_SlaveStopDetectFlag` = LPI2C\_SSR\_SDF\_MASK,  
`kLPI2C_SlaveBitErrFlag` = LPI2C\_SSR\_BEF\_MASK,  
`kLPI2C_SlaveFifoErrFlag` = LPI2C\_SSR\_FEF\_MASK,  
`kLPI2C_SlaveAddressMatch0Flag` = LPI2C\_SSR\_AM0F\_MASK,  
`kLPI2C_SlaveAddressMatch1Flag` = LPI2C\_SSR\_AM1F\_MASK,  
`kLPI2C_SlaveGeneralCallFlag` = LPI2C\_SSR\_GCF\_MASK,  
`kLPI2C_SlaveBusyFlag` = LPI2C\_SSR\_SBF\_MASK,  
`kLPI2C_SlaveBusBusyFlag` = LPI2C\_SSR\_BBF\_MASK,  
`kLPI2C_SlaveClearFlags`,  
`kLPI2C_SlaveIrqFlags`,  
`kLPI2C_SlaveErrorFlags` = `kLPI2C_SlaveFifoErrFlag | kLPI2C_SlaveBitErrFlag` }
   
*LPI2C slave peripheral flags.*
- enum `lpi2c_slave_address_match_t` {
   
`kLPI2C_MatchAddress0` = 0U,  
`kLPI2C_MatchAddress0OrAddress1` = 2U,  
`kLPI2C_MatchAddress0ThroughAddress1` = 6U }
   
*LPI2C slave address match options.*

- enum `lpi2c_slave_transfer_event_t` {
 `kLPI2C_SlaveAddressMatchEvent` = 0x01U,
 `kLPI2C_SlaveTransmitEvent` = 0x02U,
 `kLPI2C_SlaveReceiveEvent` = 0x04U,
 `kLPI2C_SlaveTransmitAckEvent` = 0x08U,
 `kLPI2C_SlaveRepeatedStartEvent` = 0x10U,
 `kLPI2C_SlaveCompletionEvent` = 0x20U,
 `kLPI2C_SlaveAllEvents` }

*Set of events sent to the callback for non blocking slave transfers.*

## Slave initialization and deinitialization

- void `LPI2C_SlaveGetDefaultConfig` (`lpi2c_slave_config_t` \*`slaveConfig`)  
*Provides a default configuration for the LPI2C slave peripheral.*
- void `LPI2C_SlaveInit` (`LPI2C_Type` \*`base`, const `lpi2c_slave_config_t` \*`slaveConfig`, `uint32_t` `sourceClock_Hz`)  
*Initializes the LPI2C slave peripheral.*
- void `LPI2C_SlaveDeinit` (`LPI2C_Type` \*`base`)  
*Deinitializes the LPI2C slave peripheral.*
- static void `LPI2C_SlaveReset` (`LPI2C_Type` \*`base`)  
*Performs a software reset of the LPI2C slave peripheral.*
- static void `LPI2C_SlaveEnable` (`LPI2C_Type` \*`base`, `bool` `enable`)  
*Enables or disables the LPI2C module as slave.*

## Slave status

- static `uint32_t` `LPI2C_SlaveGetStatusFlags` (`LPI2C_Type` \*`base`)  
*Gets the LPI2C slave status flags.*
- static void `LPI2C_SlaveClearStatusFlags` (`LPI2C_Type` \*`base`, `uint32_t` `statusMask`)  
*Clears the LPI2C status flag state.*

## Slave interrupts

- static void `LPI2C_SlaveEnableInterrupts` (`LPI2C_Type` \*`base`, `uint32_t` `interruptMask`)  
*Enables the LPI2C slave interrupt requests.*
- static void `LPI2C_SlaveDisableInterrupts` (`LPI2C_Type` \*`base`, `uint32_t` `interruptMask`)  
*Disables the LPI2C slave interrupt requests.*
- static `uint32_t` `LPI2C_SlaveGetEnabledInterrupts` (`LPI2C_Type` \*`base`)  
*Returns the set of currently enabled LPI2C slave interrupt requests.*

## Slave DMA control

- static void `LPI2C_SlaveEnableDMA` (`LPI2C_Type` \*`base`, `bool` `enableAddressValid`, `bool` `enableRx`, `bool` `enableTx`)

*Enables or disables the LPI2C slave peripheral DMA requests.*

## Slave bus operations

- static bool [LPI2C\\_SlaveGetBusIdleState](#) (LPI2C\_Type \*base)  
*Returns whether the bus is idle.*
- static void [LPI2C\\_SlaveTransmitAck](#) (LPI2C\_Type \*base, bool ackOrNack)  
*Transmits either an ACK or NAK on the I2C bus in response to a byte from the master.*
- static uint32\_t [LPI2C\\_SlaveGetReceivedAddress](#) (LPI2C\_Type \*base)  
*Returns the slave address sent by the I2C master.*
- status\_t [LPI2C\\_SlaveSend](#) (LPI2C\_Type \*base, void \*txBuff, size\_t txSize, size\_t \*actualTxSize)  
*Performs a polling send transfer on the I2C bus.*
- status\_t [LPI2C\\_SlaveReceive](#) (LPI2C\_Type \*base, void \*rxBuff, size\_t rxSize, size\_t \*actualRxSize)  
*Performs a polling receive transfer on the I2C bus.*

## Slave non-blocking

- void [LPI2C\\_SlaveTransferCreateHandle](#) (LPI2C\_Type \*base, lpi2c\_slave\_handle\_t \*handle, [lpi2c\\_slave\\_transfer\\_callback\\_t](#) callback, void \*userData)  
*Creates a new handle for the LPI2C slave non-blocking APIs.*
- status\_t [LPI2C\\_SlaveTransferNonBlocking](#) (LPI2C\_Type \*base, lpi2c\_slave\_handle\_t \*handle, uint32\_t eventMask)  
*Starts accepting slave transfers.*
- status\_t [LPI2C\\_SlaveTransferGetCount](#) (LPI2C\_Type \*base, lpi2c\_slave\_handle\_t \*handle, size\_t \*count)  
*Gets the slave transfer status during a non-blocking transfer.*
- void [LPI2C\\_SlaveTransferAbort](#) (LPI2C\_Type \*base, lpi2c\_slave\_handle\_t \*handle)  
*Aborts the slave non-blocking transfers.*

## Slave IRQ handler

- void [LPI2C\\_SlaveTransferHandleIRQ](#) (LPI2C\_Type \*base, lpi2c\_slave\_handle\_t \*handle)  
*Reusable routine to handle slave interrupts.*

### 16.5.2 Data Structure Documentation

#### 16.5.2.1 struct lpi2c\_slave\_config\_t

This structure holds configuration settings for the LPI2C slave peripheral. To initialize this structure to reasonable defaults, call the [LPI2C\\_SlaveGetDefaultConfig\(\)](#) function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

## Data Fields

- bool `enableSlave`  
*Enable slave mode.*
- uint8\_t `address0`  
*Slave's 7-bit address.*
- uint8\_t `address1`  
*Alternate slave 7-bit address.*
- `lpi2c_slave_address_match_t addressMatchMode`  
*Address matching options.*
- bool `filterDozeEnable`  
*Enable digital glitch filter in doze mode.*
- bool `filterEnable`  
*Enable digital glitch filter.*
- bool `enableGeneralCall`  
*Enable general call address matching.*
- bool `ignoreAck`  
*Continue transfers after a NACK is detected.*
- bool `enableReceivedAddressRead`  
*Enable reading the address received address as the first byte of data.*
- uint32\_t `sdaGlitchFilterWidth_ns`  
*Width in nanoseconds of the digital filter on the SDA signal.*
- uint32\_t `sclGlitchFilterWidth_ns`  
*Width in nanoseconds of the digital filter on the SCL signal.*
- uint32\_t `dataValidDelay_ns`  
*Width in nanoseconds of the data valid delay.*
- uint32\_t `clockHoldTime_ns`  
*Width in nanoseconds of the clock hold time.*
- bool `enableAck`  
*Enables SCL clock stretching during slave-transmit address byte(s) and slave-receiver address and data byte(s) to allow software to write the Transmit ACK Register before the ACK or NACK is transmitted.*
- bool `enableTx`  
*Enables SCL clock stretching when the transmit data flag is set during a slave-transmit transfer.*
- bool `enableRx`  
*Enables SCL clock stretching when receive data flag is set during a slave-receive transfer.*
- bool `enableAddress`  
*Enables SCL clock stretching when the address valid flag is asserted.*

## Field Documentation

- (1) `bool lpi2c_slave_config_t::enableSlave`
- (2) `uint8_t lpi2c_slave_config_t::address0`
- (3) `uint8_t lpi2c_slave_config_t::address1`
- (4) `lpi2c_slave_address_match_t lpi2c_slave_config_t::addressMatchMode`
- (5) `bool lpi2c_slave_config_t::filterDozeEnable`
- (6) `bool lpi2c_slave_config_t::filterEnable`

(7) **bool lpi2c\_slave\_config\_t::enableGeneralCall**

(8) **bool lpi2c\_slave\_config\_t::enableAck**

Clock stretching occurs when transmitting the 9th bit. When enableAckSCLStall is enabled, there is no need to set either enableRxDataSCLStall or enableAddressSCLStall.

(9) **bool lpi2c\_slave\_config\_t::enableTx**

(10) **bool lpi2c\_slave\_config\_t::enableRx**

(11) **bool lpi2c\_slave\_config\_t::enableAddress**

(12) **bool lpi2c\_slave\_config\_t::ignoreAck**

(13) **bool lpi2c\_slave\_config\_t::enableReceivedAddressRead**

(14) **uint32\_t lpi2c\_slave\_config\_t::sdaGlitchFilterWidth\_ns**

Set to 0 to disable.

(15) **uint32\_t lpi2c\_slave\_config\_t::sclGlitchFilterWidth\_ns**

Set to 0 to disable.

(16) **uint32\_t lpi2c\_slave\_config\_t::dataValidDelay\_ns**

(17) **uint32\_t lpi2c\_slave\_config\_t::clockHoldTime\_ns**

### 16.5.2.2 struct lpi2c\_slave\_transfer\_t

#### Data Fields

- [lpi2c\\_slave\\_transfer\\_event\\_t event](#)  
*Reason the callback is being invoked.*
- [uint8\\_t receivedAddress](#)  
*Matching address send by master.*
- [uint8\\_t \\* data](#)  
*Transfer buffer.*
- [size\\_t dataSize](#)  
*Transfer size.*
- [status\\_t completionStatus](#)  
*Success or error code describing how the transfer completed.*
- [size\\_t transferredCount](#)  
*Number of bytes actually transferred since start or last repeated start.*

#### Field Documentation

(1) **lpi2c\_slave\_transfer\_event\_t lpi2c\_slave\_transfer\_t::event**

- (2) `uint8_t lpi2c_slave_transfer_t::receivedAddress`
- (3) `status_t lpi2c_slave_transfer_t::completionStatus`

Only applies for [kLPI2C\\_SlaveCompletionEvent](#).

- (4) `size_t lpi2c_slave_transfer_t::transferredCount`

### 16.5.2.3 struct \_lpi2c\_slave\_handle

Note

The contents of this structure are private and subject to change.

#### Data Fields

- `lpi2c_slave_transfer_t transfer`  
*LPI2C slave transfer copy.*
- `bool isBusy`  
*Whether transfer is busy.*
- `bool wasTransmit`  
*Whether the last transfer was a transmit.*
- `uint32_t eventMask`  
*Mask of enabled events.*
- `uint32_t transferredCount`  
*Count of bytes transferred.*
- `lpi2c_slave_transfer_callback_t callback`  
*Callback function called at transfer event.*
- `void *userData`  
*Callback parameter passed to callback.*

#### Field Documentation

- (1) `lpi2c_slave_transfer_t lpi2c_slave_handle_t::transfer`
- (2) `bool lpi2c_slave_handle_t::isBusy`
- (3) `bool lpi2c_slave_handle_t::wasTransmit`
- (4) `uint32_t lpi2c_slave_handle_t::eventMask`
- (5) `uint32_t lpi2c_slave_handle_t::transferredCount`
- (6) `lpi2c_slave_transfer_callback_t lpi2c_slave_handle_t::callback`
- (7) `void* lpi2c_slave_handle_t::userData`

### 16.5.3 Typedef Documentation

### 16.5.3.1 **typedef void(\* lpi2c\_slave\_transfer\_callback\_t)(LPI2C\_Type \*base, lpi2c\_slave\_transfer\_t \*transfer, void \*userData)**

This callback is used only for the slave non-blocking transfer API. To install a callback, use the LPI2C\_SlaveSetCallback() function after you have created a handle.

## Parameters

|                 |                                                                                      |
|-----------------|--------------------------------------------------------------------------------------|
| <i>base</i>     | Base address for the LPI2C instance on which the event occurred.                     |
| <i>transfer</i> | Pointer to transfer descriptor containing values passed to and/or from the callback. |
| <i>userData</i> | Arbitrary pointer-sized value passed from the application.                           |

**16.5.4 Enumeration Type Documentation****16.5.4.1 enum \_lpi2c\_slave\_flags**

The following status register flags can be cleared:

- [kLPI2C\\_SlaveRepeatedStartDetectFlag](#)
- [kLPI2C\\_SlaveStopDetectFlag](#)
- [kLPI2C\\_SlaveBitErrFlag](#)
- [kLPI2C\\_SlaveFifoErrFlag](#)

All flags except [kLPI2C\\_SlaveBusyFlag](#) and [kLPI2C\\_SlaveBusBusyFlag](#) can be enabled as interrupts.

## Note

These enumerations are meant to be OR'd together to form a bit mask.

## Enumerator

- kLPI2C\_SlaveTxReadyFlag* Transmit data flag.  
*kLPI2C\_SlaveRxReadyFlag* Receive data flag.  
*kLPI2C\_SlaveAddressValidFlag* Address valid flag.  
*kLPI2C\_SlaveTransmitAckFlag* Transmit ACK flag.  
*kLPI2C\_SlaveRepeatedStartDetectFlag* Repeated start detect flag.  
*kLPI2C\_SlaveStopDetectFlag* Stop detect flag.  
*kLPI2C\_SlaveBitErrFlag* Bit error flag.  
*kLPI2C\_SlaveFifoErrFlag* FIFO error flag.  
*kLPI2C\_SlaveAddressMatch0Flag* Address match 0 flag.  
*kLPI2C\_SlaveAddressMatch1Flag* Address match 1 flag.  
*kLPI2C\_SlaveGeneralCallFlag* General call flag.  
*kLPI2C\_SlaveBusyFlag* Master busy flag.  
*kLPI2C\_SlaveBusBusyFlag* Bus busy flag.  
*kLPI2C\_SlaveClearFlags* All flags which are cleared by the driver upon starting a transfer.  
*kLPI2C\_SlaveIrqFlags* IRQ sources enabled by the non-blocking transactional API.  
*kLPI2C\_SlaveErrorFlags* Errors to check for.

#### 16.5.4.2 enum lpi2c\_slave\_address\_match\_t

Enumerator

*kLPI2C\_MatchAddress0* Match only address 0.

*kLPI2C\_MatchAddress0OrAddress1* Match either address 0 or address 1.

*kLPI2C\_MatchAddress0ThroughAddress1* Match a range of slave addresses from address 0 through address 1.

#### 16.5.4.3 enum lpi2c\_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 [LPI2C\\_SlaveTransferNonBlocking\(\)](#) in order 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

*kLPI2C\_SlaveAddressMatchEvent* Received the slave address after a start or repeated start.

*kLPI2C\_SlaveTransmitEvent* Callback is requested to provide data to transmit (slave-transmitter role).

*kLPI2C\_SlaveReceiveEvent* Callback is requested to provide a buffer in which to place received data (slave-receiver role).

*kLPI2C\_SlaveTransmitAckEvent* Callback needs to either transmit an ACK or NACK.

*kLPI2C\_SlaveRepeatedStartEvent* A repeated start was detected.

*kLPI2C\_SlaveCompletionEvent* A stop was detected, completing the transfer.

*kLPI2C\_SlaveAllEvents* Bit mask of all available events.

#### 16.5.5 Function Documentation

##### 16.5.5.1 void LPI2C\_SlaveGetDefaultConfig ( lpi2c\_slave\_config\_t \* *slaveConfig* )

This function provides the following default configuration for the LPI2C slave peripheral:

```
* slaveConfig->enableSlave = true;
* slaveConfig->address0 = 0U;
* slaveConfig->address1 = 0U;
* slaveConfig->addressMatchMode = kLPI2C_MatchAddress0;
* slaveConfig->filterDozeEnable = true;
* slaveConfig->filterEnable = true;
* slaveConfig->enableGeneralCall = false;
* slaveConfig->sclStall.enableAck = false;
* slaveConfig->sclStall.enableTx = true;
* slaveConfig->sclStall.enableRx = true;
* slaveConfig->sclStall.enableAddress = true;
```

```

* slaveConfig->ignoreAck = false;
* slaveConfig->enableReceivedAddressRead = false;
* slaveConfig->sdaGlitchFilterWidth_ns = 0;
* slaveConfig->sclGlitchFilterWidth_ns = 0;
* slaveConfig->dataValidDelay_ns = 0;
* slaveConfig->clockHoldTime_ns = 0;
*

```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with [LPI2C\\_SlaveInit\(\)](#). Be sure to override at least the *address0* member of the configuration structure with the desired slave address.

Parameters

|            |                    |                                                                                                                      |
|------------|--------------------|----------------------------------------------------------------------------------------------------------------------|
| <i>out</i> | <i>slaveConfig</i> | User provided configuration structure that is set to default values. Refer to <a href="#">lpi2c_slave_config_t</a> . |
|------------|--------------------|----------------------------------------------------------------------------------------------------------------------|

#### 16.5.5.2 void LPI2C\_SlaveInit ( LPI2C\_Type \* *base*, const lpi2c\_slave\_config\_t \* *slaveConfig*, uint32\_t *sourceClock\_Hz* )

This function enables the peripheral clock and initializes the LPI2C slave peripheral as described by the user provided configuration.

Parameters

|                       |                                                                                                                                           |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>           | The LPI2C peripheral base address.                                                                                                        |
| <i>slaveConfig</i>    | User provided peripheral configuration. Use <a href="#">LPI2C_SlaveGetDefaultConfig()</a> to get a set of defaults that you can override. |
| <i>sourceClock_Hz</i> | Frequency in Hertz of the LPI2C functional clock. Used to calculate the filter widths, data valid delay, and clock hold time.             |

#### 16.5.5.3 void LPI2C\_SlaveDeinit ( LPI2C\_Type \* *base* )

This function disables the LPI2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

#### 16.5.5.4 static void LPI2C\_SlaveReset ( LPI2C\_Type \* *base* ) [inline], [static]

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

#### 16.5.5.5 static void LPI2C\_SlaveEnable ( LPI2C\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

|               |                                                                       |
|---------------|-----------------------------------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.                                    |
| <i>enable</i> | Pass true to enable or false to disable the specified LPI2C as slave. |

#### 16.5.5.6 static uint32\_t LPI2C\_SlaveGetStatusFlags ( LPI2C\_Type \* *base* ) [inline], [static]

A bit mask with the state of all LPI2C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

See Also

[\\_lpi2c\\_slave\\_flags](#)

#### 16.5.5.7 static void LPI2C\_SlaveClearStatusFlags ( LPI2C\_Type \* *base*, uint32\_t *statusMask* ) [inline], [static]

The following status register flags can be cleared:

- [kLPI2C\\_SlaveRepeatedStartDetectFlag](#)
- [kLPI2C\\_SlaveStopDetectFlag](#)
- [kLPI2C\\_SlaveBitErrFlag](#)
- [kLPI2C\\_SlaveFifoErrFlag](#)

Attempts to clear other flags has no effect.

Parameters

|                   |                                                                                                                                                                                                                                     |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>       | The LPI2C peripheral base address.                                                                                                                                                                                                  |
| <i>statusMask</i> | A bitmask of status flags that are to be cleared. The mask is composed of <a href="#">_lpi2c_slave_flags</a> enumerators OR'd together. You may pass the result of a previous call to <a href="#">LPI2C_SlaveGetStatusFlags()</a> . |

See Also

[\\_lpi2c\\_slave\\_flags](#).

#### 16.5.5.8 static void LPI2C\_SlaveEnableInterrupts ( LPI2C\_Type \* *base*, uint32\_t *interruptMask* ) [inline], [static]

All flags except [kLPI2C\\_SlaveBusyFlag](#) and [kLPI2C\\_SlaveBusBusyFlag](#) can be enabled as interrupts.

Parameters

|                      |                                                                                                                                                      |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>          | The LPI2C peripheral base address.                                                                                                                   |
| <i>interruptMask</i> | Bit mask of interrupts to enable. See <a href="#">_lpi2c_slave_flags</a> for the set of constants that should be OR'd together to form the bit mask. |

#### 16.5.5.9 static void LPI2C\_SlaveDisableInterrupts ( LPI2C\_Type \* *base*, uint32\_t *interruptMask* ) [inline], [static]

All flags except [kLPI2C\\_SlaveBusyFlag](#) and [kLPI2C\\_SlaveBusBusyFlag](#) can be disabled as interrupts.

Parameters

|                      |                                                                                                                                                       |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>          | The LPI2C peripheral base address.                                                                                                                    |
| <i>interruptMask</i> | Bit mask of interrupts to disable. See <a href="#">_lpi2c_slave_flags</a> for the set of constants that should be OR'd together to form the bit mask. |

#### 16.5.5.10 static uint32\_t LPI2C\_SlaveGetEnabledInterrupts ( LPI2C\_Type \* *base* ) [inline], [static]

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Returns

A bitmask composed of [\\_lpi2c\\_slave\\_flags](#) enumerators OR'd together to indicate the set of enabled interrupts.

#### 16.5.5.11 static void LPI2C\_SlaveEnableDMA ( LPI2C\_Type \* *base*, bool *enableAddressValid*, bool *enableRx*, bool *enableTx* ) [inline], [static]

Parameters

|                           |                                                                                                                                                                    |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>               | The LPI2C peripheral base address.                                                                                                                                 |
| <i>enableAddressValid</i> | Enable flag for the address valid DMA request. Pass true for enable, false for disable. The address valid DMA request is shared with the receive data DMA request. |
| <i>enableRx</i>           | Enable flag for the receive data DMA request. Pass true for enable, false for disable.                                                                             |
| <i>enableTx</i>           | Enable flag for the transmit data DMA request. Pass true for enable, false for disable.                                                                            |

#### 16.5.5.12 static bool LPI2C\_SlaveGetBusIdleState ( LPI2C\_Type \* *base* ) [inline], [static]

Requires the slave mode to be enabled.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Return values

|              |              |
|--------------|--------------|
| <i>true</i>  | Bus is busy. |
| <i>false</i> | Bus is idle. |

#### 16.5.5.13 static void LPI2C\_SlaveTransmitAck ( LPI2C\_Type \* *base*, bool *ackOrNack* ) [inline], [static]

Use this function to send an ACK or NAK when the [KLPI2C\\_SlaveTransmitAckFlag](#) is asserted. This only happens if you enable the sclStall.enableAck field of the [lpi2c\\_slave\\_config\\_t](#) configuration structure used to initialize the slave peripheral.

Parameters

|                  |                                          |
|------------------|------------------------------------------|
| <i>base</i>      | The LPI2C peripheral base address.       |
| <i>ackOrNack</i> | Pass true for an ACK or false for a NAK. |

#### 16.5.5.14 static uint32\_t LPI2C\_SlaveGetReceivedAddress ( LPI2C\_Type \* *base* ) [inline], [static]

This function should only be called if the [kLPI2C\\_SlaveAddressValidFlag](#) is asserted.

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

Returns

The 8-bit address matched by the LPI2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

#### 16.5.5.15 status\_t LPI2C\_SlaveSend ( LPI2C\_Type \* *base*, void \* *txBuff*, size\_t *txSize*, size\_t \* *actualTxSize* )

Parameters

|     |                     |                                                    |
|-----|---------------------|----------------------------------------------------|
|     | <i>base</i>         | The LPI2C peripheral base address.                 |
|     | <i>txBuff</i>       | The pointer to the data to be transferred.         |
|     | <i>txSize</i>       | The length in bytes of the data to be transferred. |
| out | <i>actualTxSize</i> |                                                    |

Returns

Error or success status returned by API.

#### 16.5.5.16 status\_t LPI2C\_SlaveReceive ( LPI2C\_Type \* *base*, void \* *rxBuff*, size\_t *rxSize*, size\_t \* *actualRxSize* )

## Parameters

|     |                     |                                                    |
|-----|---------------------|----------------------------------------------------|
|     | <i>base</i>         | The LPI2C peripheral base address.                 |
|     | <i>rxBuff</i>       | The pointer to the data to be transferred.         |
|     | <i>rxSize</i>       | The length in bytes of the data to be transferred. |
| out | <i>actualRxSize</i> |                                                    |

## Returns

Error or success status returned by API.

**16.5.5.17 void LPI2C\_SlaveTransferCreateHandle ( LPI2C\_Type \* *base*, Ipi2c\_slave\_handle\_t \* *handle*, Ipi2c\_slave\_transfer\_callback\_t *callback*, void \* *userData* )**

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the [LPI2C\\_SlaveTransferAbort\(\)](#) API shall be called.

## Note

The function also enables the NVIC IRQ for the input LPI2C. Need to notice that on some SoCs the LPI2C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

## Parameters

|     |                 |                                                              |
|-----|-----------------|--------------------------------------------------------------|
|     | <i>base</i>     | The LPI2C peripheral base address.                           |
| out | <i>handle</i>   | Pointer to the LPI2C slave driver handle.                    |
|     | <i>callback</i> | User provided pointer to the asynchronous callback function. |
|     | <i>userData</i> | User provided pointer to the application callback data.      |

**16.5.5.18 status\_t LPI2C\_SlaveTransferNonBlocking ( LPI2C\_Type \* *base*, Ipi2c\_slave\_handle\_t \* *handle*, uint32\_t *eventMask* )**

Call this API after calling [I2C\\_SlaveInit\(\)](#) and [LPI2C\\_SlaveTransferCreateHandle\(\)](#) to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to [LPI2C\\_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 [ipi2c\\_slave\\_transfer\\_event\\_t](#) enumerators for the events you wish to receive. The

`kLPI2C_SlaveTransmitEvent` and `kLPI2C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `kLPI2C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

|                  |                                                                                                                                                                                                                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>      | The LPI2C peripheral base address.                                                                                                                                                                                                                                                               |
| <i>handle</i>    | Pointer to <code>lpi2c_slave_handle_t</code> structure which stores the transfer state.                                                                                                                                                                                                          |
| <i>eventMask</i> | Bit mask formed by OR'ing together <code>lpi2c_slave_transfer_event_t</code> 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 <code>kLPI2C_SlaveAllEvents</code> to enable all events. |

Return values

|                                 |                                                           |
|---------------------------------|-----------------------------------------------------------|
| <code>kStatus_Success</code>    | Slave transfers were successfully started.                |
| <code>kStatus_LPI2C_Busy</code> | Slave transfers have already been started on this handle. |

#### 16.5.5.19 `status_t LPI2C_SlaveTransferGetCount ( LPI2C_Type * base, lpi2c_slave_handle_t * handle, size_t * count )`

Parameters

|            |               |                                                                                                       |
|------------|---------------|-------------------------------------------------------------------------------------------------------|
|            | <i>base</i>   | The LPI2C peripheral base address.                                                                    |
|            | <i>handle</i> | Pointer to <code>i2c_slave_handle_t</code> structure.                                                 |
| <i>out</i> | <i>count</i>  | Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required. |

Return values

|                                            |  |
|--------------------------------------------|--|
| <code>kStatus_Success</code>               |  |
| <code>kStatus_NoTransferIn-Progress</code> |  |

#### 16.5.5.20 `void LPI2C_SlaveTransferAbort ( LPI2C_Type * base, lpi2c_slave_handle_t * handle )`

## Note

This API could be called at any time to stop slave for handling the bus events.

## Parameters

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.                                         |
| <i>handle</i> | Pointer to lpi2c_slave_handle_t structure which stores the transfer state. |

## Return values

|                           |  |
|---------------------------|--|
| <i>kStatus_Success</i>    |  |
| <i>kStatus_LPI2C_Idle</i> |  |

**16.5.5.21 void LPI2C\_SlaveTransferHandleIRQ ( LPI2C\_Type \* *base*, lpi2c\_slave\_handle\_t \* *handle* )**

## Note

This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

## Parameters

|               |                                                                            |
|---------------|----------------------------------------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.                                         |
| <i>handle</i> | Pointer to lpi2c_slave_handle_t structure which stores the transfer state. |

## 16.6 LPI2C Master DMA Driver

### 16.6.1 Overview

#### Data Structures

- struct `lpi2c_master_edma_handle_t`  
*Driver handle for master DMA APIs.* [More...](#)

#### Typedefs

- typedef void(\* `lpi2c_master_edma_transfer_callback_t`)(LPI2C\_Type \*base, lpi2c\_master\_edma\_handle\_t \*handle, `status_t` completionStatus, void \*userData)  
*Master DMA completion callback function pointer type.*

#### Master DMA

- void `LPI2C_MasterCreateEDMAHandle` (LPI2C\_Type \*base, lpi2c\_master\_edma\_handle\_t \*handle, `edma_handle_t` \*rxDmaHandle, `edma_handle_t` \*txDmaHandle, `lpi2c_master_edma_transfer_callback_t` callback, void \*userData)  
*Create a new handle for the LPI2C master DMA APIs.*
- `status_t LPI2C_MasterTransferEDMA` (LPI2C\_Type \*base, lpi2c\_master\_edma\_handle\_t \*handle, lpi2c\_master\_transfer\_t \*transfer)  
*Performs a non-blocking DMA-based transaction on the I2C bus.*
- `status_t LPI2C_MasterTransferGetCountEDMA` (LPI2C\_Type \*base, lpi2c\_master\_edma\_handle\_t \*handle, `size_t` \*count)  
*Returns number of bytes transferred so far.*
- `status_t LPI2C_MasterTransferAbortEDMA` (LPI2C\_Type \*base, lpi2c\_master\_edma\_handle\_t \*handle)  
*Terminates a non-blocking LPI2C master transmission early.*

### 16.6.2 Data Structure Documentation

#### 16.6.2.1 struct \_lpi2c\_master\_edma\_handle

Note

The contents of this structure are private and subject to change.

#### Data Fields

- `LPI2C_Type * base`  
*LPI2C base pointer.*
- `bool isBusy`

- *Transfer state machine current state.*
- `uint8_t nbytes`  
*eDMA minor byte transfer count initially configured.*
- `uint16_t commandBuffer[10]`  
*LPI2C command sequence.*
- `lpi2c_master_transfer_t transfer`  
*Copy of the current transfer info.*
- `lpi2c_master_edma_transfer_callback_t completionCallback`  
*Callback function pointer.*
- `void *userData`  
*Application data passed to callback.*
- `edma_handle_t *rx`  
*Handle for receive DMA channel.*
- `edma_handle_t *tx`  
*Handle for transmit DMA channel.*
- `edma_tcd_t tcds[3]`  
*Software TCD.*

### Field Documentation

- (1) `LPI2C_Type* lpi2c_master_edma_handle_t::base`
- (2) `bool lpi2c_master_edma_handle_t::isBusy`
- (3) `uint8_t lpi2c_master_edma_handle_t::nbytes`
- (4) `uint16_t lpi2c_master_edma_handle_t::commandBuffer[10]`

When all 10 command words are used: Start&addr&write[1 word] + subaddr[4 words] + restart&addr&read[1 word] + receive&Size[4 words]

- (5) `lpi2c_master_transfer_t lpi2c_master_edma_handle_t::transfer`
- (6) `lpi2c_master_edma_transfer_callback_t lpi2c_master_edma_handle_t::completionCallback`
- (7) `void* lpi2c_master_edma_handle_t::userData`
- (8) `edma_handle_t* lpi2c_master_edma_handle_t::rx`
- (9) `edma_handle_t* lpi2c_master_edma_handle_t::tx`
- (10) `edma_tcd_t lpi2c_master_edma_handle_t::tcds[3]`

Three are allocated to provide enough room to align to 32-bytes.

### 16.6.3 Typedef Documentation

**16.6.3.1 `typedef void(* Ipi2c_master_edma_transfer_callback_t)(LPI2C_Type *base,  
Ipi2c_master_edma_handle_t *handle, status_t completionStatus, void  
*userData)`**

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to [LPI2C\\_MasterCreateEDMAHandle\(\)](#).

Parameters

|                          |                                                                                |
|--------------------------|--------------------------------------------------------------------------------|
| <i>base</i>              | The LPI2C peripheral base address.                                             |
| <i>handle</i>            | Handle associated with the completed transfer.                                 |
| <i>completion-Status</i> | Either kStatus_Success or an error code describing how the transfer completed. |
| <i>userData</i>          | Arbitrary pointer-sized value passed from the application.                     |

## 16.6.4 Function Documentation

**16.6.4.1 void LPI2C\_MasterCreateEDMAHandle ( LPI2C\_Type \* *base*, Ipi2c\_master\_edma\_handle\_t \* *handle*, edma\_handle\_t \* *rxDmaHandle*, edma\_handle\_t \* *txDmaHandle*, Ipi2c\_master\_edma\_transfer\_callback\_t *callback*, void \* *userData* )**

The creation of a handle is for use with the DMA APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the [LPI2C\\_MasterTransferAbort-EDMA\(\)](#) API shall be called.

For devices where the LPI2C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

|     |                    |                                                                                           |
|-----|--------------------|-------------------------------------------------------------------------------------------|
|     | <i>base</i>        | The LPI2C peripheral base address.                                                        |
| out | <i>handle</i>      | Pointer to the LPI2C master driver handle.                                                |
|     | <i>rxDmaHandle</i> | Handle for the eDMA receive channel. Created by the user prior to calling this function.  |
|     | <i>txDmaHandle</i> | Handle for the eDMA transmit channel. Created by the user prior to calling this function. |
|     | <i>callback</i>    | User provided pointer to the asynchronous callback function.                              |
|     | <i>userData</i>    | User provided pointer to the application callback data.                                   |

**16.6.4.2 status\_t LPI2C\_MasterTransferEDMA ( LPI2C\_Type \* *base*, Ipi2c\_master\_edma\_handle\_t \* *handle*, Ipi2c\_master\_transfer\_t \* *transfer* )**

The callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

|                 |                                            |
|-----------------|--------------------------------------------|
| <i>base</i>     | The LPI2C peripheral base address.         |
| <i>handle</i>   | Pointer to the LPI2C master driver handle. |
| <i>transfer</i> | The pointer to the transfer descriptor.    |

Return values

|                           |                                                                                                          |
|---------------------------|----------------------------------------------------------------------------------------------------------|
| <i>kStatus_Success</i>    | The transaction was started successfully.                                                                |
| <i>kStatus_LPI2C_Busy</i> | Either another master is currently utilizing the bus, or another DMA transaction is already in progress. |

#### 16.6.4.3 status\_t LPI2C\_MasterTransferGetCountEDMA ( **LPI2C\_Type \* base,**                   *lpi2c\_master\_edma\_handle\_t \* handle, size\_t \* count* )

Parameters

|            |               |                                                                     |
|------------|---------------|---------------------------------------------------------------------|
|            | <i>base</i>   | The LPI2C peripheral base address.                                  |
|            | <i>handle</i> | Pointer to the LPI2C master driver handle.                          |
| <i>out</i> | <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction. |

Return values

|                                     |                                                       |
|-------------------------------------|-------------------------------------------------------|
| <i>kStatus_Success</i>              |                                                       |
| <i>kStatus_NoTransferInProgress</i> | There is not a DMA transaction currently in progress. |

#### 16.6.4.4 status\_t LPI2C\_MasterTransferAbortEDMA ( **LPI2C\_Type \* base,**                   *lpi2c\_master\_edma\_handle\_t \* handle* )

Note

It is not safe to call this function from an IRQ handler that has a higher priority than the eDMA peripheral's IRQ priority.

## Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.         |
| <i>handle</i> | Pointer to the LPI2C master driver handle. |

## Return values

|                           |                                                       |
|---------------------------|-------------------------------------------------------|
| <i>kStatus_Success</i>    | A transaction was successfully aborted.               |
| <i>kStatus_LPI2C_Idle</i> | There is not a DMA transaction currently in progress. |

## 16.7 LPI2C FreeRTOS Driver

### 16.7.1 Overview

#### Driver version

- `#define FSL_LPI2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 3, 0))`  
*LPI2C FreeRTOS driver version.*

#### LPI2C RTOS Operation

- `status_t LPI2C_RRTOS_Init (lpi2c_rtos_handle_t *handle, LPI2C_Type *base, const lpi2c_master_config_t *masterConfig, uint32_t srcClock_Hz)`  
*Initializes LPI2C.*
- `status_t LPI2C_RRTOS_Deinit (lpi2c_rtos_handle_t *handle)`  
*Deinitializes the LPI2C.*
- `status_t LPI2C_RRTOS_Transfer (lpi2c_rtos_handle_t *handle, lpi2c_master_transfer_t *transfer)`  
*Performs I2C transfer.*

### 16.7.2 Macro Definition Documentation

#### 16.7.2.1 `#define FSL_LPI2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 3, 0))`

### 16.7.3 Function Documentation

#### 16.7.3.1 `status_t LPI2C_RRTOS_Init ( lpi2c_rtos_handle_t * handle, LPI2C_Type * base, const lpi2c_master_config_t * masterConfig, uint32_t srcClock_Hz )`

This function initializes the LPI2C module and related RTOS context.

Parameters

|                           |                                                                            |
|---------------------------|----------------------------------------------------------------------------|
| <code>handle</code>       | The RTOS LPI2C handle, the pointer to an allocated space for RTOS context. |
| <code>base</code>         | The pointer base address of the LPI2C instance to initialize.              |
| <code>masterConfig</code> | Configuration structure to set-up LPI2C in master mode.                    |
| <code>srcClock_Hz</code>  | Frequency of input clock of the LPI2C module.                              |

Returns

status of the operation.

### 16.7.3.2 status\_t LPI2C\_RTOS\_Deinit ( *lpi2c\_rtos\_handle\_t \* handle* )

This function deinitializes the LPI2C module and related RTOS context.

Parameters

|               |                        |
|---------------|------------------------|
| <i>handle</i> | The RTOS LPI2C handle. |
|---------------|------------------------|

#### 16.7.3.3 status\_t LPI2C\_RTOS\_Transfer ( *lpi2c\_rtos\_handle\_t \* handle,* *lpi2c\_master\_transfer\_t \* transfer* )

This function performs an I2C transfer using LPI2C module according to data given in the transfer structure.

Parameters

|                 |                                               |
|-----------------|-----------------------------------------------|
| <i>handle</i>   | The RTOS LPI2C handle.                        |
| <i>transfer</i> | Structure specifying the transfer parameters. |

Returns

status of the operation.

## 16.8 LPI2C CMSIS Driver

This section describes the programming interface of the LPI2C Cortex Microcontroller Software Interface Standard (CMSIS) driver. And this driver defines generic peripheral driver interfaces for middleware making it reusable across a wide range of supported microcontroller devices. The API connects microcontroller peripherals with middleware that implements for example communication stacks, file systems, or graphic user interfaces. More information and usage method see <http://www.keil.com/pack/doc/cmsis/Driver/html/index.html>.

The LPI2C CMSIS driver includes transactional APIs.

Transactional APIs are transaction 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 accessing the hardware registers.

### 16.8.1 LPI2C CMSIS Driver

#### 16.8.1.1 Master Operation in interrupt transactional method

```
void I2C_MasterSignalEvent_t(uint32_t event)
{
 if (event == ARM_I2C_EVENT_TRANSFER_DONE)
 {
 g_MasterCompletionFlag = true;
 }
}
/*Init I2C0*/
Driver_I2C0.Initialize(I2C_MasterSignalEvent_t);

Driver_I2C0.PowerControl(ARM_POWER_FULL);

/*config transmit speed/
Driver_I2C0.Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_STANDARD);

/*start transmit*/
Driver_I2C0.MasterTransmit(I2C_MASTER_SLAVE_ADDR, g_master_buff, I2C_DATA_LENGTH, false);

/* Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

#### 16.8.1.2 Master Operation in DMA transactional method

```
void I2C_MasterSignalEvent_t(uint32_t event)
{
 /* Transfer done */
 if (event == ARM_I2C_EVENT_TRANSFER_DONE)
 {
 g_MasterCompletionFlag = true;
 }
}

/* DMAMux init and EDMA init. */
DMAMUX_Init(EXAMPLE_LPI2C_DMAMUX_BASEADDR);
```

```

edma_config_t edmaConfig;
EDMA_GetDefaultConfig(&edmaConfig);
EDMA_Init(EXAMPLE_LPI2C_DMA_BASEADDR, &edmaConfig);

/*Init I2C0*/
Driver_I2C0.Initialize(I2C_MasterSignalEvent_t);

Driver_I2C0.PowerControl(ARM_POWER_FULL);

/*config transmit speed*/
Driver_I2C0.Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_STANDARD);

/*start transfer*/
Driver_I2C0.MasterReceive(I2C_MASTER_SLAVE_ADDR, g_master_buff, I2C_DATA_LENGTH, false);

/* Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;

```

### 16.8.1.3 Slave Operation in interrupt transactional method

```

void I2C_SlaveSignalEvent_t(uint32_t event)
{
 /* Transfer done */
 if (event == ARM_I2C_EVENT_TRANSFER_DONE)
 {
 g_SlaveCompletionFlag = true;
 }
}

/*Init I2C1*/
Driver_I2C1.Initialize(I2C_SlaveSignalEvent_t);

Driver_I2C1.PowerControl(ARM_POWER_FULL);

/*config slave addr*/
Driver_I2C1.Control(ARM_I2C_OWN_ADDRESS, I2C_MASTER_SLAVE_ADDR);

/*start transfer*/
Driver_I2C1.SlaveReceive(g_slave_buff, I2C_DATA_LENGTH);

/* Wait for transfer completed. */
while (!g_SlaveCompletionFlag)
{
}
g_SlaveCompletionFlag = false;

```

# Chapter 17

## LPIT: Low-Power Interrupt Timer

### 17.1 Overview

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

### 17.2 Function groups

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

#### 17.2.1 Initialization and deinitialization

The function [LPIT\\_Init\(\)](#) initializes the LPIT with specified configurations. The function [LPIT\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the LPIT operation in doze mode and debug mode.

The function [LPIT\\_SetupChannel\(\)](#) configures the operation of each LPIT channel.

The function [LPIT\\_Deinit\(\)](#) disables the LPIT module and disables the module clock.

#### 17.2.2 Timer period Operations

The function [LPITR\\_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 [LPIT\\_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. User can call the utility macros provided in `fsl_common.h` to convert to microseconds or milliseconds

#### 17.2.3 Start and Stop timer operations

The function [LPIT\\_StartTimer\(\)](#) starts the timer counting. After calling this function, the timer loads the period value set earlier via the [LPIT\\_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 [LPIT\\_StopTimer\(\)](#) stops the timer counting.

## 17.2.4 Status

Provides functions to get and clear the LPIT status.

## 17.2.5 Interrupt

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

## 17.3 Typical use case

### 17.3.1 LPIT tick example

Updates the LPIT period and toggles an LED periodically. Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/lpit

## Data Structures

- struct [lpit\\_chnl\\_params\\_t](#)  
*Structure to configure the channel timer. [More...](#)*
- struct [lpit\\_config\\_t](#)  
*LPIT configuration structure. [More...](#)*

## Functions

- static void [LPIT\\_Reset](#) (LPIT\_Type \*base)  
*Performs a software reset on the LPIT module.*

## Driver version

- enum [lpit\\_chnl\\_t](#) {  
  kLPIT\_Chnl\_0 = 0U,  
  kLPIT\_Chnl\_1,  
  kLPIT\_Chnl\_2,  
  kLPIT\_Chnl\_3 }  
*List of LPIT channels.*
- enum [lpit\\_timer\\_modes\\_t](#) {  
  kLPIT\_PeriodicCounter = 0U,  
  kLPIT\_DualPeriodicCounter,  
  kLPIT\_TriggerAccumulator,  
  kLPIT\_InputCapture }  
*Mode options available for the LPIT timer.*
- enum [lpit\\_trigger\\_select\\_t](#) {

```
kLPIT_Trigger_TimerChn0 = 0U,
kLPIT_Trigger_TimerChn1,
kLPIT_Trigger_TimerChn2,
kLPIT_Trigger_TimerChn3,
kLPIT_Trigger_TimerChn4,
kLPIT_Trigger_TimerChn5,
kLPIT_Trigger_TimerChn6,
kLPIT_Trigger_TimerChn7,
kLPIT_Trigger_TimerChn8,
kLPIT_Trigger_TimerChn9,
kLPIT_Trigger_TimerChn10,
kLPIT_Trigger_TimerChn11,
kLPIT_Trigger_TimerChn12,
kLPIT_Trigger_TimerChn13,
kLPIT_Trigger_TimerChn14,
kLPIT_Trigger_TimerChn15 }
```

*Trigger options available.*

- enum `lpit_trigger_source_t` {
   
kLPIT\_TriggerSource\_External = 0U,
   
kLPIT\_TriggerSource\_Internal }

*Trigger source options available.*

- enum `lpit_interrupt_enable_t` {
   
kLPIT\_Channel0TimerInterruptEnable = (1U << 0),
   
kLPIT\_Channel1TimerInterruptEnable = (1U << 1),
   
kLPIT\_Channel2TimerInterruptEnable = (1U << 2),
   
kLPIT\_Channel3TimerInterruptEnable = (1U << 3) }

*List of LPIT interrupts.*

- enum `lpit_status_flags_t` {
   
kLPIT\_Channel0TimerFlag = (1U << 0),
   
kLPIT\_Channel1TimerFlag = (1U << 1),
   
kLPIT\_Channel2TimerFlag = (1U << 2),
   
kLPIT\_Channel3TimerFlag = (1U << 3) }

*List of LPIT status flags.*

- #define `FSL_LPIT_DRIVER_VERSION` (MAKE\_VERSION(2, 0, 2))
   
*Version 2.0.2.*

## Initialization and deinitialization

- void `LPIT_Init` (LPIT\_Type \*base, const `lpit_config_t` \*config)
   
*Ungates the LPIT clock and configures the peripheral for a basic operation.*
- void `LPIT_Deinit` (LPIT\_Type \*base)
   
*Disables the module and gates the LPIT clock.*
- void `LPIT_GetDefaultConfig` (`lpit_config_t` \*config)
   
*Fills in the LPIT configuration structure with default settings.*
- `status_t LPIT_SetupChannel` (LPIT\_Type \*base, `lpit_chnl_t` channel, const `lpit_chnl_params_t` \*chnlSetup)
   
*Sets up an LPIT channel based on the user's preference.*

## Interrupt Interface

- static void [LPIT\\_EnableInterrupts](#) (LPIT\_Type \*base, uint32\_t mask)  
*Enables the selected PIT interrupts.*
- static void [LPIT\\_DisableInterrupts](#) (LPIT\_Type \*base, uint32\_t mask)  
*Disables the selected PIT interrupts.*
- static uint32\_t [LPIT\\_GetEnabledInterrupts](#) (LPIT\_Type \*base)  
*Gets the enabled LPIT interrupts.*

## Status Interface

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

## Read and Write the timer period

- static void [LPIT\\_SetTimerPeriod](#) (LPIT\_Type \*base, lpit\_chnl\_t channel, uint32\_t ticks)  
*Sets the timer period in units of count.*
- static uint32\_t [LPIT\\_GetCurrentTimerCount](#) (LPIT\_Type \*base, lpit\_chnl\_t channel)  
*Reads the current timer counting value.*

## Timer Start and Stop

- static void [LPIT\\_StartTimer](#) (LPIT\_Type \*base, lpit\_chnl\_t channel)  
*Starts the timer counting.*
- static void [LPIT\\_StopTimer](#) (LPIT\_Type \*base, lpit\_chnl\_t channel)  
*Stops the timer counting.*

## 17.4 Data Structure Documentation

### 17.4.1 struct lpit\_chnl\_params\_t

#### Data Fields

- bool [chainChannel](#)  
*true: Timer chained to previous timer; false: Timer not chained*
- [lpit\\_timer\\_modes\\_t timerMode](#)  
*Timers mode of operation.*
- [lpit\\_trigger\\_select\\_t triggerSelect](#)  
*Trigger selection for the timer.*
- [lpit\\_trigger\\_source\\_t triggerSource](#)  
*Decides if we use external or internal trigger.*
- bool [enableReloadOnTrigger](#)  
*true: Timer reloads when a trigger is detected; false: No effect*
- bool [enableStopOnTimeout](#)  
*true: Timer will stop after timeout; false: does not stop after timeout*
- bool [enableStartOnTrigger](#)  
*true: Timer starts when a trigger is detected; false: decrement immediately*

**Field Documentation**

- (1) `lpit_timer_modes_t lpit_chnl_params_t::timerMode`
- (2) `lpit_trigger_source_t lpit_chnl_params_t::triggerSource`

**17.4.2 struct lpit\_config\_t**

This structure holds the configuration settings for the LPIT peripheral. To initialize this structure to reasonable defaults, call the `LPIT_GetDefaultConfig()` function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

**Data Fields**

- bool `enableRunInDebug`  
*true: Timers run in debug mode; false: Timers stop in debug mode*
- bool `enableRunInDoze`  
*true: Timers run in doze mode; false: Timers stop in doze mode*

**17.5 Enumeration Type Documentation****17.5.1 enum lpit\_chnl\_t**

Note

Actual number of available channels is SoC-dependent

Enumerator

- `kLPIT_Chnl_0` LPIT channel number 0.
- `kLPIT_Chnl_1` LPIT channel number 1.
- `kLPIT_Chnl_2` LPIT channel number 2.
- `kLPIT_Chnl_3` LPIT channel number 3.

**17.5.2 enum lpit\_timer\_modes\_t**

Enumerator

- `kLPIT_PeriodicCounter` Use the all 32-bits, counter loads and decrements to zero.
- `kLPIT_DualPeriodicCounter` Counter loads, lower 16-bits decrement to zero, then upper 16-bits decrement.
- `kLPIT_TriggerAccumulator` Counter loads on first trigger and decrements on each trigger.
- `kLPIT_InputCapture` Counter loads with 0xFFFFFFFF, decrements to zero. It stores the inverse of the current value when a input trigger is detected

### 17.5.3 enum lpit\_trigger\_select\_t

This is used for both internal and external trigger sources. The actual trigger options available is SoC-specific, user should refer to the reference manual.

Enumerator

|                                 |                                             |
|---------------------------------|---------------------------------------------|
| <i>kLPIT_Trigger_TimerChn0</i>  | Channel 0 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn1</i>  | Channel 1 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn2</i>  | Channel 2 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn3</i>  | Channel 3 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn4</i>  | Channel 4 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn5</i>  | Channel 5 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn6</i>  | Channel 6 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn7</i>  | Channel 7 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn8</i>  | Channel 8 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn9</i>  | Channel 9 is selected as a trigger source.  |
| <i>kLPIT_Trigger_TimerChn10</i> | Channel 10 is selected as a trigger source. |
| <i>kLPIT_Trigger_TimerChn11</i> | Channel 11 is selected as a trigger source. |
| <i>kLPIT_Trigger_TimerChn12</i> | Channel 12 is selected as a trigger source. |
| <i>kLPIT_Trigger_TimerChn13</i> | Channel 13 is selected as a trigger source. |
| <i>kLPIT_Trigger_TimerChn14</i> | Channel 14 is selected as a trigger source. |
| <i>kLPIT_Trigger_TimerChn15</i> | Channel 15 is selected as a trigger source. |

### 17.5.4 enum lpit\_trigger\_source\_t

Enumerator

|                                     |                             |
|-------------------------------------|-----------------------------|
| <i>kLPIT_TriggerSource_External</i> | Use external trigger input. |
| <i>kLPIT_TriggerSource_Internal</i> | Use internal trigger.       |

### 17.5.5 enum lpit\_interrupt\_enable\_t

Note

Number of timer channels are SoC-specific. See the SoC Reference Manual.

Enumerator

|                                           |                            |
|-------------------------------------------|----------------------------|
| <i>kLPIT_Channel0TimerInterruptEnable</i> | Channel 0 Timer interrupt. |
| <i>kLPIT_Channel1TimerInterruptEnable</i> | Channel 1 Timer interrupt. |
| <i>kLPIT_Channel2TimerInterruptEnable</i> | Channel 2 Timer interrupt. |
| <i>kLPIT_Channel3TimerInterruptEnable</i> | Channel 3 Timer interrupt. |

### 17.5.6 enum lpit\_status\_flags\_t

Note

Number of timer channels are SoC-specific. See the SoC Reference Manual.

Enumerator

|                                |                                 |
|--------------------------------|---------------------------------|
| <i>kLPIT_Channel0TimerFlag</i> | Channel 0 Timer interrupt flag. |
| <i>kLPIT_Channel1TimerFlag</i> | Channel 1 Timer interrupt flag. |
| <i>kLPIT_Channel2TimerFlag</i> | Channel 2 Timer interrupt flag. |
| <i>kLPIT_Channel3TimerFlag</i> | Channel 3 Timer interrupt flag. |

## 17.6 Function Documentation

### 17.6.1 void LPIT\_Init ( LPIT\_Type \* *base*, const lpit\_config\_t \* *config* )

This function issues a software reset to reset all channels and registers except the Module Control register.

Note

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

Parameters

|               |                                              |
|---------------|----------------------------------------------|
| <i>base</i>   | LPIT peripheral base address.                |
| <i>config</i> | Pointer to the user configuration structure. |

### 17.6.2 void LPIT\_Deinit ( LPIT\_Type \* *base* )

Parameters

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

### 17.6.3 void LPIT\_GetDefaultConfig ( lpit\_config\_t \* *config* )

The default values are:

```
* config->enableRunInDebug = false;
* config->enableRunInDoze = false;
*
```

Parameters

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

#### 17.6.4 **status\_t LPIT\_SetupChannel ( LPIT\_Type \* *base*, lpit\_chnl\_t *channel*, const lpit\_chnl\_params\_t \* *chnlSetup* )**

This function sets up the operation mode to one of the options available in the enumeration [lpit\\_timer\\_modes\\_t](#). It sets the trigger source as either internal or external, trigger selection and the timers behaviour when a timeout occurs. It also chains the timer if a prior timer if requested by the user.

Parameters

|                  |                                   |
|------------------|-----------------------------------|
| <i>base</i>      | LPIT peripheral base address.     |
| <i>channel</i>   | Channel that is being configured. |
| <i>chnlSetup</i> | Configuration parameters.         |

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

Parameters

|             |                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPIT peripheral base address.                                                                                        |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">lpit_interrupt_enable_t</a> |

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

Parameters

|             |                                                                                                                      |
|-------------|----------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPIT peripheral base address.                                                                                        |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">lpit_interrupt_enable_t</a> |

17.6.7 **static uint32\_t LPIT\_GetEnabledInterrupts ( LPIT\_Type \* *base* )**  
[**inline**], [**static**]

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [lpit\\_interrupt\\_enable\\_t](#)

### 17.6.8 static uint32\_t LPIT\_GetStatusFlags ( LPIT\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration [lpit\\_status\\_flags\\_t](#)

### 17.6.9 static void LPIT\_ClearStatusFlags ( LPIT\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPIT peripheral base address.                                                                                     |
| <i>mask</i> | The status flags to clear. This is a logical OR of members of the enumeration <a href="#">lpit_status_flags_t</a> |

### 17.6.10 static void LPIT\_SetTimerPeriod ( LPIT\_Type \* *base*, lpit\_chnl\_t *channel*, uint32\_t *ticks* ) [inline], [static]

Timers begin counting down from the value set by this function until it reaches 0, at which point it generates an interrupt and loads 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

User can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

|                |                                 |
|----------------|---------------------------------|
| <i>base</i>    | LPIT peripheral base address.   |
| <i>channel</i> | Timer channel number.           |
| <i>ticks</i>   | Timer period in units of ticks. |

### 17.6.11 static uint32\_t LPIT\_GetCurrentTimerCount ( LPIT\_Type \* *base*, lpit\_chnl\_t *channel* ) [inline], [static]

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

User can call the utility macros provided in fsl\_common.h to convert ticks to microseconds or milliseconds.

Parameters

|                |                               |
|----------------|-------------------------------|
| <i>base</i>    | LPIT peripheral base address. |
| <i>channel</i> | Timer channel number.         |

Returns

Current timer counting value in ticks.

### 17.6.12 static void LPIT\_StartTimer ( LPIT\_Type \* *base*, lpit\_chnl\_t *channel* ) [inline], [static]

After calling this function, timers load the period value and count down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

|                |                               |
|----------------|-------------------------------|
| <i>base</i>    | LPIT peripheral base address. |
| <i>channel</i> | Timer channel number.         |

### 17.6.13 static void LPIT\_StopTimer ( LPIT\_Type \* *base*, lpit\_chnl\_t *channel* ) [inline], [static]

Parameters

|                |                               |
|----------------|-------------------------------|
| <i>base</i>    | LPIT peripheral base address. |
| <i>channel</i> | Timer channel number.         |

#### 17.6.14 static void LPIT\_Reset( LPIT\_Type \* *base* ) [inline], [static]

This resets all channels and registers except the Module Control Register.

Parameters

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

# Chapter 18

## LPSPI: Low Power Serial Peripheral Interface

### 18.1 Overview

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

### Modules

- [LPSPI CMSIS Driver](#)
- [LPSPI FreeRTOS Driver](#)
- [LPSPI Peripheral driver](#)
- [LPSPI eDMA Driver](#)

## 18.2 LPSPI Peripheral driver

### 18.2.1 Overview

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

### 18.2.2 Function groups

#### 18.2.2.1 LPSPI Initialization and De-initialization

This function group initializes the default configuration structure for master and slave, initializes the LPSPI master with a master configuration, initializes the LPSPI slave with a slave configuration, and de-initializes the LPSPI module.

#### 18.2.2.2 LPSPI Basic Operation

This function group enables/disables the LPSPI module both interrupt and DMA, gets the data register address for the DMA transfer, sets master and slave, starts and stops the transfer, and so on.

#### 18.2.2.3 LPSPI Transfer Operation

This function group controls the transfer, master send/receive data, and slave send/receive data.

#### 18.2.2.4 LPSPI Status Operation

This function group gets/clears the LPSPI status.

#### 18.2.2.5 LPSPI Block Transfer Operation

This function group transfers a block of data, gets the transfer status, and aborts the transfer.

### 18.2.3 Typical use case

#### 18.2.3.1 Master Operation

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/lpspi

### 18.2.3.2 Slave Operation

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/lpspi

## Data Structures

- struct `lpspi_master_config_t`  
*LPSPI master configuration structure. [More...](#)*
- struct `lpspi_slave_config_t`  
*LPSPI slave configuration structure. [More...](#)*
- struct `lpspi_transfer_t`  
*LPSPI master/slave transfer structure. [More...](#)*
- struct `lpspi_master_handle_t`  
*LPSPI master transfer handle structure used for transactional API. [More...](#)*
- struct `lpspi_slave_handle_t`  
*LPSPI slave transfer handle structure used for transactional API. [More...](#)*

## Macros

- #define `LPSPI_DUMMY_DATA` (0x00U)  
*LPSPI dummy data if no Tx data.*
- #define `SPI_RETRY_TIMES` 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/  
*Retry times for waiting flag.*
- #define `LPSPI_MASTER_PCS_SHIFT` (4U)  
*LPSPI master PCS shift macro , internal used.*
- #define `LPSPI_MASTER_PCS_MASK` (0xF0U)  
*LPSPI master PCS shift macro , internal used.*
- #define `LPSPI_SLAVE_PCS_SHIFT` (4U)  
*LPSPI slave PCS shift macro , internal used.*
- #define `LPSPI_SLAVE_PCS_MASK` (0xF0U)  
*LPSPI slave PCS shift macro , internal used.*

## Typedefs

- typedef void(\* `lpspi_master_transfer_callback_t` )(LPSPI\_Type \*base, lpspi\_master\_handle\_t \*handle, `status_t` status, void \*userData)  
*Master completion callback function pointer type.*
- typedef void(\* `lpspi_slave_transfer_callback_t` )(LPSPI\_Type \*base, lpspi\_slave\_handle\_t \*handle, `status_t` status, void \*userData)  
*Slave completion callback function pointer type.*

## Enumerations

- enum {
   
kStatus\_LPSPI\_Busy = MAKE\_STATUS(kStatusGroup\_LPSPI, 0),
   
kStatus\_LPSPI\_Error = MAKE\_STATUS(kStatusGroup\_LPSPI, 1),
   
kStatus\_LPSPI\_Idle = MAKE\_STATUS(kStatusGroup\_LPSPI, 2),
   
kStatus\_LPSPI\_OutOfRange = MAKE\_STATUS(kStatusGroup\_LPSPI, 3),
   
kStatus\_LPSPI\_Timeout = MAKE\_STATUS(kStatusGroup\_LPSPI, 4) }
   
*Status for the LPSPI driver.*
  - enum \_lpspi\_flags {
   
kLPSPI\_TxDataRequestFlag = LPSPI\_SR\_TDF\_MASK,
   
kLPSPI\_RxDataReadyFlag = LPSPI\_SR\_RDF\_MASK,
   
kLPSPI\_WordCompleteFlag = LPSPI\_SR\_WCF\_MASK,
   
kLPSPI\_FrameCompleteFlag = LPSPI\_SR\_FCF\_MASK,
   
kLPSPI\_TransferCompleteFlag = LPSPI\_SR\_TCF\_MASK,
   
kLPSPI\_TransmitErrorFlag = LPSPI\_SR\_TEF\_MASK,
   
kLPSPI\_ReceiveErrorFlag = LPSPI\_SR\_REF\_MASK,
   
kLPSPI\_DataMatchFlag = LPSPI\_SR\_DMF\_MASK,
   
kLPSPI\_ModuleBusyFlag = LPSPI\_SR\_MBFI\_MASK,
   
kLPSPI\_AllStatusFlag }
   
*LPSPI status flags in SPIx\_SR register.*
  - enum \_lpspi\_interrupt\_enable {
   
kLPSPI\_TxInterruptEnable = LPSPI\_IER\_TDIE\_MASK,
   
kLPSPI\_RxInterruptEnable = LPSPI\_IER\_RDIE\_MASK,
   
kLPSPI\_WordCompleteInterruptEnable = LPSPI\_IER\_WCIE\_MASK,
   
kLPSPI\_FrameCompleteInterruptEnable = LPSPI\_IER\_FCIE\_MASK,
   
kLPSPI\_TransferCompleteInterruptEnable = LPSPI\_IER\_TCIE\_MASK,
   
kLPSPI\_TransmitErrorInterruptEnable = LPSPI\_IER\_TEIE\_MASK,
   
kLPSPI\_ReceiveErrorInterruptEnable = LPSPI\_IER\_REIE\_MASK,
   
kLPSPI\_DataMatchInterruptEnable = LPSPI\_IER\_DMIE\_MASK,
   
kLPSPI\_AllInterruptEnable }
   
*LPSPI interrupt source.*
  - enum \_lpspi\_dma\_enable {
   
kLPSPI\_TxDmaEnable = LPSPI\_DER\_TDDE\_MASK,
   
kLPSPI\_RxDmaEnable = LPSPI\_DER\_RDDE\_MASK }
  - enum lpspi\_master\_slave\_mode\_t {
   
kLPSPI\_Master = 1U,
   
kLPSPI\_Slave = 0U }
  - enum lpspi\_which\_pcs\_t {
   
kLPSPI\_Pcs0 = 0U,
   
kLPSPI\_Pcs1 = 1U,
   
kLPSPI\_Pcs2 = 2U,
   
kLPSPI\_Pcs3 = 3U }
- LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure).*

- enum `lpspi_pcs_polarity_config_t` {
   
  `kLPSPI_PcsActiveHigh` = 1U,
   
  `kLPSPI_PcsActiveLow` = 0U }

*LPSPI Peripheral Chip Select (PCS) Polarity configuration.*

- enum `_lpspi_pcs_polarity` {
   
  `kLPSPI_Pcs0ActiveLow` = 1U << 0,
   
  `kLPSPI_Pcs1ActiveLow` = 1U << 1,
   
  `kLPSPI_Pcs2ActiveLow` = 1U << 2,
   
  `kLPSPI_Pcs3ActiveLow` = 1U << 3,
   
  `kLPSPI_PcsAllActiveLow` = 0xFU }

*LPSPI Peripheral Chip Select (PCS) Polarity.*

- enum `lpspi_clock_polarity_t` {
   
  `kLPSPI_ClockPolarityActiveHigh` = 0U,
   
  `kLPSPI_ClockPolarityActiveLow` = 1U }

*LPSPI clock polarity configuration.*

- enum `lpspi_clock_phase_t` {
   
  `kLPSPI_ClockPhaseFirstEdge` = 0U,
   
  `kLPSPI_ClockPhaseSecondEdge` = 1U }

*LPSPI clock phase configuration.*

- enum `lpspi_shift_direction_t` {
   
  `kLPSPI_MsbFirst` = 0U,
   
  `kLPSPI_LsbFirst` = 1U }

*LPSPI data shifter direction options.*

- enum `lpspi_host_request_select_t` {
   
  `kLPSPI_HostReqExtPin` = 0U,
   
  `kLPSPI_HostReqInternalTrigger` = 1U }

*LPSPI Host Request select configuration.*

- enum `lpspi_match_config_t` {
   
  `kLPSI_MatchDisabled` = 0x0U,
   
  `kLPSI_1stWordEqualsM0orM1` = 0x2U,
   
  `kLPSI_AnyWordEqualsM0orM1` = 0x3U,
   
  `kLPSI_1stWordEqualsM0and2ndWordEqualsM1` = 0x4U,
   
  `kLPSI_AnyWordEqualsM0andNxtWordEqualsM1` = 0x5U,
   
  `kLPSI_1stWordAndM1EqualsM0andM1` = 0x6U,
   
  `kLPSI_AnyWordAndM1EqualsM0andM1` = 0x7U }

*LPSPI Match configuration options.*

- enum `lpspi_pin_config_t` {
   
  `kLPSPI_SdiInSdoOut` = 0U,
   
  `kLPSPI_SdiInSdiOut` = 1U,
   
  `kLPSPI_SdoInSdoOut` = 2U,
   
  `kLPSPI_SdoInSdiOut` = 3U }

*LPSPI pin (SDO and SDI) configuration.*

- enum `lpspi_data_out_config_t` {
   
  `kLpspiDataOutRetained` = 0U,
   
  `kLpspiDataOutTristate` = 1U }

*LPSPI data output configuration.*

- enum `lpspi_transfer_width_t` {

- ```

kLPSPI_SingleBitXfer = 0U,
kLPSPI_TwoBitXfer = 1U,
kLPSPI_FourBitXfer = 2U }
    LPSPI transfer width configuration.
• enum lpspi\_delay\_type\_t {
    kLPSPI_PcsToSck = 1U,
    kLPSPI_LastSckToPcs,
    kLPSPI_BetweenTransfer }

    LPSPI delay type selection.
• enum \_lpspi\_transfer\_config\_flag\_for\_master {
    kLPSPI_MasterPcs0 = 0U << LPSPI_MASTER_PCS_SHIFT,
    kLPSPI_MasterPcs1 = 1U << LPSPI_MASTER_PCS_SHIFT,
    kLPSPI_MasterPcs2 = 2U << LPSPI_MASTER_PCS_SHIFT,
    kLPSPI_MasterPcs3 = 3U << LPSPI_MASTER_PCS_SHIFT,
    kLPSPI_MasterPcsContinuous = 1U << 20,
    kLPSPI_MasterByteSwap }

    Use this enumeration for LPSPI master transfer configFlags.
• enum \_lpspi\_transfer\_config\_flag\_for\_slave {
    kLPSPI_SlavePcs0 = 0U << LPSPI_SLAVE_PCS_SHIFT,
    kLPSPI_SlavePcs1 = 1U << LPSPI_SLAVE_PCS_SHIFT,
    kLPSPI_SlavePcs2 = 2U << LPSPI_SLAVE_PCS_SHIFT,
    kLPSPI_SlavePcs3 = 3U << LPSPI_SLAVE_PCS_SHIFT,
    kLPSPI_SlaveByteSwap }

    Use this enumeration for LPSPI slave transfer configFlags.
• enum \_lpspi\_transfer\_state {
    kLPSPI_Idle = 0x0U,
    kLPSPI_Busy,
    kLPSPI_Error }

    LPSPI transfer state, which is used for LPSPI transactional API state machine.

```

Variables

- volatile uint8_t [g_lpspiDummyData](#) []
 Global variable for dummy data value setting.

Driver version

- #define [FSL_LPSPI_DRIVER_VERSION](#) (MAKE_VERSION(2, 2, 1))
 LPSPI driver version.

Initialization and deinitialization

- void [LPSPI_MasterInit](#) (LPSPI_Type *base, const [lpspi_master_config_t](#) *masterConfig, uint32_t srcClock_Hz)

- **void LPSPI_MasterGetDefaultConfig (lpspi_master_config_t *masterConfig)**
Sets the lpspi_master_config_t structure to default values.
- **void LPSPI_SlaveInit (LPSPI_Type *base, const lpspi_slave_config_t *slaveConfig)**
LPSPI slave configuration.
- **void LPSPI_SlaveGetDefaultConfig (lpspi_slave_config_t *slaveConfig)**
Sets the lpspi_slave_config_t structure to default values.
- **void LPSPI_Deinit (LPSPI_Type *base)**
De-initializes the LPSPI peripheral.
- **void LPSPI_Reset (LPSPI_Type *base)**
Restores the LPSPI peripheral to reset state.
- **uint32_t LPSPIGetInstance (LPSPI_Type *base)**
Get the LPSPI instance from peripheral base address.
- **static void LPSPI_Enable (LPSPI_Type *base, bool enable)**
Enables the LPSPI peripheral and sets the MCR MDIS to 0.

Status

- **static uint32_t LPSPI_GetStatusFlags (LPSPI_Type *base)**
Gets the LPSPI status flag state.
- **static uint8_t LPSPI_GetTxFifoSize (LPSPI_Type *base)**
Gets the LPSPI Tx FIFO size.
- **static uint8_t LPSPI_GetRxFifoSize (LPSPI_Type *base)**
Gets the LPSPI Rx FIFO size.
- **static uint32_t LPSPI_GetTxFifoCount (LPSPI_Type *base)**
Gets the LPSPI Tx FIFO count.
- **static uint32_t LPSPI_GetRxFifoCount (LPSPI_Type *base)**
Gets the LPSPI Rx FIFO count.
- **static void LPSPI_ClearStatusFlags (LPSPI_Type *base, uint32_t statusFlags)**
Clears the LPSPI status flag.

Interrupts

- **static void LPSPI_EnableInterrupts (LPSPI_Type *base, uint32_t mask)**
Enables the LPSPI interrupts.
- **static void LPSPI_DisableInterrupts (LPSPI_Type *base, uint32_t mask)**
Disables the LPSPI interrupts.

DMA Control

- **static void LPSPI_EnableDMA (LPSPI_Type *base, uint32_t mask)**
Enables the LPSPI DMA request.
- **static void LPSPI_DisableDMA (LPSPI_Type *base, uint32_t mask)**
Disables the LPSPI DMA request.
- **static uint32_t LPSPI_GetTxRegisterAddress (LPSPI_Type *base)**
Gets the LPSPI Transmit Data Register address for a DMA operation.
- **static uint32_t LPSPI_GetRxRegisterAddress (LPSPI_Type *base)**

Gets the LPSPI Receive Data Register address for a DMA operation.

Bus Operations

- bool [LPSPI_CheckTransferArgument](#) (LPSPI_Type *base, lpspi_transfer_t *transfer, bool isEdma)
Check the argument for transfer.
- static void [LPSPI_SetMasterSlaveMode](#) (LPSPI_Type *base, lpspi_master_slave_mode_t mode)
Configures the LPSPI for either master or slave.
- static void [LPSPI_SelectTransferPCS](#) (LPSPI_Type *base, lpspi_which_pcs_t select)
Configures the peripheral chip select used for the transfer.
- static void [LPSPI_SetPCSContinous](#) (LPSPI_Type *base, bool IsContinous)
Set the PCS signal to continuous or uncontinuous mode.
- static bool [LPSPI_IsMaster](#) (LPSPI_Type *base)
Returns whether the LPSPI module is in master mode.
- static void [LPSPI_FlushFifo](#) (LPSPI_Type *base, bool flushTxFifo, bool flushRxFifo)
Flushes the LPSPI FIFOs.
- static void [LPSPI_SetFifoWatermarks](#) (LPSPI_Type *base, uint32_t txWater, uint32_t rxWater)
Sets the transmit and receive FIFO watermark values.
- static void [LPSPI_SetAllPcsPolarity](#) (LPSPI_Type *base, uint32_t mask)
Configures all LPSPI peripheral chip select polarities simultaneously.
- static void [LPSPI_SetFrameSize](#) (LPSPI_Type *base, uint32_t frameSize)
Configures the frame size.
- uint32_t [LPSPI_MasterSetBaudRate](#) (LPSPI_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz, uint32_t *tcrPrescaleValue)
Sets the LPSPI baud rate in bits per second.
- void [LPSPI_MasterSetDelayScaler](#) (LPSPI_Type *base, uint32_t scaler, lpspi_delay_type_t whichDelay)
Manually configures a specific LPSPI delay parameter (module must be disabled to change the delay values).
- uint32_t [LPSPI_MasterSetDelayTimes](#) (LPSPI_Type *base, uint32_t delayTimeInNanoSec, lpspi_delay_type_t whichDelay, uint32_t srcClock_Hz)
Calculates the delay based on the desired delay input in nanoseconds (module must be disabled to change the delay values).
- static void [LPSPI_WriteData](#) (LPSPI_Type *base, uint32_t data)
Writes data into the transmit data buffer.
- static uint32_t [LPSPI_ReadData](#) (LPSPI_Type *base)
Reads data from the data buffer.
- void [LPSPI_SetDummyData](#) (LPSPI_Type *base, uint8_t dummyData)
Set up the dummy data.

Transactional

- void [LPSPI_MasterTransferCreateHandle](#) (LPSPI_Type *base, lpspi_master_handle_t *handle, lpspi_master_transfer_callback_t callback, void *userData)
Initializes the LPSPI master handle.
- status_t [LPSPI_MasterTransferBlocking](#) (LPSPI_Type *base, lpspi_transfer_t *transfer)
LPSPI master transfer data using a polling method.

- `status_t LPSPI_MasterTransferNonBlocking (LPSPI_Type *base, lpspi_master_handle_t *handle, lpspi_transfer_t *transfer)`
LPSPI master transfer data using an interrupt method.
- `status_t LPSPI_MasterTransferGetCount (LPSPI_Type *base, lpspi_master_handle_t *handle, size_t *count)`
Gets the master transfer remaining bytes.
- `void LPSPI_MasterTransferAbort (LPSPI_Type *base, lpspi_master_handle_t *handle)`
LPSPI master abort transfer which uses an interrupt method.
- `void LPSPI_MasterTransferHandleIRQ (LPSPI_Type *base, lpspi_master_handle_t *handle)`
LPSPI Master IRQ handler function.
- `void LPSPI_SlaveTransferCreateHandle (LPSPI_Type *base, lpspi_slave_handle_t *handle, lpspi_slave_transfer_callback_t callback, void *userData)`
Initializes the LPSPI slave handle.
- `status_t LPSPI_SlaveTransferNonBlocking (LPSPI_Type *base, lpspi_slave_handle_t *handle, lpspi_transfer_t *transfer)`
LPSPI slave transfer data using an interrupt method.
- `status_t LPSPI_SlaveTransferGetCount (LPSPI_Type *base, lpspi_slave_handle_t *handle, size_t *count)`
Gets the slave transfer remaining bytes.
- `void LPSPI_SlaveTransferAbort (LPSPI_Type *base, lpspi_slave_handle_t *handle)`
LPSPI slave aborts a transfer which uses an interrupt method.
- `void LPSPI_SlaveTransferHandleIRQ (LPSPI_Type *base, lpspi_slave_handle_t *handle)`
LPSPI Slave IRQ handler function.

18.2.4 Data Structure Documentation

18.2.4.1 struct lpspi_master_config_t

Data Fields

- `uint32_t baudRate`
Baud Rate for LPSPI.
- `uint32_t bitsPerFrame`
Bits per frame, minimum 8, maximum 4096.
- `lpspi_clock_polarity_t cpol`
Clock polarity.
- `lpspi_clock_phase_t cpha`
Clock phase.
- `lpspi_shift_direction_t direction`
MSB or LSB data shift direction.
- `uint32_t pcsToSckDelayInNanoSec`
PCS to SCK delay time in nanoseconds, setting to 0 sets the minimum delay.
- `uint32_t lastSckToPcsDelayInNanoSec`
Last SCK to PCS delay time in nanoseconds, setting to 0 sets the minimum delay.
- `uint32_t betweenTransferDelayInNanoSec`
After the SCK delay time with nanoseconds, setting to 0 sets the minimum delay.
- `lpspi_which_pcs_t whichPcs`
Desired Peripheral Chip Select (PCS).

- [lpspi_pcs_polarity_config_t pcsActiveHighOrLow](#)
Desired PCS active high or low.
- [lpspi_pin_config_t pinCfg](#)
Configures which pins are used for input and output data during single bit transfers.
- [lpspi_data_out_config_t dataOutConfig](#)
Configures if the output data is tristated between accesses (LPSPI_PCS is negated).

Field Documentation

- (1) [uint32_t lpspi_master_config_t::baudRate](#)
- (2) [uint32_t lpspi_master_config_t::bitsPerFrame](#)
- (3) [lpspi_clock_polarity_t lpspi_master_config_t::cpol](#)
- (4) [lpspi_clock_phase_t lpspi_master_config_t::cpha](#)
- (5) [lpspi_shift_direction_t lpspi_master_config_t::direction](#)
- (6) [uint32_t lpspi_master_config_t::pcsToSckDelayInNanoSec](#)

It sets the boundary value if out of range.

- (7) [uint32_t lpspi_master_config_t::lastSckToPcsDelayInNanoSec](#)

It sets the boundary value if out of range.

- (8) [uint32_t lpspi_master_config_t::betweenTransferDelayInNanoSec](#)

It sets the boundary value if out of range.

- (9) [lpspi_which_pcs_t lpspi_master_config_t::whichPcs](#)
- (10) [lpspi_pin_config_t lpspi_master_config_t::pinCfg](#)
- (11) [lpspi_data_out_config_t lpspi_master_config_t::dataOutConfig](#)

18.2.4.2 struct lpspi_slave_config_t

Data Fields

- [uint32_t bitsPerFrame](#)
Bits per frame, minimum 8, maximum 4096.
- [lpspi_clock_polarity_t cpol](#)
Clock polarity.
- [lpspi_clock_phase_t cpha](#)
Clock phase.
- [lpspi_shift_direction_t direction](#)
MSB or LSB data shift direction.
- [lpspi_which_pcs_t whichPcs](#)
Desired Peripheral Chip Select (pcs)

- [lpspi_pcs_polarity_config_t](#) `pcsActiveHighOrLow`
Desired PCS active high or low.
- [lpspi_pin_config_t](#) `pinCfg`
Configures which pins are used for input and output data during single bit transfers.
- [lpspi_data_out_config_t](#) `dataOutConfig`
Configures if the output data is tristated between accesses (LPSPI_PCS is negated).

Field Documentation

- (1) `uint32_t lpspi_slave_config_t::bitsPerFrame`
- (2) `lpspi_clock_polarity_t lpspi_slave_config_t::cpol`
- (3) `lpspi_clock_phase_t lpspi_slave_config_t::cpha`
- (4) `lpspi_shift_direction_t lpspi_slave_config_t::direction`
- (5) `lpspi_pin_config_t lpspi_slave_config_t::pinCfg`
- (6) `lpspi_data_out_config_t lpspi_slave_config_t::dataOutConfig`

18.2.4.3 struct lpspi_transfer_t**Data Fields**

- `uint8_t * txData`
Send buffer.
- `uint8_t * rxData`
Receive buffer.
- `volatile size_t dataSize`
Transfer bytes.
- `uint32_t configFlags`
Transfer transfer configuration flags.

Field Documentation

- (1) `uint8_t* lpspi_transfer_t::txData`
- (2) `uint8_t* lpspi_transfer_t::rxData`
- (3) `volatile size_t lpspi_transfer_t::dataSize`
- (4) `uint32_t lpspi_transfer_t::configFlags`

Set from _lpspi_transfer_config_flag_for_master if the transfer is used for master or _lpspi_transfer_config_flag_for_slave enumeration if the transfer is used for slave.

18.2.4.4 struct _lpspi_master_handle

Forward declaration of the `_lpspi_master_handle` typedefs.

Data Fields

- volatile bool **isPcsContinuous**
Is PCS continuous in transfer.
- volatile bool **writeTcrInIsr**
A flag that whether should write TCR in ISR.
- volatile bool **isByteSwap**
A flag that whether should byte swap.
- volatile bool **isTxMask**
A flag that whether TCR[TXMSK] is set.
- volatile uint16_t **bytesPerFrame**
Number of bytes in each frame.
- volatile uint8_t **fifoSize**
FIFO dataSize.
- volatile uint8_t **rxWatermark**
Rx watermark.
- volatile uint8_t **bytesEachWrite**
Bytes for each write TDR.
- volatile uint8_t **bytesEachRead**
Bytes for each read RDR.
- uint8_t *volatile **txData**
Send buffer.
- uint8_t *volatile **rxData**
Receive buffer.
- volatile size_t **txRemainingByteCount**
Number of bytes remaining to send.
- volatile size_t **rxRemainingByteCount**
Number of bytes remaining to receive.
- volatile uint32_t **writeRegRemainingTimes**
Write TDR register remaining times.
- volatile uint32_t **readRegRemainingTimes**
Read RDR register remaining times.
- uint32_t **totalByteCount**
Number of transfer bytes.
- uint32_t **txBuffIfNull**
Used if the txData is NULL.
- volatile uint8_t **state**
LPSPI transfer state , _lpspi_transfer_state.
- **lpspi_master_transfer_callback_t callback**
Completion callback.
- void * **userData**
Callback user data.

Field Documentation

- (1) **volatile bool lpspi_master_handle_t::isPcsContinuous**
- (2) **volatile bool lpspi_master_handle_t::writeTcrInIsr**
- (3) **volatile bool lpspi_master_handle_t::isByteSwap**

- (4) volatile bool lpspi_master_handle_t::isTxMask
- (5) volatile uint8_t lpspi_master_handle_t::fifoSize
- (6) volatile uint8_t lpspi_master_handle_t::rxWatermark
- (7) volatile uint8_t lpspi_master_handle_t::bytesEachWrite
- (8) volatile uint8_t lpspi_master_handle_t::bytesEachRead
- (9) uint8_t* volatile lpspi_master_handle_t::txData
- (10) uint8_t* volatile lpspi_master_handle_t::rxData
- (11) volatile size_t lpspi_master_handle_t::txRemainingByteCount
- (12) volatile size_t lpspi_master_handle_t::rxRemainingByteCount
- (13) volatile uint32_t lpspi_master_handle_t::writeRegRemainingTimes
- (14) volatile uint32_t lpspi_master_handle_t::readRegRemainingTimes
- (15) uint32_t lpspi_master_handle_t::txBuffIfNull
- (16) volatile uint8_t lpspi_master_handle_t::state
- (17) lpspi_master_transfer_callback_t lpspi_master_handle_t::callback
- (18) void* lpspi_master_handle_t::userData

18.2.4.5 struct _lpspi_slave_handle

Forward declaration of the [_lpspi_slave_handle](#) typedefs.

Data Fields

- volatile bool [isByteSwap](#)
A flag that whether should byte swap.
- volatile uint8_t [fifoSize](#)
FIFO dataSize.
- volatile uint8_t [rxWatermark](#)
Rx watermark.
- volatile uint8_t [bytesEachWrite](#)
Bytes for each write TDR.
- volatile uint8_t [bytesEachRead](#)
Bytes for each read RDR.
- uint8_t *volatile [txData](#)
Send buffer.
- uint8_t *volatile [rxData](#)
Receive buffer.

- volatile size_t **txRemainingByteCount**
Number of bytes remaining to send.
- volatile size_t **rxRemainingByteCount**
Number of bytes remaining to receive.
- volatile uint32_t **writeRegRemainingTimes**
Write TDR register remaining times.
- volatile uint32_t **readRegRemainingTimes**
Read RDR register remaining times.
- uint32_t **totalByteCount**
Number of transfer bytes.
- volatile uint8_t **state**
LPSPI transfer state , _lpspi_transfer_state.
- volatile uint32_t **errorCount**
Error count for slave transfer.
- **lpspi_slave_transfer_callback_t callback**
Completion callback.
- void * **userData**
Callback user data.

Field Documentation

- (1) volatile bool **lpspi_slave_handle_t::isByteSwap**
- (2) volatile uint8_t **lpspi_slave_handle_t::fifoSize**
- (3) volatile uint8_t **lpspi_slave_handle_t::rxWatermark**
- (4) volatile uint8_t **lpspi_slave_handle_t::bytesEachWrite**
- (5) volatile uint8_t **lpspi_slave_handle_t::bytesEachRead**
- (6) uint8_t* volatile **lpspi_slave_handle_t::txData**
- (7) uint8_t* volatile **lpspi_slave_handle_t::rxData**
- (8) volatile size_t **lpspi_slave_handle_t::txRemainingByteCount**
- (9) volatile size_t **lpspi_slave_handle_t::rxRemainingByteCount**
- (10) volatile uint32_t **lpspi_slave_handle_t::writeRegRemainingTimes**
- (11) volatile uint32_t **lpspi_slave_handle_t::readRegRemainingTimes**
- (12) volatile uint8_t **lpspi_slave_handle_t::state**
- (13) volatile uint32_t **lpspi_slave_handle_t::errorCount**
- (14) **lpspi_slave_transfer_callback_t lpspi_slave_handle_t::callback**
- (15) void* **lpspi_slave_handle_t::userData**

18.2.5 Macro Definition Documentation

18.2.5.1 #define FSL_LPSPI_DRIVER_VERSION (MAKE_VERSION(2, 2, 1))

18.2.5.2 #define LPSPI_DUMMY_DATA (0x00U)

Dummy data used for tx if there is not txData.

18.2.5.3 #define SPI_RETRY_TIMES 0U /* Define to zero means keep waiting until the flag is assert/deassert. */

18.2.5.4 #define LPSPI_MASTER_PCS_SHIFT (4U)

18.2.5.5 #define LPSPI_MASTER_PCS_MASK (0xF0U)

18.2.5.6 #define LPSPI_SLAVE_PCS_SHIFT (4U)

18.2.5.7 #define LPSPI_SLAVE_PCS_MASK (0xF0U)

18.2.6 Typedef Documentation

18.2.6.1 typedef void(* lpspi_master_transfer_callback_t)(LPSPI_Type *base, lpspi_master_handle_t *handle, status_t status, void *userData)

Parameters

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

18.2.6.2 typedef void(* lpspi_slave_transfer_callback_t)(LPSPI_Type *base, lpspi_slave_handle_t *handle, status_t status, void *userData)

Parameters

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

18.2.7 Enumeration Type Documentation

18.2.7.1 anonymous enum

Enumerator

kStatus_LPSPI_Busy LPSPI transfer is busy.
kStatus_LPSPI_Error LPSPI driver error.
kStatus_LPSPI_Idle LPSPI is idle.
kStatus_LPSPI_OutOfRange LPSPI transfer out Of range.
kStatus_LPSPI_Timeout LPSPI timeout polling status flags.

18.2.7.2 enum _lpspi_flags

Enumerator

kLPSPI_TxDataRequestFlag Transmit data flag.
kLPSPI_RxDataReadyFlag Receive data flag.
kLPSPI_WordCompleteFlag Word Complete flag.
kLPSPI_FrameCompleteFlag Frame Complete flag.
kLPSPI_TransferCompleteFlag Transfer Complete flag.
kLPSPI_TransmitErrorFlag Transmit Error flag (FIFO underrun)
kLPSPI_ReceiveErrorFlag Receive Error flag (FIFO overrun)
kLPSPI_DataMatchFlag Data Match flag.
kLPSPI_ModuleBusyFlag Module Busy flag.
kLPSPI_AllStatusFlag Used for clearing all w1c status flags.

18.2.7.3 enum _lpspi_interrupt_enable

Enumerator

kLPSPI_TxInterruptEnable Transmit data interrupt enable.
kLPSPI_RxInterruptEnable Receive data interrupt enable.
kLPSPI_WordCompleteInterruptEnable Word complete interrupt enable.
kLPSPI_FrameCompleteInterruptEnable Frame complete interrupt enable.
kLPSPI_TransferCompleteInterruptEnable Transfer complete interrupt enable.

kLPSPI_TransmitErrorInterruptEnable Transmit error interrupt enable(FIFO underrun)
kLPSPI_ReceiveErrorInterruptEnable Receive Error interrupt enable (FIFO overrun)
kLPSPI_DataMatchInterruptEnable Data Match interrupt enable.
kLPSPI_AllInterruptEnable All above interrupts enable.

18.2.7.4 enum _lpspi_dma_enable

Enumerator

kLPSPI_TxDmaEnable Transmit data DMA enable.
kLPSPI_RxDmaEnable Receive data DMA enable.

18.2.7.5 enum lpspi_master_slave_mode_t

Enumerator

kLPSPI_Master LPSPI peripheral operates in master mode.
kLPSPI_Slave LPSPI peripheral operates in slave mode.

18.2.7.6 enum lpspi_which_pcs_t

Enumerator

kLPSPI_Pcs0 PCS[0].
kLPSPI_Pcs1 PCS[1].
kLPSPI_Pcs2 PCS[2].
kLPSPI_Pcs3 PCS[3].

18.2.7.7 enum lpspi_pcs_polarity_config_t

Enumerator

kLPSPI_PcsActiveHigh PCS Active High (idles low)
kLPSPI_PcsActiveLow PCS Active Low (idles high)

18.2.7.8 enum _lpspi_pcs_polarity

Enumerator

kLPSPI_Pcs0ActiveLow Pcs0 Active Low (idles high).
kLPSPI_Pcs1ActiveLow Pcs1 Active Low (idles high).

kLPSPI_Pcs2ActiveLow Pcs2 Active Low (idles high).

kLPSPI_Pcs3ActiveLow Pcs3 Active Low (idles high).

kLPSPI_PcsAllActiveLow Pcs0 to Pcs5 Active Low (idles high).

18.2.7.9 enum lpspi_clock_polarity_t

Enumerator

kLPSPI_ClockPolarityActiveHigh CPOL=0. Active-high LPSPI clock (idles low)

kLPSPI_ClockPolarityActiveLow CPOL=1. Active-low LPSPI clock (idles high)

18.2.7.10 enum lpspi_clock_phase_t

Enumerator

kLPSPI_ClockPhaseFirstEdge CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.

kLPSPI_ClockPhaseSecondEdge CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

18.2.7.11 enum lpspi_shift_direction_t

Enumerator

kLPSPI_MsbFirst Data transfers start with most significant bit.

kLPSPI_LsbFirst Data transfers start with least significant bit.

18.2.7.12 enum lpspi_host_request_select_t

Enumerator

kLPSPI_HostReqExtPin Host Request is an ext pin.

kLPSPI_HostReqInternalTrigger Host Request is an internal trigger.

18.2.7.13 enum lpspi_match_config_t

Enumerator

kLPSI_MatchDisabled LPSPI Match Disabled.

kLPSI_1stWordEqualsM0orM1 LPSPI Match Enabled.

kLPSI_AnyWordEqualsM0orM1 LPSPI Match Enabled.

kLPSI_1stWordEqualsM0and2ndWordEqualsM1 LPSPI Match Enabled.
kLPSI_AnyWordEqualsM0andNxtWordEqualsM1 LPSPI Match Enabled.
kLPSI_1stWordAndM1EqualsM0andM1 LPSPI Match Enabled.
kLPSI_AnyWordAndM1EqualsM0andM1 LPSPI Match Enabled.

18.2.7.14 enum lpspi_pin_config_t

Enumerator

kLPSPISdiInSdoOut LPSPI SDI input, SDO output.
kLPSPISdiInSdiOut LPSPI SDI input, SDI output.
kLPSPISdoInSdoOut LPSPI SDO input, SDO output.
kLPSPISdoInSdiOut LPSPI SDO input, SDI output.

18.2.7.15 enum lpspi_data_out_config_t

Enumerator

kLpspiDataOutRetained Data out retains last value when chip select is de-asserted.
kLpspiDataOutTristate Data out is tristated when chip select is de-asserted.

18.2.7.16 enum lpspi_transfer_width_t

Enumerator

kLPSPISingleBitXfer 1-bit shift at a time, data out on SDO, in on SDI (normal mode)
kLPSPITwoBitXfer 2-bits shift out on SDO/SDI and in on SDO/SDI
kLPSPIFourBitXfer 4-bits shift out on SDO/SDI/PCS[3:2] and in on SDO/SDI/PCS[3:2]

18.2.7.17 enum lpspi_delay_type_t

Enumerator

kLPSPIPcsToSck PCS-to-SCK delay.
kLPSPILastSckToPcs Last SCK edge to PCS delay.
kLPSPIBetweenTransfer Delay between transfers.

18.2.7.18 enum _lpspi_transfer_config_flag_for_master

Enumerator

kLPSPI_MasterPcs0 LPSPI master transfer use PCS0 signal.

kLPSPI_MasterPcs1 LPSPI master transfer use PCS1 signal.

kLPSPI_MasterPcs2 LPSPI master transfer use PCS2 signal.

kLPSPI_MasterPcs3 LPSPI master transfer use PCS3 signal.

kLPSPI_MasterPcsContinuous Is PCS signal continuous.

kLPSPI_MasterByteSwap Is master swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

1. If you set bitPerFrame = 8 , no matter the kLPSPI_MasterByteSwap flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
2. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPI_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_MasterByteSwap flag.
3. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPI_MasterByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_MasterByteSwap flag.

18.2.7.19 enum _lpspi_transfer_config_flag_for_slave

Enumerator

kLPSPI_SlavePcs0 LPSPI slave transfer use PCS0 signal.

kLPSPI_SlavePcs1 LPSPI slave transfer use PCS1 signal.

kLPSPI_SlavePcs2 LPSPI slave transfer use PCS2 signal.

kLPSPI_SlavePcs3 LPSPI slave transfer use PCS3 signal.

kLPSPI_SlaveByteSwap Is slave swap the byte. For example, when want to send data 1 2 3 4 5 6 7 8 (suppose you set lpspi_shift_direction_t to MSB).

1. If you set bitPerFrame = 8 , no matter the kLPSPI_SlaveByteSwap flag is used or not, the waveform is 1 2 3 4 5 6 7 8.
2. If you set bitPerFrame = 16 : (1) the waveform is 2 1 4 3 6 5 8 7 if you do not use the kLPSPI_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_SlaveByteSwap flag.
3. If you set bitPerFrame = 32 : (1) the waveform is 4 3 2 1 8 7 6 5 if you do not use the kLPSPI_SlaveByteSwap flag. (2) the waveform is 1 2 3 4 5 6 7 8 if you use the kLPSPI_SlaveByteSwap flag.

18.2.7.20 enum _lpspi_transfer_state

Enumerator

kLPSPI_Idle Nothing in the transmitter/receiver.

kLPSPI_Busy Transfer queue is not finished.
kLPSPI_Error Transfer error.

18.2.8 Function Documentation

18.2.8.1 void LPSPI_MasterInit (**LPSPI_Type** * *base*, const **lpspi_master_config_t** * *masterConfig*, uint32_t *srcClock_Hz*)

Parameters

<i>base</i>	LPSPI peripheral address.
<i>masterConfig</i>	Pointer to structure lpspi_master_config_t .
<i>srcClock_Hz</i>	Module source input clock in Hertz

18.2.8.2 void LPSPI_MasterGetDefaultConfig (**lpspi_master_config_t** * *masterConfig*)

This API initializes the configuration structure for [LPSPI_MasterInit\(\)](#). The initialized structure can remain unchanged in [LPSPI_MasterInit\(\)](#), or can be modified before calling the [LPSPI_MasterInit\(\)](#). Example:

```
* lpspi_master_config_t masterConfig;
* LPSPI_MasterGetDefaultConfig(&masterConfig);
*
```

Parameters

<i>masterConfig</i>	pointer to lpspi_master_config_t structure
---------------------	--

18.2.8.3 void LPSPI_SlaveInit (**LPSPI_Type** * *base*, const **lpspi_slave_config_t** * *slaveConfig*)

Parameters

<i>base</i>	LPSPI peripheral address.
<i>slaveConfig</i>	Pointer to a structure lpspi_slave_config_t .

18.2.8.4 void LPSPI_SlaveGetDefaultConfig (**lpspi_slave_config_t** * *slaveConfig*)

This API initializes the configuration structure for [LPSPI_SlaveInit\(\)](#). The initialized structure can remain unchanged in [LPSPI_SlaveInit\(\)](#) or can be modified before calling the [LPSPI_SlaveInit\(\)](#). Example:

```
* lpspi_slave_config_t slaveConfig;
* LPSPI_SlaveGetDefaultConfig(&slaveConfig);
*
```

Parameters

<i>slaveConfig</i>	pointer to lpspi_slave_config_t structure.
--------------------	--

18.2.8.5 void LPSPI_Deinit (LPSPI_Type * *base*)

Call this API to disable the LPSPI clock.

Parameters

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

18.2.8.6 void LPSPI_Reset (LPSPI_Type * *base*)

Note that this function sets all registers to reset state. As a result, the LPSPI module can't work after calling this API.

Parameters

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

18.2.8.7 uint32_t LPSPIGetInstance (LPSPI_Type * *base*)

Parameters

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

Returns

LPSPI instance.

18.2.8.8 static void LPSPI_Enable (LPSPI_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

18.2.8.9 static uint32_t LPSPI_GetStatusFlags (LPSPI_Type * *base*) [inline], [static]

Parameters

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

Returns

The LPSPI status(in SR register).

18.2.8.10 static uint8_t LPSPI_GetTxFifoSize (LPSPI_Type * *base*) [inline], [static]

Parameters

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

Returns

The LPSPI Tx FIFO size.

18.2.8.11 static uint8_t LPSPI_GetRxFifoSize (LPSPI_Type * *base*) [inline], [static]

Parameters

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

Returns

The LPSPI Rx FIFO size.

18.2.8.12 static uint32_t LPSPI_GetTxFifoCount (LPSPI_Type * *base*) [inline], [static]

Parameters

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

Returns

The number of words in the transmit FIFO.

18.2.8.13 static uint32_t LPSPI_GetRxFifoCount (LPSPI_Type * *base*) [inline], [static]

Parameters

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

Returns

The number of words in the receive FIFO.

18.2.8.14 static void LPSPI_ClearStatusFlags (LPSPI_Type * *base*, uint32_t *statusFlags*) [inline], [static]

This function clears the desired status bit by using a write-1-to-clear. The user passes in the base and the desired status flag bit to clear. The list of status flags is defined in the _lpspi_flags. Example usage:

```
* LPSPI_ClearStatusFlags(base, kLPSPI_TxDataRequestFlag|
    kLPSPI_RxDataReadyFlag);
*
```

Parameters

<i>base</i>	LPSPI peripheral address.
<i>statusFlags</i>	The status flag used from type _lpspi_flags.

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

18.2.8.15 static void LPSPI_EnableInterrupts (LPSPI_Type * *base*, uint32_t *mask*) [inline], [static]

This function configures the various interrupt masks of the LPSPI. The parameters are base and an interrupt mask. Note that, for Tx fill and Rx FIFO drain requests, enabling the interrupt request disables the DMA request.

```
* LPSPI_EnableInterrupts(base, kLPSPI_TxInterruptEnable |
    kLPSPI_RxInterruptEnable );
*
```

Parameters

<i>base</i>	LPSPI peripheral address.
<i>mask</i>	The interrupt mask; Use the enum _lpspi_interrupt_enable.

18.2.8.16 static void LPSPI_DisableInterrupts (LPSPI_Type * *base*, uint32_t *mask*) [inline], [static]

```
* LPSPI_DisableInterrupts(base, kLPSPI_TxInterruptEnable |
    kLPSPI_RxInterruptEnable );
*
```

Parameters

<i>base</i>	LPSPI peripheral address.
<i>mask</i>	The interrupt mask; Use the enum _lpspi_interrupt_enable.

18.2.8.17 static void LPSPI_EnableDMA (LPSPI_Type * *base*, uint32_t *mask*) [inline], [static]

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

```
* LPSPI_EnableDMA(base, kLPSPI_TxDmaEnable |
    kLPSPI_RxDmaEnable );
*
```

Parameters

<i>base</i>	LPSPI peripheral address.
<i>mask</i>	The interrupt mask; Use the enum _lpspi_dma_enable.

18.2.8.18 static void LPSPI_DisableDMA (LPSPI_Type * *base*, uint32_t *mask*) [inline], [static]

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

```
* SPI_DisableDMA(base, kLPSPI_TxDmaEnable |
    kLPSPI_RxDmaEnable );
*
```

Parameters

<i>base</i>	LPSPI peripheral address.
<i>mask</i>	The interrupt mask; Use the enum _lpspi_dma_enable.

18.2.8.19 static uint32_t LPSPI_GetTxRegisterAddress (LPSPI_Type * *base*) [inline], [static]

This function gets the LPSPI Transmit Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

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

Returns

The LPSPI Transmit Data Register address.

18.2.8.20 static uint32_t LPSPI_GetRxRegisterAddress (LPSPI_Type * *base*) [inline], [static]

This function gets the LPSPI Receive Data Register address because this value is needed for the DMA operation. This function can be used for either master or slave mode.

Parameters

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

Returns

The LPSPI Receive Data Register address.

18.2.8.21 bool LPSPI_CheckTransferArgument (LPSPI_Type * *base*, lpspi_transfer_t * *transfer*, bool *isEdma*)

Parameters

<i>base</i>	LPSPI peripheral address.
<i>transfer</i>	the transfer struct to be used.
<i>isEdma</i>	True to check for EDMA transfer, false to check interrupt non-blocking transfer

Returns

Return true for right and false for wrong.

18.2.8.22 static void LPSPI_SetMasterSlaveMode (LPSPI_Type * *base*, lpspi_master_slave_mode_t *mode*) [inline], [static]

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

Parameters

<i>base</i>	LPSPI peripheral address.
<i>mode</i>	Mode setting (master or slave) of type lpspi_master_slave_mode_t.

18.2.8.23 static void LPSPI_SelectTransferPCS (LPSPI_Type * *base*, lpspi_which_pcs_t *select*) [inline], [static]

Parameters

<i>base</i>	LPSPI peripheral address.
<i>select</i>	LPSPI Peripheral Chip Select (PCS) configuration.

18.2.8.24 static void LPSPI_SetPCSContinuous (LPSPI_Type * *base*, bool *IsContinuous*) [inline], [static]

Note

In master mode, continuous transfer will keep the PCS asserted at the end of the frame size, until a command word is received that starts a new frame. So PCS must be set back to uncontinuous when transfer finishes. In slave mode, when continuous transfer is enabled, the LPSPI will only transmit the first frame size bits, after that the LPSPI will transmit received data back (assuming a 32-bit shift register).

Parameters

<i>base</i>	LPSPI peripheral address.
<i>IsContinous</i>	True to set the transfer PCS to continuous mode, false to set to uncontinuous mode.

18.2.8.25 static bool LPSPI_IsMaster (LPSPI_Type * *base*) [inline], [static]

Parameters

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

Returns

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

18.2.8.26 static void LPSPI_FlushFifo (LPSPI_Type * *base*, bool *flushTxFifo*, bool *flushRxFifo*) [inline], [static]

Parameters

<i>base</i>	LPSPI peripheral address.
<i>flushTxFifo</i>	Flushes (true) the Tx FIFO, else do not flush (false) the Tx FIFO.
<i>flushRxFifo</i>	Flushes (true) the Rx FIFO, else do not flush (false) the Rx FIFO.

18.2.8.27 static void LPSPI_SetFifoWatermarks (LPSPI_Type * *base*, uint32_t *txWater*, uint32_t *rxWater*) [inline], [static]

This function allows the user to set the receive and transmit FIFO watermarks. The function does not compare the watermark settings to the FIFO size. The FIFO watermark should not be equal to or greater than the FIFO size. It is up to the higher level driver to make this check.

Parameters

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

<i>txWater</i>	The TX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.
<i>rxWater</i>	The RX FIFO watermark value. Writing a value equal or greater than the FIFO size is truncated.

18.2.8.28 static void LPSPI_SetAllPcsPolarity (LPSPI_Type * *base*, uint32_t *mask*) [inline], [static]

Note that the CFGR1 should only be written when the LPSPI is disabled (LPSPIx_CR_MEN = 0).

This is an example: PCS0 and PCS1 set to active low and other PCSs set to active high. Note that the number of PCS is device-specific.

```
* LPSPI_SetAllPcsPolarity(base, kLPSPI_Pcs0ActiveLow |
    kLPSPI_Pcs1ActiveLow);
*
```

Parameters

<i>base</i>	LPSPI peripheral address.
<i>mask</i>	The PCS polarity mask; Use the enum _lpspi_pcs_polarity.

18.2.8.29 static void LPSPI_SetFrameSize (LPSPI_Type * *base*, uint32_t *frameSize*) [inline], [static]

The minimum frame size is 8-bits and the maximum frame size is 4096-bits. If the frame size is less than or equal to 32-bits, the word size and frame size are identical. If the frame size is greater than 32-bits, the word size is 32-bits for each word except the last (the last word contains the remainder bits if the frame size is not divisible by 32). The minimum word size is 2-bits. A frame size of 33-bits (or similar) is not supported.

Note 1: The transmit command register should be initialized before enabling the LPSPI in slave mode, although the command register does not update until after the LPSPI is enabled. After it is enabled, the transmit command register should only be changed if the LPSPI is idle.

Note 2: The transmit and command FIFO is a combined FIFO that includes both transmit data and command words. That means the TCR register should be written to when the Tx FIFO is not full.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>frameSize</i>	The frame size in number of bits.

18.2.8.30 `uint32_t LPSPI_MasterSetBaudRate (LPSPI_Type * base, uint32_t baudRate_Bps, uint32_t srcClock_Hz, uint32_t * tcrPrescaleValue)`

This function takes in the desired bitsPerSec (baud rate) and calculates the nearest possible baud rate without exceeding the desired baud rate and returns the calculated baud rate in bits-per-second. It requires the caller to provide the frequency of the module source clock (in Hertz). Note that the baud rate does not go into effect until the Transmit Control Register (TCR) is programmed with the prescale value. Hence, this function returns the prescale tcrPrescaleValue parameter for later programming in the TCR. The higher level peripheral driver should alert the user of an out of range baud rate input.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>baudRate_Bps</i>	The desired baud rate in bits per second.
<i>srcClock_Hz</i>	Module source input clock in Hertz.
<i>tcrPrescale-Value</i>	The TCR prescale value needed to program the TCR.

Returns

The actual calculated baud rate. This function may also return a "0" if the LPSPI is not configured for master mode or if the LPSPI module is not disabled.

18.2.8.31 `void LPSPI_MasterSetDelayScaler (LPSPI_Type * base, uint32_t scaler, lpspi_delay_type_t whichDelay)`

This function configures the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type lpspi_delay_type_t.

The user passes the desired delay along with the delay value. This allows the user to directly set the delay values if they have pre-calculated them or if they simply wish to manually increment the value.

Note that the LPSPI module must first be disabled before configuring this. Note that the LPSPI module must be configured for master mode before configuring this.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>scaler</i>	The 8-bit delay value 0x00 to 0xFF (255).
<i>whichDelay</i>	The desired delay to configure, must be of type lpspi_delay_type_t.

18.2.8.32 `uint32_t LPSPI_MasterSetDelayTimes (LPSPI_Type * base, uint32_t delayTimeInNanoSec, lpspi_delay_type_t whichDelay, uint32_t srcClock_Hz)`

This function calculates the values for the following: SCK to PCS delay, or PCS to SCK delay, or The configurations must occur between the transfer delay.

The delay names are available in type `lpspi_delay_type_t`.

The user passes the desired delay and the desired delay value in nano-seconds. The function calculates the value needed for the desired delay parameter and returns the actual calculated delay because an exact delay match may not be possible. In this case, the closest match is calculated without going below the desired delay value input. It is possible to input a very large delay value that exceeds the capability of the part, in which case the maximum supported delay is returned. It is up to the higher level peripheral driver to alert the user of an out of range delay input.

Note that the LPSPI module must be configured for master mode before configuring this. And note that the `delayTime = LPSPI_clockSource / (PRESCALE * Delay_scaler)`.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>delayTimeIn-NanoSec</i>	The desired delay value in nano-seconds.
<i>whichDelay</i>	The desired delay to configuration, which must be of type <code>lpspi_delay_type_t</code> .
<i>srcClock_Hz</i>	Module source input clock in Hertz.

Returns

actual Calculated delay value in nano-seconds.

18.2.8.33 `static void LPSPI_WriteData (LPSPI_Type * base, uint32_t data) [inline], [static]`

This function writes data passed in by the user to the Transmit Data Register (TDR). The user can pass up to 32-bits of data to load into the TDR. If the frame size exceeds 32-bits, the user has to manage sending the data one 32-bit word at a time. Any writes to the TDR result in an immediate push to the transmit FIFO. This function can be used for either master or slave modes.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>data</i>	The data word to be sent.

18.2.8.34 static uint32_t LPSPI_ReadData (LPSPI_Type * *base*) [inline], [static]

This function reads the data from the Receive Data Register (RDR). This function can be used for either master or slave mode.

Parameters

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

Returns

The data read from the data buffer.

18.2.8.35 void LPSPI_SetDummyData (LPSPI_Type * *base*, uint8_t *dummyData*)

Parameters

<i>base</i>	LPSPI peripheral address.
<i>dummyData</i>	Data to be transferred when tx buffer is NULL. Note: This API has no effect when LPSPI in slave interrupt mode, because driver will set the TXMSK bit to 1 if txData is NULL, no data is loaded from transmit FIFO and output pin is tristated.

18.2.8.36 void LPSPI_MasterTransferCreateHandle (LPSPI_Type * *base*, lpspi_master_handle_t * *handle*, lpspi_master_transfer_callback_t *callback*, void * *userData*)

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

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	LPSPI handle pointer to lpspi_master_handle_t.
<i>callback</i>	DSPI callback.
<i>userData</i>	callback function parameter.

18.2.8.37 status_t LPSPI_MasterTransferBlocking (LPSPI_Type * *base*, lpspi_transfer_t * *transfer*)

This function transfers data using a polling method. This is a blocking function, which does not return until all transfers have been completed.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>transfer</i>	pointer to lpspi_transfer_t structure.

Returns

status of status_t.

18.2.8.38 status_t LPSPI_MasterTransferNonBlocking (LPSPI_Type * *base*, lpspi_master_handle_t * *handle*, lpspi_transfer_t * *transfer*)

This function transfers data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not integer multiples of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	pointer to lpspi_master_handle_t structure which stores the transfer state.
<i>transfer</i>	pointer to lpspi_transfer_t structure.

Returns

status of status_t.

18.2.8.39 status_t LPSPI_MasterTransferGetCount (LPSPI_Type * *base*, lpspi_master_handle_t * *handle*, size_t * *count*)

This function gets the master transfer remaining bytes.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	pointer to <code>lpspi_master_handle_t</code> structure which stores the transfer state.
<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

18.2.8.40 void LPSPI_MasterTransferAbort (`LPSPI_Type * base`, `lpspi_master_handle_t * handle`)

This function aborts a transfer which uses an interrupt method.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	pointer to <code>lpspi_master_handle_t</code> structure which stores the transfer state.

18.2.8.41 void LPSPI_MasterTransferHandleIRQ (`LPSPI_Type * base`, `lpspi_master_handle_t * handle`)

This function processes the LPSPI transmit and receive IRQ.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	pointer to <code>lpspi_master_handle_t</code> structure which stores the transfer state.

18.2.8.42 void LPSPI_SlaveTransferCreateHandle (`LPSPI_Type * base`, `lpspi_slave_handle_t * handle`, `lpspi_slave_transfer_callback_t callback`, `void * userData`)

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

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	LPSPI handle pointer to <code>lpspi_slave_handle_t</code> .
<i>callback</i>	DSPI callback.
<i>userData</i>	callback function parameter.

18.2.8.43 `status_t LPSPI_SlaveTransferNonBlocking (LPSPI_Type * base, lpspi_slave_handle_t * handle, lpspi_transfer_t * transfer)`

This function transfer data using an interrupt method. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be integer multiples of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	pointer to <code>lpspi_slave_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	pointer to <code>lpspi_transfer_t</code> structure.

Returns

status of `status_t`.

18.2.8.44 `status_t LPSPI_SlaveTransferGetCount (LPSPI_Type * base, lpspi_slave_handle_t * handle, size_t * count)`

This function gets the slave transfer remaining bytes.

Parameters

<i>base</i>	LPSPI peripheral address.
<i>handle</i>	pointer to <code>lpspi_slave_handle_t</code> structure which stores the transfer state.
<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

18.2.8.45 void LPSPI_SlaveTransferAbort (LPSPI_Type * *base*, Ispspi_slave_handle_t * *handle*)

This function aborts a transfer which uses an interrupt method.

Parameters

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

18.2.8.46 void LPSPI_SlaveTransferHandleIRQ (LPSPI_Type * *base*, lpspi_slave_handle_t * *handle*)

This function processes the LPSPI transmit and receives an IRQ.

Parameters

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

18.2.9 Variable Documentation

18.2.9.1 volatile uint8_t g_lpspiDummyData[]

18.3 LPSPI eDMA Driver

18.3.1 Overview

Data Structures

- struct `lpspi_master_edma_handle_t`
LPSPI master eDMA transfer handle structure used for transactional API. [More...](#)
- struct `lpspi_slave_edma_handle_t`
LPSPI slave eDMA transfer handle structure used for transactional API. [More...](#)

TypeDefs

- typedef void(* `lpspi_master_edma_transfer_callback_t`)(LPSPI_Type *base, lpspi_master_edma_handle_t *handle, status_t status, void *userData)
Completion callback function pointer type.
- typedef void(* `lpspi_slave_edma_transfer_callback_t`)(LPSPI_Type *base, lpspi_slave_edma_handle_t *handle, status_t status, void *userData)
Completion callback function pointer type.

Functions

- void `LPSPI_MasterTransferCreateHandleEDMA` (LPSPI_Type *base, lpspi_master_edma_handle_t *handle, `lpspi_master_edma_transfer_callback_t` callback, void *userData, `edma_handle_t` *edmaRxRegToRxDataHandle, `edma_handle_t` *edmaTxDataToTxRegHandle)
Initializes the LPSPI master eDMA handle.
- status_t `LPSPI_MasterTransferEDMA` (LPSPI_Type *base, lpspi_master_edma_handle_t *handle, `lpspi_transfer_t` *transfer)
LPSPI master transfer data using eDMA.
- void `LPSPI_MasterTransferAbortEDMA` (LPSPI_Type *base, lpspi_master_edma_handle_t *handle)
LPSPI master aborts a transfer which is using eDMA.
- status_t `LPSPI_MasterTransferGetCountEDMA` (LPSPI_Type *base, lpspi_master_edma_handle_t *handle, size_t *count)
Gets the master eDMA transfer remaining bytes.
- void `LPSPI_SlaveTransferCreateHandleEDMA` (LPSPI_Type *base, lpspi_slave_edma_handle_t *handle, `lpspi_slave_edma_transfer_callback_t` callback, void *userData, `edma_handle_t` *edmaRxRegToRxDataHandle, `edma_handle_t` *edmaTxDataToTxRegHandle)
Initializes the LPSPI slave eDMA handle.
- status_t `LPSPI_SlaveTransferEDMA` (LPSPI_Type *base, lpspi_slave_edma_handle_t *handle, `lpspi_transfer_t` *transfer)
LPSPI slave transfers data using eDMA.
- void `LPSPI_SlaveTransferAbortEDMA` (LPSPI_Type *base, lpspi_slave_edma_handle_t *handle)
LPSPI slave aborts a transfer which is using eDMA.
- status_t `LPSPI_SlaveTransferGetCountEDMA` (LPSPI_Type *base, lpspi_slave_edma_handle_t *handle, size_t *count)

Gets the slave eDMA transfer remaining bytes.

Driver version

- #define **FSL_LPSPI_EDMA_DRIVER_VERSION** (MAKE_VERSION(2, 1, 0))
LPSPI EDMA driver version.

18.3.2 Data Structure Documentation

18.3.2.1 struct _lpspi_master_edma_handle

Forward declaration of the **_lpspi_master_edma_handle** typedefs.

Data Fields

- volatile bool **isPcsContinuous**
Is PCS continuous in transfer.
- volatile bool **isByteSwap**
A flag that whether should byte swap.
- volatile uint8_t **fifoSize**
FIFO dataSize.
- volatile uint8_t **rxWatermark**
Rx watermark.
- volatile uint8_t **bytesEachWrite**
Bytes for each write TDR.
- volatile uint8_t **bytesEachRead**
Bytes for each read RDR.
- volatile uint8_t **bytesLastRead**
Bytes for last read RDR.
- volatile bool **isThereExtraRxBytes**
Is there extra RX byte.
- uint8_t *volatile **txData**
Send buffer.
- uint8_t *volatile **rxData**
Receive buffer.
- volatile size_t **txRemainingByteCount**
Number of bytes remaining to send.
- volatile size_t **rxRemainingByteCount**
Number of bytes remaining to receive.
- volatile uint32_t **writeRegRemainingTimes**
Write TDR register remaining times.
- volatile uint32_t **readRegRemainingTimes**
Read RDR register remaining times.
- uint32_t **totalByteCount**
Number of transfer bytes.
- uint32_t **txBuffIfNull**
Used if there is not txData for DMA purpose.

- `uint32_t rxBuffIfNull`
Used if there is not rxData for DMA purpose.
- `uint32_t transmitCommand`
Used to write TCR for DMA purpose.
- `volatile uint8_t state`
LPSPI transfer state , _lpspi_transfer_state.
- `uint8_t nbytes`
eDMA minor byte transfer count initially configured.
- `lpspi_master_edma_transfer_callback_t callback`
Completion callback.
- `void *userData`
Callback user data.
- `edma_handle_t *edmaRxRegToRxDataHandle`
edma_handle_t handle point used for RxReg to RxData buff
- `edma_handle_t *edmaTxDataToTxRegHandle`
edma_handle_t handle point used for TxData to TxReg buff
- `edma_tcd_t lpspiSoftwareTCD [3]`
SoftwareTCD, internal used.

Field Documentation

- (1) `volatile bool lpspi_master_edma_handle_t::isPcsContinuous`
- (2) `volatile bool lpspi_master_edma_handle_t::isByteSwap`
- (3) `volatile uint8_t lpspi_master_edma_handle_t::fifoSize`
- (4) `volatile uint8_t lpspi_master_edma_handle_t::rxWatermark`
- (5) `volatile uint8_t lpspi_master_edma_handle_t::bytesEachWrite`
- (6) `volatile uint8_t lpspi_master_edma_handle_t::bytesEachRead`
- (7) `volatile uint8_t lpspi_master_edma_handle_t::bytesLastRead`
- (8) `volatile bool lpspi_master_edma_handle_t::isThereExtraRxBytes`
- (9) `uint8_t* volatile lpspi_master_edma_handle_t::txData`
- (10) `uint8_t* volatile lpspi_master_edma_handle_t::rxData`
- (11) `volatile size_t lpspi_master_edma_handle_t::txRemainingByteCount`
- (12) `volatile size_t lpspi_master_edma_handle_t::rxRemainingByteCount`
- (13) `volatile uint32_t lpspi_master_edma_handle_t::writeRegRemainingTimes`
- (14) `volatile uint32_t lpspi_master_edma_handle_t::readRegRemainingTimes`
- (15) `uint32_t lpspi_master_edma_handle_t::txBuffIfNull`

- (16) `uint32_t lpspi_master_edma_handle_t::rxBuffIfNull`
- (17) `uint32_t lpspi_master_edma_handle_t::transmitCommand`
- (18) `volatile uint8_t lpspi_master_edma_handle_t::state`
- (19) `uint8_t lpspi_master_edma_handle_t::nbytes`
- (20) `lpspi_master_edma_transfer_callback_t lpspi_master_edma_handle_t::callback`
- (21) `void* lpspi_master_edma_handle_t::userData`

18.3.2.2 struct _lpspi_slave_edma_handle

Forward declaration of the `_lpspi_slave_edma_handle` typedefs.

Data Fields

- `volatile bool isByteSwap`
A flag that whether should byte swap.
- `volatile uint8_t fifoSize`
FIFO dataSize.
- `volatile uint8_t rxWatermark`
Rx watermark.
- `volatile uint8_t bytesEachWrite`
Bytes for each write TDR.
- `volatile uint8_t bytesEachRead`
Bytes for each read RDR.
- `volatile uint8_t bytesLastRead`
Bytes for last read RDR.
- `volatile bool isThereExtraRxBytes`
Is there extra RX byte.
- `uint8_t nbytes`
eDMA minor byte transfer count initially configured.
- `uint8_t *volatile txData`
Send buffer.
- `uint8_t *volatile rxData`
Receive buffer.
- `volatile size_t txRemainingByteCount`
Number of bytes remaining to send.
- `volatile size_t rxRemainingByteCount`
Number of bytes remaining to receive.
- `volatile uint32_t writeRegRemainingTimes`
Write TDR register remaining times.
- `volatile uint32_t readRegRemainingTimes`
Read RDR register remaining times.
- `uint32_t totalByteCount`
Number of transfer bytes.
- `uint32_t txBuffIfNull`
Used if there is not txData for DMA purpose.

- `uint32_t rxBuffIfNull`
Used if there is not rxData for DMA purpose.
- `volatile uint8_t state`
LPSPI transfer state.
- `uint32_t errorCount`
Error count for slave transfer.
- `lpspi_slave_edma_transfer_callback_t callback`
Completion callback.
- `void *userData`
Callback user data.
- `edma_handle_t *edmaRxRegToRxDataHandle`
edma_handle_t handle point used for RxReg to RxData buff
- `edma_handle_t *edmaTxDataToTxRegHandle`
edma_handle_t handle point used for TxData to TxReg
- `edma_tcd_t lpspiSoftwareTCD [2]`
SoftwareTCD, internal used.

Field Documentation

- (1) `volatile bool lpspi_slave_edma_handle_t::isByteSwap`
- (2) `volatile uint8_t lpspi_slave_edma_handle_t::fifoSize`
- (3) `volatile uint8_t lpspi_slave_edma_handle_t::rxWatermark`
- (4) `volatile uint8_t lpspi_slave_edma_handle_t::bytesEachWrite`
- (5) `volatile uint8_t lpspi_slave_edma_handle_t::bytesEachRead`
- (6) `volatile uint8_t lpspi_slave_edma_handle_t::bytesLastRead`
- (7) `volatile bool lpspi_slave_edma_handle_t::isThereExtraRxBytes`
- (8) `uint8_t lpspi_slave_edma_handle_t::nbytes`
- (9) `uint8_t* volatile lpspi_slave_edma_handle_t::txData`
- (10) `uint8_t* volatile lpspi_slave_edma_handle_t::rxData`
- (11) `volatile size_t lpspi_slave_edma_handle_t::txRemainingByteCount`
- (12) `volatile size_t lpspi_slave_edma_handle_t::rxRemainingByteCount`
- (13) `volatile uint32_t lpspi_slave_edma_handle_t::writeRegRemainingTimes`
- (14) `volatile uint32_t lpspi_slave_edma_handle_t::readRegRemainingTimes`
- (15) `uint32_t lpspi_slave_edma_handle_t::txBuffIfNull`
- (16) `uint32_t lpspi_slave_edma_handle_t::rxBuffIfNull`

- (17) `volatile uint8_t lpspi_slave_edma_handle_t::state`
- (18) `uint32_t lpspi_slave_edma_handle_t::errorCount`
- (19) `lpspi_slave_edma_transfer_callback_t lpspi_slave_edma_handle_t::callback`
- (20) `void* lpspi_slave_edma_handle_t::userData`

18.3.3 Macro Definition Documentation

18.3.3.1 `#define FSL_LPSPI_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 0))`

18.3.4 Typedef Documentation

18.3.4.1 `typedef void(* lpspi_master_edma_transfer_callback_t)(LPSPI_Type *base, lpspi_master_edma_handle_t *handle, status_t status, void *userData)`

Parameters

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

18.3.4.2 `typedef void(* lpspi_slave_edma_transfer_callback_t)(LPSPI_Type *base, lpspi_slave_edma_handle_t *handle, status_t status, void *userData)`

Parameters

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

18.3.5 Function Documentation

```
18.3.5.1 void LPSPI_MasterTransferCreateHandleEDMA ( LPSPI_Type * base,
    lpspi_master_edma_handle_t * handle, lpspi_master_edma_transfer_callback_t
    callback, void * userData, edma_handle_t * edmaRxRegToRxDataHandle,
    edma_handle_t * edmaTxDataToTxRegHandle )
```

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

Note that the LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx are the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for edmaRxRegToRxDataHandle and Tx DMAMUX source for edmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Tx DMAMUX source for edmaRxRegToRxDataHandle.

Parameters

<i>base</i>	LPSPI peripheral base address.
<i>handle</i>	LPSPI handle pointer to lpspi_master_edma_handle_t.
<i>callback</i>	LPSPI callback.
<i>userData</i>	callback function parameter.
<i>edmaRxRegToRxDataHandle</i>	edmaRxRegToRxDataHandle pointer to edma_handle_t .
<i>edmaTxDataToTxRegHandle</i>	edmaTxDataToTxRegHandle pointer to edma_handle_t .

```
18.3.5.2 status_t LPSPI_MasterTransferEDMA ( LPSPI_Type * base,
    lpspi_master_edma_handle_t * handle, lpspi_transfer_t * transfer )
```

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

<i>base</i>	LPSPI peripheral base address.
<i>handle</i>	pointer to <code>lpspi_master_edma_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	pointer to <code>lpspi_transfer_t</code> structure.

Returns

status of `status_t`.

18.3.5.3 `void LPSPI_MasterTransferAbortEDMA (LPSPI_Type * base, lpspi_master_edma_handle_t * handle)`

This function aborts a transfer which is using eDMA.

Parameters

<i>base</i>	LPSPI peripheral base address.
<i>handle</i>	pointer to <code>lpspi_master_edma_handle_t</code> structure which stores the transfer state.

18.3.5.4 `status_t LPSPI_MasterTransferGetCountEDMA (LPSPI_Type * base, lpspi_master_edma_handle_t * handle, size_t * count)`

This function gets the master eDMA transfer remaining bytes.

Parameters

<i>base</i>	LPSPI peripheral base address.
<i>handle</i>	pointer to <code>lpspi_master_edma_handle_t</code> structure which stores the transfer state.
<i>count</i>	Number of bytes transferred so far by the EDMA transaction.

Returns

status of `status_t`.

18.3.5.5 `void LPSPI_SlaveTransferCreateHandleEDMA (LPSPI_Type * base, lpspi_slave_edma_handle_t * handle, lpspi_slave_edma_transfer_callback_t callback, void * userData, edma_handle_t * edmaRxRegToRxDataHandle, edma_handle_t * edmaTxDataToTxRegHandle)`

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

Note that LPSPI eDMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx as the same source) DMA request source.

(1) For a separated DMA request source, enable and set the Rx DMAMUX source for edmaRxRegToRxDataHandle and Tx DMAMUX source for edmaTxDataToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for edmaRxRegToRxDataHandle .

Parameters

<i>base</i>	LPSPI peripheral base address.
<i>handle</i>	LPSPI handle pointer to lpspi_slave_edma_handle_t.
<i>callback</i>	LPSPI callback.
<i>userData</i>	callback function parameter.
<i>edmaRxRegToRxDataHandle</i>	edmaRxRegToRxDataHandle pointer to edma_handle_t .
<i>edmaTxDataToTxRegHandle</i>	edmaTxDataToTxRegHandle pointer to edma_handle_t .

18.3.5.6 status_t LPSPI_SlaveTransferEDMA (LPSPI_Type * *base*, lpspi_slave_edma_handle_t * *handle*, lpspi_transfer_t * *transfer*)

This function transfers data using eDMA. This is a non-blocking function, which return right away. When all data is transferred, the callback function is called.

Note: The transfer data size should be an integer multiple of bytesPerFrame if bytesPerFrame is less than or equal to 4. For bytesPerFrame greater than 4: The transfer data size should be equal to bytesPerFrame if the bytesPerFrame is not an integer multiple of 4. Otherwise, the transfer data size can be an integer multiple of bytesPerFrame.

Parameters

<i>base</i>	LPSPI peripheral base address.
<i>handle</i>	pointer to lpspi_slave_edma_handle_t structure which stores the transfer state.
<i>transfer</i>	pointer to lpspi_transfer_t structure.

Returns

status of status_t.

18.3.5.7 void LPSPI_SlaveTransferAbortEDMA (LPSPI_Type * *base*, lpspi_slave_edma_handle_t * *handle*)

This function aborts a transfer which is using eDMA.

Parameters

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

18.3.5.8 status_t LPSPI_SlaveTransferGetCountEDMA (LPSPI_Type * *base*, lpspi_slave_edma_handle_t * *handle*, size_t * *count*)

This function gets the slave eDMA transfer remaining bytes.

Parameters

<i>base</i>	LPSPI peripheral base address.
<i>handle</i>	pointer to lpspi_slave_edma_handle_t structure which stores the transfer state.
<i>count</i>	Number of bytes transferred so far by the eDMA transaction.

Returns

status of status_t.

18.4 LPSPI FreeRTOS Driver

18.4.1 Overview

Driver version

- #define `FSL_LPSPI_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 5))`
LPSPI FreeRTOS driver version 2.0.5.

LPSPI RTOS Operation

- `status_t LPSPI_RTOS_Init (lpspi_rtos_handle_t *handle, LPSPI_Type *base, const lpspi_master_config_t *masterConfig, uint32_t srcClock_Hz)`
Initializes LPSPI.
- `status_t LPSPI_RTOS_Deinit (lpspi_rtos_handle_t *handle)`
Deinitializes the LPSPI.
- `status_t LPSPI_RTOS_Transfer (lpspi_rtos_handle_t *handle, lpspi_transfer_t *transfer)`
Performs SPI transfer.

18.4.2 Macro Definition Documentation

18.4.2.1 #define FSL_LPSPI_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 0, 5))

18.4.3 Function Documentation

18.4.3.1 `status_t LPSPI_RTOS_Init (lpspi_rtos_handle_t * handle, LPSPI_Type * base, const lpspi_master_config_t * masterConfig, uint32_t srcClock_Hz)`

This function initializes the LPSPI module and related RTOS context.

Parameters

<code>handle</code>	The RTOS LPSPI handle, the pointer to an allocated space for RTOS context.
<code>base</code>	The pointer base address of the LPSPI instance to initialize.
<code>masterConfig</code>	Configuration structure to set-up LPSPI in master mode.
<code>srcClock_Hz</code>	Frequency of input clock of the LPSPI module.

Returns

status of the operation.

18.4.3.2 status_t LPSPI_RTOS_Deinit (*Ipspi_rtos_handle_t * handle*)

This function deinitializes the LPSPI module and related RTOS context.

Parameters

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

18.4.3.3 status_t LPSPI_RTOS_Transfer (lpspi_rtos_handle_t * *handle*, lpspi_transfer_t * *transfer*)

This function performs an SPI transfer according to data given in the transfer structure.

Parameters

<i>handle</i>	The RTOS LPSPI handle.
<i>transfer</i>	Structure specifying the transfer parameters.

Returns

status of the operation.

18.5 LPSPI CMSIS Driver

This section describes the programming interface of the LPSPI Cortex Microcontroller Software Interface Standard (CMSIS) driver. And this driver defines generic peripheral driver interfaces for middleware making it reusable across a wide range of supported microcontroller devices. The API connects microcontroller peripherals with middleware that implements for example communication stacks, file systems, or graphic user interfaces. More information and usage method please refer to <http://www.-keil.com/pack/doc/cmsis/Driver/html/index.html>.

18.5.1 Function groups

18.5.1.1 LPSPI CMSIS GetVersion Operation

This function group will return the DSPI CMSIS Driver version to user.

18.5.1.2 LPSPI CMSIS GetCapabilities Operation

This function group will return the capabilities of this driver.

18.5.1.3 LPSPI CMSIS Initialize and Uninitialize Operation

This function will initialize and uninitialized the instance in master mode or slave mode. And this API must be called before you configure an instance or after you Deinit an instance. The right steps to start an instance is that you must initialize the instance which been selected firstly, then you can power on the instance. After these all have been done, you can configure the instance by using control operation. If you want to Uninitialize the instance, you must power off the instance first.

18.5.1.4 LPSPI Transfer Operation

This function group controls the transfer, master send/receive data, and slave send/receive data.

18.5.1.5 LPSPI Status Operation

This function group gets the LPSPI transfer status.

18.5.1.6 LPSPI CMSIS Control Operation

This function can select instance as master mode or slave mode, set baudrate for master mode transfer, get current baudrate of master mode transfer, set transfer data bits and set other control command.

18.5.2 Typical use case

18.5.2.1 Master Operation

```
/* Variables */
uint8_t masterRxData[TRANSFER_SIZE] = {0U};
uint8_t masterTxData[TRANSFER_SIZE] = {0U};

/*DSPI master init*/
Driver_SPI0.Initialize(DSPI_MasterSignalEvent_t);
Driver_SPI0.PowerControl(ARM_POWER_FULL);
Driver_SPI0.Control(ARM_SPI_MODE_MASTER, TRANSFER_BAUDRATE);

/* Start master transfer */
Driver_SPI0.Transfer(masterTxData, masterRxData, TRANSFER_SIZE);

/* Master power off */
Driver_SPI0.PowerControl(ARM_POWER_OFF);

/* Master uninitialize */
Driver_SPI0.Uninitialize();
```

18.5.2.2 Slave Operation

```
/* Variables */
uint8_t slaveRxData[TRANSFER_SIZE] = {0U};
uint8_t slaveTxData[TRANSFER_SIZE] = {0U};

/*DSPI slave init*/
Driver_SPI2.Initialize(DSPI_SlaveSignalEvent_t);
Driver_SPI2.PowerControl(ARM_POWER_FULL);
Driver_SPI2.Control(ARM_SPI_MODE_SLAVE, false);

/* Start slave transfer */
Driver_SPI2.Transfer(slaveTxData, slaveRxData, TRANSFER_SIZE);

/* slave power off */
Driver_SPI2.PowerControl(ARM_POWER_OFF);

/* slave uninitialize */
Driver_SPI2.Uninitialize();
```

Chapter 19

LPTMR: Low-Power Timer

19.1 Overview

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

19.2 Function groups

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

19.2.1 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.

19.2.2 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.

19.2.3 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.

19.2.4 Status

Provides functions to get and clear the LPTMR status.

19.2.5 Interrupt

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

19.3 Typical use case

19.3.1 LPTMR tick example

Updates the LPTMR period and toggles an LED periodically. Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/lptmr

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.*

Functions

- static void **LPTMR_EnableTimerDMA** (LPTMR_Type *base, bool enable)
Enable or disable timer DMA request.

Driver version

- #define **FSL_LPTMR_DRIVER_VERSION** (MAKE_VERSION(2, 1, 1))
Version 2.1.1.

Initialization and deinitialization

- void **LPTMR_Init** (LPTMR_Type *base, const lptmr_config_t *config)
Ungates the LPTMR clock and configures the peripheral for a basic operation.
- void **LPTMR_Deinit** (LPTMR_Type *base)
Gates the LPTMR clock.
- void **LPTMR_GetDefaultConfig** (lptmr_config_t *config)
Fills in the LPTMR configuration structure with default settings.

Interrupt Interface

- static void [LPTMR_EnableInterrupts](#) (LPTMR_Type *base, uint32_t mask)
Enables the selected LPTMR interrupts.
- static void [LPTMR_DisableInterrupts](#) (LPTMR_Type *base, uint32_t mask)
Disables the selected LPTMR interrupts.
- static uint32_t [LPTMR_GetEnabledInterrupts](#) (LPTMR_Type *base)
Gets the enabled LPTMR interrupts.

Status Interface

- static uint32_t [LPTMR_GetStatusFlags](#) (LPTMR_Type *base)
Gets the LPTMR status flags.
- static void [LPTMR_ClearStatusFlags](#) (LPTMR_Type *base, uint32_t mask)
Clears the LPTMR status flags.

Read and write the timer period

- static void [LPTMR_SetTimerPeriod](#) (LPTMR_Type *base, uint32_t ticks)
Sets the timer period in units of count.
- static uint32_t [LPTMR_GetCurrentTimerCount](#) (LPTMR_Type *base)
Reads the current timer counting value.

Timer Start and Stop

- static void [LPTMR_StartTimer](#) (LPTMR_Type *base)
Starts the timer.
- static void [LPTMR_StopTimer](#) (LPTMR_Type *base)
Stops the timer.

19.4 Data Structure Documentation

19.4.1 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.

19.5 Enumeration Type Documentation

19.5.1 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.

19.5.2 enum lptmr_pin_polarity_t

Enumerator

kLPTMR_PinPolarityActiveHigh Pulse Counter input source is active-high.

kLPTMR_PinPolarityActiveLow Pulse Counter input source is active-low.

19.5.3 enum lptmr_timer_mode_t

Enumerator

kLPTMR_TimerModeTimeCounter Time Counter mode.

kLPTMR_TimerModePulseCounter Pulse Counter mode.

19.5.4 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.

19.5.5 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.

19.5.6 enum lptmr_interrupt_enable_t

Enumerator

kLPTMR_TimerInterruptEnable Timer interrupt enable.

19.5.7 enum lptmr_status_flags_t

Enumerator

kLPTMR_TimerCompareFlag Timer compare flag.

19.6 Function Documentation

19.6.1 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.

19.6.2 void LPTMR_Deinit (LPTMR_Type * *base*)

Parameters

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

19.6.3 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

<i>config</i>	A pointer to the LPTMR configuration structure.
---------------	---

**19.6.4 static void LPTMR_EnableInterrupts (LPTMR_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	LPTMR peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t

19.6.5 static void LPTMR_DisableInterrupts (LPTMR_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	LPTMR peripheral base address
<i>mask</i>	The interrupts to disable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t .

19.6.6 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](#)

19.6.7 static void LPTMR_EnableTimerDMA (LPTMR_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	base LPTMR peripheral base address
<i>enable</i>	Switcher of timer DMA feature. "true" means to enable, "false" means to disable.

19.6.8 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](#)

19.6.9 static void LPTMR_ClearStatusFlags (LPTMR_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	LPTMR peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration lptmr_status_flags_t .

19.6.10 static void LPTMR_SetTimerPeriod (LPTMR_Type * *base*, uint32_t *ticks*) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note

1. The TCF flag is set with the CNR equals the count provided here and then increments.
2. Call the utility macros provided in the `fsl_common.h` to convert to ticks.

Parameters

<i>base</i>	LPTMR peripheral base address
<i>ticks</i>	A timer period in units of ticks, which should be equal or greater than 1.

19.6.11 static uint32_t LPTMR_GetCurrentTimerCount (LPTMR_Type * *base*) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

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

**19.6.12 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
-------------	-------------------------------

**19.6.13 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 20

LPUART: Low Power Universal Asynchronous Receiver-/Transmitter Driver

20.1 Overview

Modules

- [LPUART CMSIS Driver](#)
- [LPUART Driver](#)
- [LPUART FreeRTOS Driver](#)
- [LPUART eDMA Driver](#)

20.2 LPUART Driver

20.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Low Power UART (LPUART) module of MCUXpresso SDK devices.

20.2.2 Typical use case

20.2.2.1 LPUART Operation

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/lpuart

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...](#)

Macros

- #define [UART_RETRY_TIMES](#) 0U /* Defining to zero means to keep waiting for the flag until it is assert/deassert. */
Retry times for waiting flag.

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 {

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,

kStatus_LPUART_IdleLineDetected = MAKE_STATUS(kStatusGroup_LPUART, 14),

kStatus_LPUART_Timeout = MAKE_STATUS(kStatusGroup_LPUART, 15) }

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,

kLPUART_SevenDataBits = 0x1U }
- LPUART data bits count.*
- enum `lpuart_stop_bit_count_t` {

kLPUART_OneStopBit = 0U,

kLPUART_TwoStopBit = 1U }
- LPUART stop bit count.*
- enum `lpuart_transmit_cts_source_t` {

kLPUART_CtsSourcePin = 0U,

kLPUART_CtsSourceMatchResult = 1U }
- LPUART transmit CTS source.*
- enum `lpuart_transmit_cts_config_t` {

kLPUART_CtsSampleAtStart = 0U,

kLPUART_CtsSampleAtIdle = 1U }
- LPUART transmit CTS configure.*
- enum `lpuart_idle_type_select_t` {

kLPUART_IdleTypeStartBit = 0U,

kLPUART_IdleTypeStopBit = 1U }
- LPUART idle flag type defines when the receiver starts counting.*
- enum `lpuart_idle_config_t` {

```
kLPUART_IdleCharacter1 = 0U,
kLPUART_IdleCharacter2 = 1U,
kLPUART_IdleCharacter4 = 2U,
kLPUART_IdleCharacter8 = 3U,
kLPUART_IdleCharacter16 = 4U,
kLPUART_IdleCharacter32 = 5U,
kLPUART_IdleCharacter64 = 6U,
kLPUART_IdleCharacter128 = 7U }
```

LPUART idle detected configuration.

- enum _lpuart_interrupt_enable {

```
kLPUART_LinBreakInterruptEnable = (LPUART_BAUD_LBKDIIE_MASK >> 8U),
kLPUART_RxActiveEdgeInterruptEnable = (LPUART_BAUD_RXEDGIE_MASK >> 8U),
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),
kLPUART_Match1InterruptEnable = (LPUART_CTRL_MA1IE_MASK),
kLPUART_Match2InterruptEnable = (LPUART_CTRL_MA2IE_MASK),
kLPUART_TxFifoOverflowInterruptEnable = (LPUART_FIFO_TXOFE_MASK),
kLPUART_RxFifoUnderflowInterruptEnable = (LPUART_FIFO_RXUFE_MASK) }
```

LPUART interrupt configuration structure, default settings all disabled.

- enum _lpuart_flags {

```
kLPUART_TxDataRegEmptyFlag,
kLPUART_TransmissionCompleteFlag,
kLPUART_RxDataRegFullFlag = (LPUART_STAT_RDRF_MASK),
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 = (LPUART_STAT_RXEDGIF_MASK),
kLPUART_RxActiveFlag,
kLPUART_DataMatch1Flag,
kLPUART_DataMatch2Flag,
kLPUART_TxFifoEmptyFlag,
kLPUART_RxFifoEmptyFlag,
kLPUART_TxFifoOverflowFlag,
kLPUART_RxFifoUnderflowFlag }
```

LPUART status flags.

Driver version

- `#define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 5, 3))`
LPUART driver version.

Software Reset

- static void `LPUART_SoftwareReset` (LPUART_Type *base)
Resets the LPUART using software.

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.

Module configuration

- `status_t LPUART_SetBaudRate` (LPUART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
Sets the LPUART instance baudrate.
- void `LPUART_Enable9bitMode` (LPUART_Type *base, bool enable)
Enable 9-bit data mode for LPUART.
- static void `LPUART_SetMatchAddress` (LPUART_Type *base, uint16_t address1, uint16_t address2)
Set the LPUART address.
- static void `LPUART_EnableMatchAddress` (LPUART_Type *base, bool match1, bool match2)
Enable the LPUART match address feature.
- static void `LPUART_SetRxFifoWatermark` (LPUART_Type *base, uint8_t water)
Sets the rx FIFO watermark.
- static void `LPUART_SetTxFifoWatermark` (LPUART_Type *base, uint8_t water)
Sets the tx FIFO watermark.

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.

DMA Configuration

- 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

- uint32_t [LPUARTGetInstance](#) (LPUART_Type *base)
Get the LPUART instance from peripheral base address.
- 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.
- static uint8_t [LPUART_GetRxFifoCount](#) (LPUART_Type *base)
Gets the rx FIFO data count.
- static uint8_t [LPUART_GetTxFifoCount](#) (LPUART_Type *base)
Gets the tx FIFO data count.
- void [LPUART_SendAddress](#) (LPUART_Type *base, uint8_t address)
Transmit an address frame in 9-bit data mode.
- status_t [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.
- `size_t LPUART_TransferGetRxRingBufferLength (LPUART_Type *base, lpuart_handle_t *handle)`

Get the length of received data in RX 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 sent out to bus.
- `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, void *irqHandle)`

LPUART IRQ handle function.
- `void LPUART_TransferHandleErrorIRQ (LPUART_Type *base, void *irqHandle)`

LPUART Error IRQ handle function.

20.2.3 Data Structure Documentation

20.2.3.1 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.
- `uint8_t txFifoWatermark`

TX FIFO watermark.
- `uint8_t rxFifoWatermark`

- `bool enableRxRTS`
RX RTS enable.
- `bool enableTxCTS`
TX CTS enable.
- `lpuart_transmit_cts_source_t txCtsSource`
TX CTS source.
- `lpuart_transmit_cts_config_t txCtsConfig`
TX CTS configure.
- `lpuart_idle_type_select_t rxIdleType`
RX IDLE type.
- `lpuart_idle_config_t rxIdleConfig`
RX IDLE configuration.
- `bool enableTx`
Enable TX.
- `bool enableRx`
Enable RX.

Field Documentation

(1) `lpuart_idle_type_select_t lpuart_config_t::rxIdleType`

(2) `lpuart_idle_config_t lpuart_config_t::rxIdleConfig`

20.2.3.2 struct lpuart_transfer_t

Data Fields

- `size_t dataSize`
The byte count to be transfer.
- `uint8_t * data`
The buffer of data to be transfer.
- `uint8_t * rxData`
The buffer to receive data.
- `const uint8_t * txData`
The buffer of data to be sent.

Field Documentation

(1) `uint8_t* lpuart_transfer_t::data`

(2) `uint8_t* lpuart_transfer_t::rxData`

(3) `const uint8_t* lpuart_transfer_t::txData`

(4) `size_t lpuart_transfer_t::dataSize`

20.2.3.3 struct _lpuart_handle

Data Fields

- const 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.
- bool **isSevenDataBits**
Seven data bits flag.

Field Documentation

- (1) const uint8_t* volatile lpuart_handle_t::txData
- (2) volatile size_t lpuart_handle_t::txDataSize
- (3) size_t lpuart_handle_t::txDataSizeAll
- (4) uint8_t* volatile lpuart_handle_t::rxData
- (5) volatile size_t lpuart_handle_t::rxDataSize
- (6) size_t lpuart_handle_t::rxDataSizeAll
- (7) uint8_t* lpuart_handle_t::rxRingBuffer

- (8) `size_t lpuart_handle_t::rxRingBufferSize`
- (9) `volatile uint16_t lpuart_handle_t::rxRingBufferHead`
- (10) `volatile uint16_t lpuart_handle_t::rxRingBufferTail`
- (11) `lpuart_transfer_callback_t lpuart_handle_t::callback`
- (12) `void* lpuart_handle_t::userData`
- (13) `volatile uint8_t lpuart_handle_t::txState`
- (14) `volatile uint8_t lpuart_handle_t::rxState`
- (15) `bool lpuart_handle_t::isSevenDataBits`

20.2.4 Macro Definition Documentation

20.2.4.1 #define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 5, 3))

20.2.4.2 #define UART_RETRY_TIMES 0U /* Defining to zero means to keep waiting for the flag until it is assert/deassert. */

20.2.5 Typedef Documentation

20.2.5.1 typedef void(* lpuart_transfer_callback_t)(LPUART_Type *base, lpuart_handle_t *handle, status_t status, void *userData)

20.2.6 Enumeration Type Documentation

20.2.6.1 anonymous enum

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.

kStatus_LPUART_IdleLineDetected IDLE flag.

kStatus_LPUART_Timeout LPUART times out.

20.2.6.2 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.

20.2.6.3 enum lpuart_data_bits_t

Enumerator

kLPUART_EightDataBits Eight data bit.

kLPUART_SevenDataBits Seven data bit.

20.2.6.4 enum lpuart_stop_bit_count_t

Enumerator

kLPUART_OneStopBit One stop bit.

kLPUART_TwoStopBit Two stop bits.

20.2.6.5 enum lpuart_transmit_cts_source_t

Enumerator

kLPUART_CtsSourcePin CTS resource is the LPUART_CTS pin.

kLPUART_CtsSourceMatchResult CTS resource is the match result.

20.2.6.6 enum lpuart_transmit_cts_config_t

Enumerator

kLPUART_CtsSampleAtStart CTS input is sampled at the start of each character.

kLPUART_CtsSampleAtIdle CTS input is sampled when the transmitter is idle.

20.2.6.7 enum lpuart_idle_type_select_t

Enumerator

kLPUART_IdleTypeStartBit Start counting after a valid start bit.

kLPUART_IdleTypeStopBit Start counting after a stop bit.

20.2.6.8 enum lpuart_idle_config_t

This structure defines the number of idle characters that must be received before the IDLE flag is set.

Enumerator

kLPUART_IdleCharacter1 the number of idle characters.

kLPUART_IdleCharacter2 the number of idle characters.

kLPUART_IdleCharacter4 the number of idle characters.

kLPUART_IdleCharacter8 the number of idle characters.

kLPUART_IdleCharacter16 the number of idle characters.

kLPUART_IdleCharacter32 the number of idle characters.

kLPUART_IdleCharacter64 the number of idle characters.

kLPUART_IdleCharacter128 the number of idle characters.

20.2.6.9 enum _lpuart_interrupt_enable

This structure contains the settings for all LPUART interrupt configurations.

Enumerator

kLPUART_LinBreakInterruptEnable LIN break detect. bit 7

kLPUART_RxActiveEdgeInterruptEnable Receive Active Edge. bit 6

kLPUART_TxDataRegEmptyInterruptEnable Transmit data register empty. bit 23

kLPUART_TransmissionCompleteInterruptEnable Transmission complete. bit 22

kLPUART_RxDataRegFullInterruptEnable Receiver data register full. bit 21

kLPUART_IdleLineInterruptEnable Idle line. bit 20

kLPUART_RxOverrunInterruptEnable Receiver Overrun. bit 27

kLPUART_NoiseErrorInterruptEnable Noise error flag. bit 26

kLPUART_FramingErrorInterruptEnable Framing error flag. bit 25

kLPUART_ParityErrorInterruptEnable Parity error flag. bit 24

kLPUART_Match1InterruptEnable Parity error flag. bit 15

kLPUART_Match2InterruptEnable Parity error flag. bit 14

kLPUART_TxFifoOverflowInterruptEnable Transmit FIFO Overflow. bit 9

kLPUART_RxFifoUnderflowInterruptEnable Receive FIFO Underflow. bit 8

20.2.6.10 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. bit 23

kLPUART_TransmissionCompleteFlag Transmission complete flag, sets when transmission activity complete. bit 22

kLPUART_RxDataRegFullFlag Receive data register full flag, sets when the receive data buffer is full. bit 21

kLPUART_IdleLineFlag Idle line detect flag, sets when idle line detected. bit 20

kLPUART_RxOverrunFlag Receive Overrun, sets when new data is received before data is read from receive register. bit 19

kLPUART_NoiseErrorFlag Receive takes 3 samples of each received bit. If any of these samples differ, noise flag sets. bit 18

kLPUART_FramingErrorFlag Frame error flag, sets if logic 0 was detected where stop bit expected. bit 17

kLPUART_ParityErrorFlag If parity enabled, sets upon parity error detection. bit 16

kLPUART_LinBreakFlag LIN break detect interrupt flag, sets when LIN break char detected and LIN circuit enabled. bit 31

kLPUART_RxActiveEdgeFlag Receive pin active edge interrupt flag, sets when active edge detected. bit 30

kLPUART_RxActiveFlag Receiver Active Flag (RAF), sets at beginning of valid start. bit 24

kLPUART_DataMatch1Flag The next character to be read from LPUART_DATA matches MA1. bit 15

kLPUART_DataMatch2Flag The next character to be read from LPUART_DATA matches MA2. bit 14

kLPUART_TxFifoEmptyFlag TXEMPT bit, sets if transmit buffer is empty. bit 7

kLPUART_RxFifoEmptyFlag RXEMPT bit, sets if receive buffer is empty. bit 6

kLPUART_TxFifoOverflowFlag TXOF bit, sets if transmit buffer overflow occurred. bit 1

kLPUART_RxFifoUnderflowFlag RXUF bit, sets if receive buffer underflow occurred. bit 0

20.2.7 Function Documentation

20.2.7.1 static void LPUART_SoftwareReset (LPUART_Type * *base*) [inline], [static]

This function resets all internal logic and registers except the Global Register. Remains set until cleared by software.

Parameters

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

20.2.7.2 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

20.2.7.3 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.
-------------	---------------------------------

20.2.7.4 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;
lpuartConfig->dataBitsCount = kLPUART_EightDataBits; lpuartConfig->isMsb = false; lpuartConfig->stopBitCount = kLPUART_OneStopBit; lpuartConfig->txFifoWatermark = 0; lpuartConfig->rxFifoWatermark = 1; lpuartConfig->rxIdleType = kLPUART_IdleTypeStartBit; lpuartConfig->rxIdleConfig = kLPUART_IdleCharacter1; lpuartConfig->enableTx = false; lpuartConfig->enableRx = false;

Parameters

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

20.2.7.5 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.

20.2.7.6 void LPUART_Enable9bitMode (LPUART_Type * *base*, bool *enable*)

This function set the 9-bit mode for LPUART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

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

20.2.7.7 static void LPUART_SetMatchAddress (LPUART_Type * *base*, uint16_t *address1*, uint16_t *address2*) [inline], [static]

This function configures the address for LPUART module that works as slave in 9-bit data mode. One or two address fields can be configured. When the address field's match enable bit is set, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches one of slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer, otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note

Any LPUART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>address1</i>	LPUART slave address1.
<i>address2</i>	LPUART slave address2.

20.2.7.8 static void LPUART_EnableMatchAddress (LPUART_Type * *base*, bool *match1*, bool *match2*) [inline], [static]

Parameters

<i>base</i>	LPUART peripheral base address.
<i>match1</i>	true to enable match address1, false to disable.
<i>match2</i>	true to enable match address2, false to disable.

20.2.7.9 static void LPUART_SetRxFifoWatermark (LPUART_Type * *base*, uint8_t *water*) [inline], [static]

Parameters

<i>base</i>	LPUART peripheral base address.
<i>water</i>	Rx FIFO watermark.

20.2.7.10 static void LPUART_SetTxFifoWatermark (LPUART_Type * *base*, uint8_t *water*) [inline], [static]

Parameters

<i>base</i>	LPUART peripheral base address.
<i>water</i>	Tx FIFO watermark.

20.2.7.11 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))
*     {
*     ...
*     }
```

Parameters

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

Returns

LPUART status flags which are ORed by the enumerators in the `_lpuart_flags`.

20.2.7.12 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_NoiseErrorFlag, kLPUART_ParityErrorFlag, 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 <code>_lpuart_status_flag_t</code> to do the OR operation and get the mask.

Returns

0 succeed, others failed.

Return values

<i>kStatus_LPUART_Flag_CannotClearManually</i>	The flag can't be cleared by this function but it is cleared automatically by hardware.
<i>kStatus_Success</i>	Status in the mask are cleared.

20.2.7.13 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 _lpuart_interrupt_enable .

20.2.7.14 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 _lpuart_interrupt_enable .

20.2.7.15 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](#).

20.2.7.16 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.

20.2.7.17 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.

20.2.7.18 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.

20.2.7.19 uint32_t LPUART_GetInstance (LPUART_Type * *base*)

Parameters

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

Returns

LPUART instance.

20.2.7.20 static void LPUART_EnableTx (LPUART_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the LPUART transmitter.

Parameters

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

20.2.7.21 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.

20.2.7.22 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.

20.2.7.23 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.

20.2.7.24 static uint8_t LPUART_GetRx_fifoCount (LPUART_Type * *base*) [inline], [static]

Parameters

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

Returns

rx FIFO data count.

20.2.7.25 static uint8_t LPUART_GetTxFifoCount (LPUART_Type * *base*) [inline], [static]

Parameters

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

Returns

tx FIFO data count.

20.2.7.26 void LPUART_SendAddress (LPUART_Type * *base*, uint8_t *address*)

Parameters

<i>base</i>	LPUART peripheral base address.
<i>address</i>	LPUART slave address.

20.2.7.27 status_t LPUART_WriteBlocking (LPUART_Type * *base*, const uint8_t * *data*, size_t *length*)

This function polls the transmitter register, first waits for the register to be empty or TX FIFO to have room, and writes data to the transmitter buffer, then waits for the dat to be sent out to the bus.

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.

Return values

<i>kStatus_LPUART_- Timeout</i>	Transmission timed out and was aborted.
<i>kStatus_Success</i>	Successfully wrote all data.

20.2.7.28 **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_LPUART_- Timeout</i>	Transmission timed out and was aborted.
<i>kStatus_Success</i>	Successfully received all data.

20.2.7.29 **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

<code>base</code>	LPUART peripheral base address.
<code>handle</code>	LPUART handle pointer.
<code>callback</code>	Callback function.
<code>userData</code>	User data.

20.2.7.30 `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

<code>base</code>	LPUART peripheral base address.
<code>handle</code>	LPUART handle pointer.
<code>xfer</code>	LPUART transfer structure, see Ipuart_transfer_t .

Return values

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

20.2.7.31 `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.

20.2.7.32 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.

20.2.7.33 size_t LPUART_TransferGetRxRingBufferLength (`LPUART_Type * base, Ipuart_handle_t * handle`)

Parameters

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

Returns

Length of received data in RX ring buffer.

20.2.7.34 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.

20.2.7.35 status_t LPUART_TransferGetSendCount (**LPUART_Type * base,** **Ipuart_handle_t * handle, uint32_t * count**)

This function gets the number of bytes that have been sent out to bus 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_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> ;

20.2.7.36 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.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	LPUART handle pointer.
<i>xfer</i>	LPUART transfer structure, see <code>uart_transfer_t</code> .
<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.

20.2.7.37 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.

20.2.7.38 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;

20.2.7.39 void LPUART_TransferHandleIRQ (LPUART_Type * *base*, void * *irqHandle*)

This function handles the LPUART transmit and receive IRQ request.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>irqHandle</i>	LPUART handle pointer.

20.2.7.40 void LPUART_TransferHandleErrorIRQ (LPUART_Type * *base*, void * *irqHandle*)

This function handles the LPUART error IRQ request.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>irqHandle</i>	LPUART handle pointer.

20.3 LPUART eDMA Driver

20.3.1 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.

Driver version

- #define [FSL_LPUART_EDMA_DRIVER_VERSION](#) ([MAKE_VERSION](#)(2, 5, 2))
LPUART EDMA driver version.

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.
- void [LPUART_TransferEdmaHandleIRQ](#) (LPUART_Type *base, void *lpuartEdmaHandle)
LPUART eDMA IRQ handle function.

20.3.2 Data Structure Documentation

20.3.2.1 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.

Field Documentation

- (1) `lpuart_edma_transfer_callback_t lpuart_edma_handle_t::callback`
- (2) `void* lpuart_edma_handle_t::userData`
- (3) `size_t lpuart_edma_handle_t::rxDataSizeAll`
- (4) `size_t lpuart_edma_handle_t::txDataSizeAll`
- (5) `edma_handle_t* lpuart_edma_handle_t::txEdmaHandle`
- (6) `edma_handle_t* lpuart_edma_handle_t::rxEdmaHandle`
- (7) `uint8_t lpuart_edma_handle_t::nbytes`
- (8) `volatile uint8_t lpuart_edma_handle_t::txState`

20.3.3 Macro Definition Documentation

20.3.3.1 #define FSL_LPUART_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 5, 2))

20.3.4 Typedef Documentation

20.3.4.1 `typedef void(* lpuart_edma_transfer_callback_t)(LPUART_Type *base, lpuart_edma_handle_t *handle, status_t status, void *userData)`

20.3.5 Function Documentation

20.3.5.1 `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)`

Note

This function disables all LPUART interrupts.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>handle</i>	Pointer to <code>lpuart_edma_handle_t</code> 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.

20.3.5.2 `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 lpuart_transfer_t .

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.

20.3.5.3 status_t LPUART_ReceiveEDMA (LPUART_Type * *base*, Ipuart_edma_handle_t * *handle*, Ipuart_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 Ipuart_edma_handle_t structure.
<i>xfer</i>	LPUART eDMA transfer structure, see Ipuart_transfer_t .

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.

20.3.5.4 void LPUART_TransferAbortSendEDMA (LPUART_Type * *base*, Ipuart_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 Ipuart_edma_handle_t structure.

20.3.5.5 void LPUART_TransferAbortReceiveEDMA (LPUART_Type * *base*, Ipuart_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.

20.3.5.6 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.

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> ;

20.3.5.7 status_t LPUART_TransferGetReceiveCountEDMA (LPUART_Type * *base*, lpuart_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_NoTransferInProgress</i>	No receive in progress.
<i>kStatus_InvalidArgument</i>	Parameter is invalid.
<i>kStatus_Success</i>	Get successfully through the parameter count;

20.3.5.8 void LPUART_TransferEdmaHandleIRQ (LPUART_Type * *base*, void * *lpuartEdmaHandle*)

This function handles the LPUART tx complete IRQ request and invoke user callback. It is not set to static so that it can be used in user application.

Note

This function is used as default IRQ handler by double weak mechanism. If user's specific IRQ handler is implemented, make sure this function is invoked in the handler.

Parameters

<i>base</i>	LPUART peripheral base address.
<i>lpuartEdmaHandle</i>	LPUART handle pointer.

20.4 LPUART FreeRTOS Driver

20.4.1 Overview

Data Structures

- struct `lpuart_rtos_config_t`
LPUART RTOS configuration structure. [More...](#)

Driver version

- #define `FSL_LPUART_FREERTOS_DRIVER_VERSION` (`MAKE_VERSION(2, 6, 0)`)
LPUART FreeRTOS driver version.

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, 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.
- int `LPUART_RTOS_SetRxTimeout` (`lpuart_rtos_handle_t *handle, uint32_t rx_timeout_constant_ms, uint32_t rx_timeout_multiplier_ms`)
Set RX timeout in runtime.
- int `LPUART_RTOS_SetTxTimeout` (`lpuart_rtos_handle_t *handle, uint32_t tx_timeout_constant_ms, uint32_t tx_timeout_multiplier_ms`)
Set TX timeout in runtime.

20.4.2 Data Structure Documentation

20.4.2.1 struct `lpuart_rtos_config_t`

Data Fields

- `LPUART_Type * base`
UART base address.

- `uint32_t srclk`
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.
- `uint32_t rx_timeout_constant_ms`
RX timeout applied per receive.
- `uint32_t rx_timeout_multiplier_ms`
RX timeout added for each byte of the receive.
- `uint32_t tx_timeout_constant_ms`
TX timeout applied per transmission.
- `uint32_t tx_timeout_multiplier_ms`
TX timeout added for each byte of the transmission.
- `bool enableRxRTS`
RX RTS enable.
- `bool enableTxCTS`
TX CTS enable.
- `lpuart_transmit_cts_source_t txCtsSource`
TX CTS source.
- `lpuart_transmit_cts_config_t txCtsConfig`
TX CTS configure.

Field Documentation

- (1) `uint32_t lpuart_rtos_config_t::rx_timeout_multiplier_ms`
- (2) `uint32_t lpuart_rtos_config_t::tx_timeout_multiplier_ms`

20.4.3 Macro Definition Documentation

20.4.3.1 #define FSL_LPUART_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 6, 0))

20.4.4 Function Documentation

20.4.4.1 int LPUART_RTOS_Init (`lpuart_rtos_handle_t * handle, lpuart_handle_t * t_handle, const lpuart_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

20.4.4.2 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.
---------------	-------------------------

20.4.4.3 int LPUART_RTOS_Send (*Ipuart_rtos_handle_t * handle, 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.

20.4.4.4 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.

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.

20.4.4.5 int LPUART_RTOSETXTIMEOUT (Ipuart_rtos_handle_t * *handle*, uint32_t *rx_timeout_constant_ms*, uint32_t *rx_timeout_multiplier_ms*)

This function can modify RX timeout between initialization and receive.

param handle The RTOS LPUART handle. param rx_timeout_constant_ms RX timeout applied per receive. param rx_timeout_multiplier_ms RX timeout added for each byte of the receive.

20.4.4.6 int LPUART_RTOSETXTIMEOUT (Ipuart_rtos_handle_t * *handle*, uint32_t *tx_timeout_constant_ms*, uint32_t *tx_timeout_multiplier_ms*)

This function can modify TX timeout between initialization and send.

param handle The RTOS LPUART handle. param tx_timeout_constant_ms TX timeout applied per transmission. param tx_timeout_multiplier_ms TX timeout added for each byte of the transmission.

20.5 LPUART CMSIS Driver

This section describes the programming interface of the LPUART Cortex Microcontroller Software Interface Standard (CMSIS) driver. And this driver defines generic peripheral driver interfaces for middleware making it reusable across a wide range of supported microcontroller devices. The API connects microcontroller peripherals with middleware that implements for example communication stacks, file systems, or graphic user interfaces. More information and usage method please refer to <http://www.keil.com/pack/doc/cmsis/Driver/html/index.html>.

The LPUART driver includes transactional APIs.

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 please write custom code.

20.5.1 Function groups

20.5.1.1 LPUART CMSIS GetVersion Operation

This function group will return the LPUART CMSIS Driver version to user.

20.5.1.2 LPUART CMSIS GetCapabilities Operation

This function group will return the capabilities of this driver.

20.5.1.3 LPUART CMSIS Initialize and Uninitialize Operation

This function will initialize and uninitialized the lpuart instance . And this API must be called before you configure a lpuart instance or after you Deinit a lpuart instance.The right steps to start an instance is that you must initialize the instance which been selected firstly,then you can power on the instance.After these all have been done,you can configure the instance by using control operation.If you want to Uninitialize the instance, you must power off the instance first.

20.5.1.4 LPUART CMSIS Transfer Operation

This function group controls the transfer, send/receive data.

20.5.1.5 LPUART CMSIS Status Operation

This function group gets the LPUART transfer status.

20.5.1.6 LPUART CMSIS Control Operation

This function can configure an instance ,set baudrate for lpuart, get current baudrate ,set transfer data bits and other control command.

Chapter 21

PMC: Power Management Controller

21.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Power Management Controller (PMC) module of MCUXpresso SDK devices. The PMC module contains internal voltage regulator, power on reset, low-voltage detect system, and high-voltage detect system.

Data Structures

- struct [pmc_low_volt_detect_config_t](#)
Low-voltage Detect Configuration Structure. [More...](#)
- struct [pmc_low_volt_warning_config_t](#)
Low-voltage Warning Configuration Structure. [More...](#)

Driver version

- #define [FSL_PMC_DRIVER_VERSION\(MAKE_VERSION\(2, 0, 3\)\)](#)
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.
- 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.

21.2 Data Structure Documentation

21.2.1 struct pmc_low_volt_detect_config_t

Data Fields

- bool [enableInt](#)

- **bool enableReset**
Enable system reset when Low-voltage detect.

21.2.2 struct pmc_low_volt_warning_config_t

Data Fields

- **bool enableInt**
Enable interrupt when low-voltage warning.

21.3 Macro Definition Documentation

21.3.1 #define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

Version 2.0.3.

21.4 Function Documentation

21.4.1 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

<i>base</i>	PMC peripheral base address.
<i>config</i>	Low-voltage detect configuration structure.

21.4.2 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

21.4.3 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.
-------------	------------------------------

21.4.4 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.

21.4.5 static bool PMC_GetLowVoltWarningFlag (**PMC_Type * *base*) [inline], [static]**

This function polls the current LWWF status. When 1 is returned, it indicates a low-voltage warning event. LWWF 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 LWWF status
- true: Low-voltage Warning Flag is set.
 - false: the Low-voltage Warning does not happen.

21.4.6 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.
-------------	------------------------------

Chapter 22

PORT: Port Control and Interrupts

22.1 Overview

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

Data Structures

- struct `port_digital_filter_config_t`
PORT digital filter feature configuration definition. [More...](#)
- struct `port_pin_config_t`
PORT pin configuration structure. [More...](#)

Enumerations

- enum `_port_pull` {
 `kPORT_PullDisable` = 0U,
 `kPORT_PullDown` = 2U,
 `kPORT_PullUp` = 3U }
Internal resistor pull feature 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_lock_register` {
 `kPORT_UnlockRegister` = 0U,
 `kPORT_LockRegister` = 1U }
Unlock/lock the pin control register field[15:0].
- enum `port_mux_t` {

```
kPORT_PinDisabledOrAnalog = 0U,
kPORT_MuxAsGpio = 1U,
kPORT_MuxAlt2 = 2U,
kPORT_MuxAlt3 = 3U,
kPORT_MuxAlt4 = 4U,
kPORT_MuxAlt5 = 5U,
kPORT_MuxAlt6 = 6U,
kPORT_MuxAlt7 = 7U,
kPORT_MuxAlt8 = 8U,
kPORT_MuxAlt9 = 9U,
kPORT_MuxAlt10 = 10U,
kPORT_MuxAlt11 = 11U,
kPORT_MuxAlt12 = 12U,
kPORT_MuxAlt13 = 13U,
kPORT_MuxAlt14 = 14U,
kPORT_MuxAlt15 = 15U }
```

Pin mux selection.

- enum `port_interrupt_t` {


```
kPORT_InterruptOrDMADisabled = 0x0U,
kPORT_DMARisingEdge = 0x1U,
kPORT_DMAFallingEdge = 0x2U,
kPORT_DMAEitherEdge = 0x3U,
kPORT_FlagRisingEdge = 0x05U,
kPORT_FlagFallingEdge = 0x06U,
kPORT_FlagEitherEdge = 0x07U,
kPORT_InterruptLogicZero = 0x8U,
kPORT_InterruptRisingEdge = 0x9U,
kPORT_InterruptFallingEdge = 0xAU,
kPORT_InterruptEitherEdge = 0xBU,
kPORT_InterruptLogicOne = 0xCU,
kPORT_ActiveHighTriggerOutputEnable = 0xDU,
kPORT_ActiveLowTriggerOutputEnable = 0xEU }
```

Configures the interrupt generation condition.

- enum `port_digital_filter_clock_source_t` {


```
kPORT_BusClock = 0U,
kPORT_LpoClock = 1U }
```

Digital filter clock source selection.

Driver version

- #define `FSL_PORT_DRIVER_VERSION` (`MAKE_VERSION(2, 3, 0)`)
PORT driver version.

Configuration

- static void `PORT_SetPinConfig` (`PORT_Type` *base, `uint32_t` pin, const `port_pin_config_t` *config)

- static void **PORT_SetMultiplePinsConfig** (PORT_Type *base, uint32_t mask, const **port_pin_config_t** *config)

Sets the port PCR register for multiple pins.
- static void **PORT_SetPinMux** (PORT_Type *base, uint32_t pin, **port_mux_t** mux)

Configures the pin muxing.
- static void **PORT_EnablePinsDigitalFilter** (PORT_Type *base, uint32_t mask, bool enable)

Enables the digital filter in one port, each bit of the 32-bit register represents one pin.
- static void **PORT_SetDigitalFilterConfig** (PORT_Type *base, const **port_digital_filter_config_t** *config)

Sets the digital filter in one port, each bit of the 32-bit register represents one pin.

Interrupt

- static void **PORT_SetPinInterruptConfig** (PORT_Type *base, uint32_t pin, **port_interrupt_t** config)

Configures the port pin interrupt/DMA request.
- static void **PORT_SetPinDriveStrength** (PORT_Type *base, uint32_t pin, uint8_t strength)

Configures the port pin drive strength.
- 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.

22.2 Data Structure Documentation

22.2.1 struct port_digital_filter_config_t

Data Fields

- uint32_t **digitalFilterWidth**

Set digital filter width.
- **port_digital_filter_clock_source_t** **clockSource**

Set digital filter clockSource.

22.2.2 struct port_pin_config_t

Data Fields

- uint16_t **pullSelect**: 2

No-pull/pull-down/pull-up select.
- uint16_t **passiveFilterEnable**: 1

Passive filter enable/disable.
- uint16_t **driveStrength**: 1

Fast/slow drive strength configure.
- uint16_t **mux**: 3

Pin mux Configure.
- uint16_t **lockRegister**: 1

Lock/unlock the PCR field[15:0].

22.3 Macro Definition Documentation

22.3.1 #define FSL_PORT_DRIVER_VERSION (MAKE_VERSION(2, 3, 0))

22.4 Enumeration Type Documentation

22.4.1 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.

22.4.2 enum _port_passive_filter_enable

Enumerator

kPORT_PassiveFilterDisable Passive input filter is disabled.

kPORT_PassiveFilterEnable Passive input filter is enabled.

22.4.3 enum _port_drive_strength

Enumerator

kPORT_LowDriveStrength Low-drive strength is configured.

kPORT_HighDriveStrength High-drive strength is configured.

22.4.4 enum _port_lock_register

Enumerator

kPORT_UnlockRegister Pin Control Register fields [15:0] are not locked.

kPORT_LockRegister Pin Control Register fields [15:0] are locked.

22.4.5 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.

22.4.6 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_FlagRisingEdge Flag sets on rising edge.
kPORT_FlagFallingEdge Flag sets on falling edge.
kPORT_FlagEitherEdge Flag sets 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.
kPORT_ActiveHighTriggerOutputEnable Enable active high-trigger output.
kPORT_ActiveLowTriggerOutputEnable Enable active low-trigger output.

22.4.7 enum port_digital_filter_clock_source_t

Enumerator

kPORT_BusClock Digital filters are clocked by the bus clock.
kPORT_LpoClock Digital filters are clocked by the 1 kHz LPO clock.

22.5 Function Documentation

22.5.1 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.

22.5.2 static void PORT_SetMultiplePinsConfig (PORT_Type * *base*, uint32_t *mask*, const port_pin_config_t * *config*) [inline], [static]

This is an example to define input pins or output pins PCR configuration.

```
* Define a digital input pin PCR configuration
* port_pin_config_t config = {
*     kPORT_PullUp ,
*     kPORT_PullEnable,
*     kPORT_FastSlewRate,
*     kPORT_PassiveFilterDisable,
*     kPORT_OpenDrainDisable,
*     kPORT_LowDriveStrength,
*     kPORT_MuxAsGpio,
*     kPORT_UnlockRegister,
* };
*
```

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.
<i>config</i>	PORT PCR register configuration structure.

22.5.3 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>	<p>pin muxing slot selection.</p> <ul style="list-style-type: none"> • kPORT_PinDisabledOrAnalog: Pin disabled or work in analog function. • kPORT_MuxAsGpio : Set as GPIO. • kPORT_MuxAlt2 : chip-specific. • kPORT_MuxAlt3 : chip-specific. • kPORT_MuxAlt4 : chip-specific. • kPORT_MuxAlt5 : chip-specific. • kPORT_MuxAlt6 : chip-specific. • kPORT_MuxAlt7 : chip-specific.

Note

: This function is NOT recommended to use together with the PORT_SetPinsConfig, because the PORT_SetPinsConfig need to configure the pin mux anyway (Otherwise the pin mux is reset to zero : [kPORT_PinDisabledOrAnalog](#)). This function is recommended to use to reset the pin mux

22.5.4 static void PORT_EnablePinsDigitalFilter (PORT_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	PORT peripheral base pointer.
-------------	-------------------------------

<i>mask</i>	PORT pin number macro.
<i>enable</i>	PORT digital filter configuration.

22.5.5 static void PORT_SetDigitalFilterConfig (PORT_Type * *base*, const port_digital_filter_config_t * *config*) [inline], [static]

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>config</i>	PORT digital filter configuration structure.

22.5.6 static void PORT_SetPinInterruptConfig (PORT_Type * *base*, uint32_t *pin*, port_interrupt_t *config*) [inline], [static]

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>config</i>	PORT pin interrupt configuration. <ul style="list-style-type: none"> • kPORT_InterruptOrDMADisabled: Interrupt/DMA request disabled. • kPORT_DMARisingEdge : DMA request on rising edge(if the DMA requests exit). • kPORT_DMAPFallingEdge: DMA request on falling edge(if the DMA requests exit). • kPORT_DMAEitherEdge : DMA request on either edge(if the DMA requests exit). • kPORT_FlagRisingEdge : Flag sets on rising edge(if the Flag states exit). • kPORT_FlagFallingEdge : Flag sets on falling edge(if the Flag states exit). • kPORT_FlagEitherEdge : Flag sets on either edge(if the Flag states exit). • kPORT_InterruptLogicZero : Interrupt when logic zero. • kPORT_InterruptRisingEdge : Interrupt on rising edge. • kPORT_InterruptFallingEdge: Interrupt on falling edge. • kPORT_InterruptEitherEdge : Interrupt on either edge. • kPORT_InterruptLogicOne : Interrupt when logic one. • kPORT_ActiveHighTriggerOutputEnable : Enable active high-trigger output (if the trigger states exit). • kPORT_ActiveLowTriggerOutputEnable : Enable active low-trigger output (if the trigger states exit).

22.5.7 static void PORT_SetPinDriveStrength (**PORT_Type * *base*, **uint32_t** *pin*, **uint8_t** *strength*) [inline], [static]**

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>pin</i>	PORT pin number.
<i>strength</i>	PORT pin drive strength <ul style="list-style-type: none"> • kPORT_LowDriveStrength = 0U - Low-drive strength is configured. • kPORT_HighDriveStrength = 1U - High-drive strength is configured.

22.5.8 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.

22.5.9 static void PORT_ClearPinsInterruptFlags (**PORT_Type * *base*, **uint32_t** *mask*) [inline], [static]**

Parameters

<i>base</i>	PORT peripheral base pointer.
<i>mask</i>	PORT pin number macro.

Chapter 23

PWT: Pulse Width Timer

23.1 Overview

The MCUXpresso SDK provides a driver for the Pulse Width Timer (PWT) of MCUXpresso SDK devices.

23.2 Function groups

The PWT driver supports capture or measure the pulse width mapping on its input channels. The counter of PWT has two selectable clock sources, and supports up to BUS_CLK with internal timer clock. PWT module supports programmable positive or negative pulse edges, and programmable interrupt generation upon pulse width values or counter overflow.

23.2.1 Initialization and deinitialization

The function [PWT_Init\(\)](#) initializes the PWT with specified configurations. The function [PWT_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the PWT for the requested register update mode for register with buffers.

The function [PWT_Deinit\(\)](#) disables the PWT counter and turns off the module clock.

23.2.2 Reset

The function [PWT_Reset\(\)](#) is built into PWT as a mechanism used to reset/restart the pulse width timer.

23.2.3 Status

Provides functions to get and clear the PWT status.

23.2.4 Interrupt

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

23.2.5 Start & Stop timer

The function [PWT_StartTimer\(\)](#) starts the PWT time counter.

The function [PWT_StopTimer\(\)](#) stops the PWT time counter.

23.2.6 GetInterrupt

Provides functions to generate Overflow/Pulse Width Data Ready Interrupt.

23.2.7 Get Timer value

The function [PWT_GetCurrentTimerCount\(\)](#) is set to read the current counter value.

The function [PWT_ReadPositivePulseWidth\(\)](#) is set to read the positive pulse width.

The function [PWT_ReadNegativePulseWidth\(\)](#) is set to read the negative pulse width.

23.2.8 PWT Operations

Input capture operations

The input capture operations sets up an channel for input capture.

The function EdgeCapture can be used to measure the pulse width of a signal. A channel is used during capture with the input signal coming through a channel n. The capture edge for each channel, and any filter value to be used when processing the input signal.

23.3 Typical use case

23.3.1 PWT measure

This is an example code to measure the pulse width:

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/pwt

Data Structures

- struct [pwt_config_t](#)
PWT configuration structure. [More...](#)

Macros

- #define [FSL_PWT_DRIVER_VERSION](#) (MAKE_VERSION(2, 0, 1))
Version 2.0.1.

Enumerations

- enum [pwt_clock_source_t](#) {

 kPWT_BusClock = 0U,

 kPWT_AlternativeClock
 }

PWT clock source selection.

- enum `pwt_clock_prescale_t` {
 `kPWT_Prescale_Divide_1` = 0U,
 `kPWT_Prescale_Divide_2`,
 `kPWT_Prescale_Divide_4`,
 `kPWT_Prescale_Divide_8`,
 `kPWT_Prescale_Divide_16`,
 `kPWT_Prescale_Divide_32`,
 `kPWT_Prescale_Divide_64`,
 `kPWT_Prescale_Divide_128` }

PWT prescaler factor selection for clock source.
- enum `pwt_input_select_t` {
 `kPWT_InputPort_0` = 0U,
 `kPWT_InputPort_1`,
 `kPWT_InputPort_2`,
 `kPWT_InputPort_3` }

PWT input port selection.
- enum `_pwt_interrupt_enable` {
 `kPWT_PulseWidthReadyInterruptEnable` = PWT_CS_PRDYIE_MASK,
 `kPWT_CounterOverflowInterruptEnable` = PWT_CS_POVIE_MASK }

List of PWT interrupts.
- enum `_pwt_status_flags` {
 `kPWT_CounterOverflowFlag` = PWT_CS_PWTOV_MASK,
 `kPWT_PulseWidthValidFlag` = PWT_CS_PWT RDY MASK }

List of PWT flags.

Functions

- static uint16_t `PWT_GetCurrentTimerCount` (PWT_Type *base)

Reads the current counter value.
- static uint16_t `PWT_ReadPositivePulseWidth` (PWT_Type *base)

Reads the positive pulse width.
- static uint16_t `PWT_ReadNegativePulseWidth` (PWT_Type *base)

Reads the negative pulse width.
- static void `PWT_Reset` (PWT_Type *base)

Performs a software reset on the PWT module.

Initialization and deinitialization

- void `PWT_Init` (PWT_Type *base, const `pwt_config_t` *config)

Ungates the PWT clock and configures the peripheral for basic operation.
- void `PWT_Deinit` (PWT_Type *base)

Gates the PWT clock.
- void `PWT_GetDefaultConfig` (`pwt_config_t` *config)

Fills in the PWT configuration structure with the default settings.

Interrupt Interface

- static void `PWT_EnableInterrupts` (PWT_Type *base, uint32_t mask)

- static void [PWT_DisableInterrupts](#) (PWT_Type *base, uint32_t mask)
Disables the selected PWT interrupts.
- static uint32_t [PWT_GetEnabledInterrupts](#) (PWT_Type *base)
Gets the enabled PWT interrupts.

Status Interface

- static uint32_t [PWT_GetStatusFlags](#) (PWT_Type *base)
Gets the PWT status flags.
- static void [PWT_ClearStatusFlags](#) (PWT_Type *base, uint32_t mask)
Clears the PWT status flags.

Timer Start and Stop

- static void [PWT_StartTimer](#) (PWT_Type *base)
Starts the PWT counter.
- static void [PWT_StopTimer](#) (PWT_Type *base)
Stops the PWT counter.

23.4 Data Structure Documentation

23.4.1 struct pwt_config_t

This structure holds the configuration settings for the PWT peripheral. To initialize this structure to reasonable defaults, call the [PWT_GetDefaultConfig\(\)](#) function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

Data Fields

- [pwt_clock_source_t clockSource](#)
Clock source for the counter.
- [pwt_clock_prescale_t prescale](#)
Pre-scaler to divide down the clock.
- [pwt_input_select_t inputSelect](#)
PWT Pulse input port selection.
- bool [enableFirstCounterLoad](#)
true: Load the first counter value to registers; false: Do not load first counter value

23.5 Enumeration Type Documentation

23.5.1 enum pwt_clock_source_t

Enumerator

kPWT_BusClock The Bus clock is used as the clock source of PWT counter.

kPWT_AlternativeClock Alternative clock is used as the clock source of PWT counter.

23.5.2 enum pwt_clock_prescale_t

Enumerator

- kPWT_Prescale_Divide_1* PWT clock divided by 1.
- kPWT_Prescale_Divide_2* PWT clock divided by 2.
- kPWT_Prescale_Divide_4* PWT clock divided by 4.
- kPWT_Prescale_Divide_8* PWT clock divided by 8.
- kPWT_Prescale_Divide_16* PWT clock divided by 16.
- kPWT_Prescale_Divide_32* PWT clock divided by 32.
- kPWT_Prescale_Divide_64* PWT clock divided by 64.
- kPWT_Prescale_Divide_128* PWT clock divided by 128.

23.5.3 enum pwt_input_select_t

Enumerator

- kPWT_InputPort_0* PWT input comes from PWTIN[0].
- kPWT_InputPort_1* PWT input comes from PWTIN[1].
- kPWT_InputPort_2* PWT input comes from PWTIN[2].
- kPWT_InputPort_3* PWT input comes from PWTIN[3].

23.5.4 enum _pwt_interrupt_enable

Enumerator

- kPWT_PulseWidthReadyInterruptEnable* Pulse width data ready interrupt.
- kPWT_CounterOverflowInterruptEnable* Counter overflow interrupt.

23.5.5 enum _pwt_status_flags

Enumerator

- kPWT_CounterOverflowFlag* Counter overflow flag.
- kPWT_PulseWidthValidFlag* Pulse width valid flag.

23.6 Function Documentation

23.6.1 void PWT_Init (PWT_Type * *base*, const pwt_config_t * *config*)

Note

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

Parameters

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

23.6.2 void PWT_Deinit (PWT_Type * *base*)

Parameters

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

23.6.3 void PWT_GetDefaultConfig (pwt_config_t * *config*)

The default values are:

```
* config->clockSource = kPWT_BusClock;
* config->prescale = kPWT_Prescale_Divide_1;
* config->inputSelect = kPWT_InputPort_0;
* config->enableFirstCounterLoad = false;
*
```

Parameters

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

23.6.4 static void PWT_EnableInterrupts (PWT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PWT peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration pwt_interrupt_enable_t

23.6.5 static void PWT_DisableInterrupts (PWT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PWT peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration pwt_interrupt_enable_t

23.6.6 static uint32_t PWT_GetEnabledInterrupts (PWT_Type * *base*) [inline], [static]

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration pwt_interrupt_enable_t

23.6.7 static uint32_t PWT_GetStatusFlags (PWT_Type * *base*) [inline], [static]

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration pwt_status_flags_t

23.6.8 static void PWT_ClearStatusFlags (PWT_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PWT peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration pwt_status_flags_t

23.6.9 static void PWT_StartTimer(PWT_Type * *base*) [inline], [static]

Parameters

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

23.6.10 static void PWT_StopTimer(PWT_Type * *base*) [inline], [static]

Parameters

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

23.6.11 static uint16_t PWT_GetCurrentTimerCount(PWT_Type * *base*) [inline], [static]

This function returns the timer counting value

Parameters

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

Returns

Current 16-bit timer counter value

23.6.12 static uint16_t PWT_ReadPositivePulseWidth(PWT_Type * *base*) [inline], [static]

This function reads the low and high registers and returns the 16-bit positive pulse width

Parameters

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

Returns

The 16-bit positive pulse width.

23.6.13 static uint16_t PWT_ReadPositivePulseWidth (PWT_Type * *base*) [inline], [static]

This function reads the low and high registers and returns the 16-bit positive pulse width

Parameters

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

Returns

The 16-bit positive pulse width.

23.6.14 static void PWT_Reset (PWT_Type * *base*) [inline], [static]

Parameters

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

Chapter 24

RCM: Reset Control Module Driver

24.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Reset Control Module (RCM) module of MCUXpresso SDK devices.

Data Structures

- struct `rcm_version_id_t`
IP version ID definition. [More...](#)
- struct `rcm_reset_pin_filter_config_t`
Reset pin filter configuration. [More...](#)

Enumerations

- enum `rcm_reset_source_t`{
 `kRCM_SourceLvd` = RCM_SRS_LVD_MASK,
 `kRCM_SourceLoc` = RCM_SRS_LOC_MASK,
 `kRCM_SourceLol` = RCM_SRS_LOL_MASK,
 `kRCM_SourceWdog` = RCM_SRS_WDOG_MASK,
 `kRCM_SourcePin` = RCM_SRS_PIN_MASK,
 `kRCM_SourcePor` = RCM_SRS_POR_MASK,
 `kRCM_SourceLockup` = RCM_SRS_LOCKUP_MASK,
 `kRCM_SourceSw` = RCM_SRS_SW_MASK,
 `kRCM_SourceMdmap` = RCM_SRS_MDM_AP_MASK,
 `kRCM_SourceSackerr` = RCM_SRS_SACKERR_MASK }
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_reset_delay_t`{
 `kRCM_ResetDelay8Lpo` = 0U,
 `kRCM_ResetDelay32Lpo` = 1U,
 `kRCM_ResetDelay128Lpo` = 2U,
 `kRCM_ResetDelay512Lpo` = 3U }
Maximum delay time from interrupt asserts to system reset.
- enum `rcm_interrupt_enable_t` {

```

kRCM_IntNone = 0U,
kRCM_IntLossOfClk = RCM_SRIE_LOC_MASK,
kRCM_IntLossOfLock = RCM_SRIE_LOL_MASK,
kRCM_IntWatchDog = RCM_SRIE_WDOG_MASK,
kRCM_IntExternalPin = RCM_SRIE_PIN_MASK,
kRCM_IntGlobal = RCM_SRIE_GIE_MASK,
kRCM_IntCoreLockup = RCM_SRIE_LOCKUP_MASK,
kRCM_IntSoftware = RCM_SRIE_SW_MASK,
kRCM_IntStopModeAckErr = RCM_SRIE_SACKERR_MASK,
kRCM_IntAll }

```

System reset interrupt enable bit definitions.

Driver version

- #define `FSL_RCM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 4)`)
RCM driver version 2.0.4.

Reset Control Module APIs

- static void `RCM_GetVersionId` (RCM_Type *base, `rcm_version_id_t` *versionId)
Gets the RCM version ID.
- static uint32_t `RCM_GetPreviousResetSources` (RCM_Type *base)
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 void `RCM_SetSystemResetInterruptConfig` (RCM_Type *base, uint32_t intMask, `rcm_reset_delay_t` delay)
Sets the system reset interrupt configuration.

24.2 Data Structure Documentation

24.2.1 struct rcm_version_id_t

Data Fields

- `uint16_t feature`
Feature Specification Number.
- `uint8_t minor`
Minor version number.
- `uint8_t major`
Major version number.

Field Documentation

- (1) `uint16_t rcm_version_id_t::feature`
- (2) `uint8_t rcm_version_id_t::minor`
- (3) `uint8_t rcm_version_id_t::major`

24.2.2 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.

Field Documentation

- (1) `bool rcm_reset_pin_filter_config_t::enableFilterInStop`
- (2) `rcm_run_wait_filter_mode_t rcm_reset_pin_filter_config_t::filterInRunWait`
- (3) `uint8_t rcm_reset_pin_filter_config_t::busClockFilterCount`

24.3 Macro Definition Documentation**24.3.1 #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))****24.4 Enumeration Type Documentation****24.4.1 enum rcm_reset_source_t**

Enumerator

- kRCM_SourceLvd* Low-voltage detect reset.
kRCM_SourceLoc Loss of clock reset.
kRCM_SourceLol Loss of lock reset.
kRCM_SourceWdog Watchdog reset.
kRCM_SourcePin External pin reset.
kRCM_SourcePor Power on reset.
kRCM_SourceLockup Core lock up reset.
kRCM_SourceSw Software reset.
kRCM_SourceMdmap MDM-AP system reset.
kRCM_SourceSackerr Parameter could get all reset flags.

24.4.2 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.

24.4.3 enum rcm_reset_delay_t

Enumerator

- kRCM_ResetDelay8Lpo* Delay 8 LPO cycles.
- kRCM_ResetDelay32Lpo* Delay 32 LPO cycles.
- kRCM_ResetDelay128Lpo* Delay 128 LPO cycles.
- kRCM_ResetDelay512Lpo* Delay 512 LPO cycles.

24.4.4 enum rcm_interrupt_enable_t

Enumerator

- kRCM_IntNone* No interrupt enabled.
- kRCM_IntLossOfClk* Loss of clock interrupt.
- kRCM_IntLossOfLock* Loss of lock interrupt.
- kRCM_IntWatchDog* Watch dog interrupt.
- kRCM_IntExternalPin* External pin interrupt.
- kRCM_IntGlobal* Global interrupts.
- kRCM_IntCoreLockup* Core lock up interrupt.
- kRCM_IntSoftware* software interrupt
- kRCM_IntStopModeAckErr* Stop mode ACK error interrupt.
- kRCM_IntAll* Enable all interrupts.

24.5 Function Documentation

24.5.1 static void RCM_GetVersionId(RCM_Type * base, rcm_version_id_t * versionId) [inline], [static]

This function gets the RCM version ID including the major version number, the minor version number, and the feature specification number.

Parameters

<i>base</i>	RCM peripheral base address.
<i>versionId</i>	Pointer to the version ID structure.

24.5.2 static uint32_t RCM_GetPreviousResetSources (RCM_Type * *base*) [inline], [static]

This function gets the current reset source status. Use source masks defined in the rcm_reset_source_t to get the desired source status.

This is an example.

```
* uint32_t resetStatus;
*
* To get all reset source statuses.
* resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;
*
* To test whether the MCU is reset using Watchdog.
* resetStatus = RCM_GetPreviousResetSources(RCM) &
*               kRCM_SourceWdog;
*
* To test multiple reset sources.
* resetStatus = RCM_GetPreviousResetSources(RCM) & (
*               kRCM_SourceWdog | kRCM_SourcePin);
*
```

Parameters

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

Returns

All reset source status bit map.

24.5.3 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.

24.5.4 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

24.5.5 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.

24.5.6 static void RCM_SetSystemResetInterruptConfig (RCM_Type * *base*, uint32_t *intMask*, rcm_reset_delay_t *delay*) [inline], [static]

For a graceful shut down, the RCM supports delaying the assertion of the system reset for a period of time when the reset interrupt is generated. This function can be used to enable the interrupt and the delay period. The interrupts are passed in as bit mask. See `rcm_int_t` for details. For example, to delay a reset for 512 LPO cycles after the WDOG timeout or loss-of-clock occurs, configure as follows: `RCM_SetSystemResetInterruptConfig(kRCM_IntWatchDog | kRCM_IntLossOfClk, kRCM_ResetDelay512Lpo);`

Parameters

<i>base</i>	RCM peripheral base address.
<i>intMask</i>	Bit mask of the system reset interrupts to enable. See <code>rcm_interrupt_enable_t</code> for details.
<i>delay</i>	Bit mask of the system reset interrupts to enable.

Chapter 25

SIM: System Integration Module Driver

25.1 Overview

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

Data Structures

- struct `sim_uid_t`
Unique ID. [More...](#)

Enumerations

- enum `_sim_flash_mode` {
 `kSIM_FlashDisableInWait` = SIM_FCFG1_FLASHDOZE_MASK,
 `kSIM_FlashDisable` = SIM_FCFG1_FLASHDIS_MASK }
Flash enable mode.

Functions

- 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, 1, 3)`)

25.2 Data Structure Documentation

25.2.1 struct `sim_uid_t`

Data Fields

- `uint32_t MH`
UIDMH.
- `uint32_t ML`
UIDML.
- `uint32_t L`
UIDL.

Field Documentation

- (1) `uint32_t sim_uid_t::MH`
- (2) `uint32_t sim_uid_t::ML`
- (3) `uint32_t sim_uid_t::L`

25.3 Enumeration Type Documentation

25.3.1 `enum _sim_flash_mode`

Enumerator

`kSIM_FlashDisableInWait` Disable flash in wait mode.

`kSIM_FlashDisable` Disable flash in normal mode.

25.4 Function Documentation

25.4.1 `void SIM_GetUniqueId (sim_uid_t * uid)`

Parameters

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

25.4.2 `static void SIM_SetFlashMode (uint8_t mode) [inline], [static]`

Parameters

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

Chapter 26

SMC: System Mode Controller Driver

26.1 Overview

The MCUXpresso SDK provides a peripheral driver for the System Mode Controller (SMC) module of MCUXpresso SDK devices. The SMC module sequences the system in and out of all low-power stop and run modes.

API functions are provided to configure the system for working in a dedicated power mode. For different power modes, `SMC_SetPowerModeXXX()` function accepts different parameters. System power mode state transitions are not available between power modes. For details about available transitions, see the power mode transitions section in the SoC reference manual.

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.

Disable/enable the interrupt through PRIMASK. This is an example use case. The application sets the wakeup interrupt and calls SMC function `SMC_SetPowerModeStop` to set the MCU to STOP mode, but the wakeup interrupt happens so quickly that the ISR completes before the function `SMC_SetPowerModeStop`. As a result, the MCU enters the STOP mode and never is woken up by the interrupt. In this use case, the application first disables the interrupt through PRIMASK, sets the wakeup interrupt, and enters the STOP mode. After wakeup, enable the interrupt through PRIMASK. The MCU can still be woken up by disabling the interrupt through PRIMASK. The pre and post functions handle the PRIMASK.

```
SMC_PreEnterStopModes();  
/* Enable the wakeup interrupt here. */  
SMC_SetPowerModeStop(SMC, kSMC_PartialStop);  
SMC_PostExitStopModes();
```

For legacy Kinetis, when entering stop modes, the flash speculation might be interrupted. As a result, the prefetched code or data might be broken. To make sure the flash is idle when entering the stop modes, smc driver allocates a RAM region, the code to enter stop modes are executed in RAM, thus the flash is idle and no prefetch is performed while entering stop modes. Application should make sure that, the rw data of `fsl_smci.c` is located in memory region which is not powered off in stop modes, especially LLS2 modes.

For STOP, VLPS, and LLS3, the whole RAM are powered up, so after woken up, the RAM function could continue executing. For VLLS mode, the system resets after woken up, the RAM content might be re-initialized. For LLS2 mode, only part of RAM are powered on, so application must make sure that, the

rw data of fsl_sm.c is located in memory region which is not powered off, otherwise after woken up, the MCU could not get right code to execute.

Data Structures

- struct `smc_version_id_t`
IP version ID definition. [More...](#)
- struct `smc_param_t`
IP parameter definition. [More...](#)

Enumerations

- enum `smc_power_mode_protection_t` {

 `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 }

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 }

Stop mode definition.
- enum `smc_partial_stop_option_t` {

 `kSMC_PartialStop` = 0U,

 `kSMC_PartialStop1` = 1U,

 `kSMC_PartialStop2` = 2U }

Partial STOP option.
- enum { `kStatus_SMC_StopAbort` = MAKE_STATUS(kStatusGroup_POWER, 0) }

_smc_status, SMC configuration status.

Driver version

- #define `FSL_SMC_DRIVER_VERSION` (MAKE_VERSION(2, 0, 7))

SMC driver version.

System mode controller APIs

- static void `SMC_GetVersionId` (SMC_Type *base, `smc_version_id_t` *versionId)

Gets the SMC version ID.

- void [SMC_GetParam](#) (SMC_Type *base, smc_param_t *param)
Gets the SMC parameter.
- static void [SMC_SetPowerModeProtection](#) (SMC_Type *base, uint8_t allowedModes)
Configures all power mode protection settings.
- static smc_power_state_t [SMC_GetPowerModeState](#) (SMC_Type *base)
Gets the current power mode status.
- void [SMC_PreEnterStopModes](#) (void)
Prepares to enter stop modes.
- void [SMC_PostExitStopModes](#) (void)
Recovers after wake up from stop modes.
- void [SMC_PreEnterWaitModes](#) (void)
Prepares to enter wait modes.
- void [SMC_PostExitWaitModes](#) (void)
Recovers after wake up from stop modes.
- status_t [SMC_SetPowerModeRun](#) (SMC_Type *base)
Configures the system to RUN power mode.
- status_t [SMC_SetPowerModeWait](#) (SMC_Type *base)
Configures the system to WAIT power mode.
- status_t [SMC_SetPowerModeStop](#) (SMC_Type *base, smc_partial_stop_option_t option)
Configures the system to Stop power mode.
- status_t [SMC_SetPowerModeVlpr](#) (SMC_Type *base)
Configures the system to VLPR power mode.
- status_t [SMC_SetPowerModeVlpw](#) (SMC_Type *base)
Configures the system to VLPW power mode.
- status_t [SMC_SetPowerModeVlps](#) (SMC_Type *base)
Configures the system to VLPS power mode.

26.3 Data Structure Documentation

26.3.1 struct smc_version_id_t

Data Fields

- uint16_t **feature**
Feature Specification Number.
- uint8_t **minor**
Minor version number.
- uint8_t **major**
Major version number.

Field Documentation

- (1) **uint16_t smc_version_id_t::feature**
- (2) **uint8_t smc_version_id_t::minor**
- (3) **uint8_t smc_version_id_t::major**

26.3.2 struct smc_param_t

Data Fields

- bool `hsrunEnable`
HSRUN mode enable.
- bool `llsEnable`
LLS mode enable.
- bool `lls2Enable`
LLS2 mode enable.
- bool `vlls0Enable`
VLLS0 mode enable.

Field Documentation

- (1) `bool smc_param_t::hsrunEnable`
- (2) `bool smc_param_t::llsEnable`
- (3) `bool smc_param_t::lls2Enable`
- (4) `bool smc_param_t::vlls0Enable`

26.4 Enumeration Type Documentation

26.4.1 enum smc_power_mode_protection_t

Enumerator

`kSMC_AllowPowerModeVlp` Allow Very-Low-power Mode.
`kSMC_AllowPowerModeAll` Allow all power mode.

26.4.2 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

26.4.3 enum smc_run_mode_t

Enumerator

`kSMC_RunNormal` Normal RUN mode.

kSMC_RunVlpr Very-low-power RUN mode.

26.4.4 enum smc_stop_mode_t

Enumerator

kSMC_StopNormal Normal STOP mode.

kSMC_StopVlps Very-low-power STOP mode.

26.4.5 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.

26.4.6 anonymous enum

Enumerator

kStatus_SMC_StopAbort Entering Stop mode is abort.

26.5 Function Documentation

26.5.1 static void SMC_GetVersionId(SMC_Type * *base*, smc_version_id_t * *versionId*) [inline], [static]

This function gets the SMC version ID, including major version number, minor version number, and feature specification number.

Parameters

<i>base</i>	SMC peripheral base address.
<i>versionId</i>	Pointer to the version ID structure.

26.5.2 void SMC_GetParam(SMC_Type * *base*, smc_param_t * *param*)

This function gets the SMC parameter including the enabled power modes.

Parameters

<i>base</i>	SMC peripheral base address.
<i>param</i>	Pointer to the SMC param structure.

26.5.3 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

<i>base</i>	SMC peripheral base address.
<i>allowedModes</i>	Bitmap of the allowed power modes.

26.5.4 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.

26.5.5 void SMC_PreEnterStopModes (void)

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

26.5.6 void SMC_PostExitStopModes (void)

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

26.5.7 void SMC_PreEnterWaitModes (void)

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

26.5.8 void SMC_PostExitWaitModes (void)

This function should be called after wake up from WAIT/VLPW modes. It is used with [SMC_PreEnterWaitModes](#).

26.5.9 status_t SMC_SetPowerModeRun (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.5.10 status_t SMC_SetPowerModeWait (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.5.11 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.

26.5.12 status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.5.13 status_t SMC_SetPowerModeVlpw (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

26.5.14 status_t SMC_SetPowerModeVlps (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

Chapter 27

TRGMUX: Trigger Mux Driver

27.1 Overview

The MCUXpresso SDK provides driver for the Trigger Mux (TRGMUX) module of MCUXpresso SDK devices.

27.2 Typical use case

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/trgmux

Enumerations

- enum { `kStatus_TRGMUX_Locked` = MAKE_STATUS(kStatusGroup_TRGMUX, 0) }
TRGMUX configure status.
- enum `trgmux_trigger_input_t` {
 `kTRGMUX_TriggerInput0` = TRGMUX_TRGCFG_SEL0_SHIFT,
 `kTRGMUX_TriggerInput1` = TRGMUX_TRGCFG_SEL1_SHIFT,
 `kTRGMUX_TriggerInput2` = TRGMUX_TRGCFG_SEL2_SHIFT,
 `kTRGMUX_TriggerInput3` = TRGMUX_TRGCFG_SEL3_SHIFT }
Defines the MUX select for peripheral trigger input.

Driver version

- #define `FSL_TRGMUX_DRIVER_VERSION` (MAKE_VERSION(2, 0, 1))
TRGMUX driver version.

TRGMUX Functional Operation

- static void `TRGMUX_LockRegister` (TRGMUX_Type *base, uint32_t index)
Sets the flag of the register which is used to mark writeable.
- status_t `TRGMUX_SetTriggerSource` (TRGMUX_Type *base, uint32_t index, `trgmux_trigger_input_t` input, uint32_t trigger_src)
Configures the trigger source of the appointed peripheral.

27.3 Macro Definition Documentation

27.3.1 #define FSL_TRGMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

27.4 Enumeration Type Documentation

27.4.1 anonymous enum

Enumerator

kStatus(TRGMUX_Locked) Configure failed for register is locked.

27.4.2 enum trgmux_trigger_input_t

Enumerator

kTRGMUX_TriggerInput0 The MUX select for peripheral trigger input 0.
kTRGMUX_TriggerInput1 The MUX select for peripheral trigger input 1.
kTRGMUX_TriggerInput2 The MUX select for peripheral trigger input 2.
kTRGMUX_TriggerInput3 The MUX select for peripheral trigger input 3.

27.5 Function Documentation

27.5.1 static void TRGMUX_LockRegister (***TRGMUX_Type * base***, ***uint32_t index***) [inline], [static]

The function sets the flag of the register which is used to mark writeable. Example:

```
TRGMUX_LockRegister(TRGMUX0, kTRGMUX_Trgmux0Dmamux0);
```

Parameters

<i>base</i>	TRGMUX peripheral base address.
<i>index</i>	The index of the TRGMUX register, see the enum trgmux_device_t defined in <SOC>.h.

27.5.2 status_t TRGMUX_SetTriggerSource (***TRGMUX_Type * base***, ***uint32_t index***, ***trgmux_trigger_input_t input***, ***uint32_t trigger_src***)

The function configures the trigger source of the appointed peripheral. Example:

```
TRGMUX_SetTriggerSource(TRGMUX0, kTRGMUX_Trgmux0Dmamux0,
    kTRGMUX_TriggerInput0, kTRGMUX_SourcePortPin);
```

Parameters

<i>base</i>	TRGMUX peripheral base address.
<i>index</i>	The index of the TRGMUX register, see the enum <code>trgmux_device_t</code> defined in <SOC>.h.
<i>input</i>	The MUX select for peripheral trigger input
<i>trigger_src</i>	The trigger inputs for various peripherals. See the enum <code>trgmux_source_t</code> defined in <SOC>.h.

Return values

<i>kStatus_Success</i>	Configured successfully.
<i>kStatus_TRGMUX_Locked</i>	Configuration failed because the register is locked.

Chapter 28

WDOG32: 32-bit Watchdog Timer

28.1 Overview

The MCUXpresso SDK provides a peripheral driver for the WDOG32 module of MCUXpresso SDK devices.

28.2 Typical use case

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/wdog32

Data Structures

- struct [wdog32_work_mode_t](#)
Defines WDOG32 work mode. [More...](#)
- struct [wdog32_config_t](#)
Describes WDOG32 configuration structure. [More...](#)

Enumerations

- enum [wdog32_clock_source_t](#) {
 kWDOG32_ClockSource0 = 0U,
 kWDOG32_ClockSource1 = 1U,
 kWDOG32_ClockSource2 = 2U,
 kWDOG32_ClockSource3 = 3U }
Describes WDOG32 clock source.
- enum [wdog32_clock_prescaler_t](#) {
 kWDOG32_ClockPrescalerDivide1 = 0x0U,
 kWDOG32_ClockPrescalerDivide256 = 0x1U }
Describes the selection of the clock prescaler.
- enum [wdog32_test_mode_t](#) {
 kWDOG32_TestModeDisabled = 0U,
 kWDOG32_UserModeEnabled = 1U,
 kWDOG32_LowByteTest = 2U,
 kWDOG32_HighByteTest = 3U }
Describes WDOG32 test mode.
- enum [_wdog32_interrupt_enable_t](#) { kWDOG32_InterruptEnable = WDOG_CS_INT_MASK }
WDOG32 interrupt configuration structure.
- enum [_wdog32_status_flags_t](#) {
 kWDOG32_RunningFlag = WDOG_CS_EN_MASK,
 kWDOG32_InterruptFlag = WDOG_CS_FLG_MASK }
WDOG32 status flags.

Unlock sequence

- #define **WDOG_FIRST_WORD_OF_UNLOCK** (WDOG_UPDATE_KEY & 0xFFFFU)
First word of unlock sequence.
- #define **WDOG_SECOND_WORD_OF_UNLOCK** ((WDOG_UPDATE_KEY >> 16U) & 0xFF-FFU)
Second word of unlock sequence.

Refresh sequence

- #define **WDOG_FIRST_WORD_OF_REFRESH** (WDOG_REFRESH_KEY & 0xFFFFU)
First word of refresh sequence.
- #define **WDOG_SECOND_WORD_OF_REFRESH** ((WDOG_REFRESH_KEY >> 16U) & 0xFF-FFU)
Second word of refresh sequence.

Driver version

- #define **FSL_WDOG32_DRIVER_VERSION** (MAKE_VERSION(2, 0, 4))
WDOG32 driver version.

WDOG32 Initialization and De-initialization

- void **WDOG32_GetDefaultConfig** (wdog32_config_t *config)
Initializes the WDOG32 configuration structure.
- void **WDOG32_Init** (WDOG_Type *base, const wdog32_config_t *config)
Initializes the WDOG32 module.
- void **WDOG32_Deinit** (WDOG_Type *base)
De-initializes the WDOG32 module.

WDOG32 functional Operation

- void **WDOG32_Unlock** (WDOG_Type *base)
Unlocks the WDOG32 register written.
- void **WDOG32_Enable** (WDOG_Type *base)
Enables the WDOG32 module.
- void **WDOG32_Disable** (WDOG_Type *base)
Disables the WDOG32 module.
- void **WDOG32_EnableInterrupts** (WDOG_Type *base, uint32_t mask)
Enables the WDOG32 interrupt.
- void **WDOG32_DisableInterrupts** (WDOG_Type *base, uint32_t mask)
Disables the WDOG32 interrupt.
- static uint32_t **WDOG32_GetStatusFlags** (WDOG_Type *base)
Gets the WDOG32 all status flags.
- void **WDOG32_ClearStatusFlags** (WDOG_Type *base, uint32_t mask)
Clears the WDOG32 flag.
- void **WDOG32_SetTimeoutValue** (WDOG_Type *base, uint16_t timeoutCount)
Sets the WDOG32 timeout value.
- void **WDOG32_SetWindowValue** (WDOG_Type *base, uint16_t windowValue)
Sets the WDOG32 window value.
- static void **WDOG32_Refresh** (WDOG_Type *base)

Refreshes the WDOG32 timer.

- static uint16_t [WDOG32_GetCounterValue](#) (WDOG_Type *base)
Gets the WDOG32 counter value.

28.3 Data Structure Documentation

28.3.1 struct wdog32_work_mode_t

Data Fields

- bool [enableWait](#)
Enables or disables WDOG32 in wait mode.
- bool [enableStop](#)
Enables or disables WDOG32 in stop mode.
- bool [enableDebug](#)
Enables or disables WDOG32 in debug mode.

28.3.2 struct wdog32_config_t

Data Fields

- bool [enableWdog32](#)
Enables or disables WDOG32.
- [wdog32_clock_source_t clockSource](#)
Clock source select.
- [wdog32_clock_prescaler_t prescaler](#)
Clock prescaler value.
- [wdog32_work_mode_t workMode](#)
Configures WDOG32 work mode in debug stop and wait mode.
- [wdog32_test_mode_t testMode](#)
Configures WDOG32 test mode.
- bool [enableUpdate](#)
Update write-once register enable.
- bool [enableInterrupt](#)
Enables or disables WDOG32 interrupt.
- bool [enableWindowMode](#)
Enables or disables WDOG32 window mode.
- uint16_t [windowValue](#)
Window value.
- uint16_t [timeoutValue](#)
Timeout value.

28.4 Macro Definition Documentation

28.4.1 #define FSL_WDOG32_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))

28.5 Enumeration Type Documentation

28.5.1 enum wdog32_clock_source_t

Enumerator

- kWDOG32_ClockSource0* Clock source 0.
- kWDOG32_ClockSource1* Clock source 1.
- kWDOG32_ClockSource2* Clock source 2.
- kWDOG32_ClockSource3* Clock source 3.

28.5.2 enum wdog32_clock_prescaler_t

Enumerator

- kWDOG32_ClockPrescalerDivide1* Divided by 1.
- kWDOG32_ClockPrescalerDivide256* Divided by 256.

28.5.3 enum wdog32_test_mode_t

Enumerator

- kWDOG32_TestModeDisabled* Test Mode disabled.
- kWDOG32_UserModeEnabled* User Mode enabled.
- kWDOG32_LowByteTest* Test Mode enabled, only low byte is used.
- kWDOG32_HighByteTest* Test Mode enabled, only high byte is used.

28.5.4 enum _wdog32_interrupt_enable_t

This structure contains the settings for all of the WDOG32 interrupt configurations.

Enumerator

- kWDOG32_InterruptEnable* Interrupt is generated before forcing a reset.

28.5.5 enum _wdog32_status_flags_t

This structure contains the WDOG32 status flags for use in the WDOG32 functions.

Enumerator

- kWDOG32_RunningFlag* Running flag, set when WDOG32 is enabled.
- kWDOG32_InterruptFlag* Interrupt flag, set when interrupt occurs.

28.6 Function Documentation

28.6.1 void WDOG32_GetDefaultConfig (wdog32_config_t * *config*)

This function initializes the WDOG32 configuration structure to default values. The default values are:

```
* wdog32Config->enableWdog32 = true;
* wdog32Config->clockSource = kWDOG32_ClockSource1;
* wdog32Config->prescaler = kWDOG32_ClockPrescalerDivide1;
* wdog32Config->workMode.enableWait = true;
* wdog32Config->workMode.enableStop = false;
* wdog32Config->workMode.enableDebug = false;
* wdog32Config->testMode = kWDOG32_TestModeDisabled;
* wdog32Config->enableUpdate = true;
* wdog32Config->enableInterrupt = false;
* wdog32Config->enableWindowMode = false;
* wdog32Config->>windowValue = 0U;
* wdog32Config->timeoutValue = 0xFFFFU;
*
```

Parameters

<i>config</i>	Pointer to the WDOG32 configuration structure.
---------------	--

See Also

[wdog32_config_t](#)

28.6.2 void WDOG32_Init (WDOG_Type * *base*, const wdog32_config_t * *config*)

This function initializes the WDOG32. To reconfigure the WDOG32 without forcing a reset first, enableUpdate must be set to true in the configuration.

Example:

```
* wdog32_config_t config;
* WDOG32_GetDefaultConfig(&config);
* config.timeoutValue = 0x7ffU;
* config.enableUpdate = true;
* WDOG32_Init(wdog_base,&config);
*
```

Parameters

<i>base</i>	WDOG32 peripheral base address.
<i>config</i>	The configuration of the WDOG32.

28.6.3 void WDOG32_Deinit (WDOG_Type * *base*)

This function shuts down the WDOG32. Ensure that the WDOG_CS.UPDATE is 1, which means that the register update is enabled.

Parameters

<i>base</i>	WDOG32 peripheral base address.
-------------	---------------------------------

28.6.4 void WDOG32_Unlock (WDOG_Type * *base*)

This function unlocks the WDOG32 register written.

Before starting the unlock sequence and following the configuration, disable the global interrupts. Otherwise, an interrupt could effectively invalidate the unlock sequence and the WCT may expire. After the configuration finishes, re-enable the global interrupts.

Parameters

<i>base</i>	WDOG32 peripheral base address
-------------	--------------------------------

28.6.5 void WDOG32_Enable (WDOG_Type * *base*)

This function writes a value into the WDOG_CS register to enable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling [WDOG32_Init](#) to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

<i>base</i>	WDOG32 peripheral base address.
-------------	---------------------------------

28.6.6 void WDOG32_Disable (WDOG_Type * *base*)

This function writes a value into the WDOG_CS register to disable the WDOG32. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling [WDOG32_Init](#) to

do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

<i>base</i>	WDOG32 peripheral base address
-------------	--------------------------------

28.6.7 void WDOG32_EnableInterrupts (**WDOG_Type** * *base*, **uint32_t** *mask*)

This function writes a value into the WDOG_CS register to enable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling [WDOG32_Init](#) to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

<i>base</i>	WDOG32 peripheral base address.
<i>mask</i>	The interrupts to enable. The parameter can be a combination of the following source if defined: <ul style="list-style-type: none">• kWDOG32_InterruptEnable

28.6.8 void WDOG32_DisableInterrupts (**WDOG_Type** * *base*, **uint32_t** *mask*)

This function writes a value into the WDOG_CS register to disable the WDOG32 interrupt. The WDOG_CS register is a write-once register. Please check the enableUpdate is set to true for calling [WDOG32_Init](#) to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

<i>base</i>	WDOG32 peripheral base address.
<i>mask</i>	The interrupts to disabled. The parameter can be a combination of the following source if defined: <ul style="list-style-type: none">• kWDOG32_InterruptEnable

28.6.9 static uint32_t WDOG32_GetStatusFlags (**WDOG_Type** * *base*) [inline], [static]

This function gets all status flags.

Example to get the running flag:

```
*     uint32_t status;
*     status = WDOG32_GetStatusFlags(wdog_base) &
*             kWDOG32_RunningFlag;
*
```

Parameters

<i>base</i>	WDOG32 peripheral base address
-------------	--------------------------------

Returns

State of the status flag: asserted (true) or not-asserted (false).

See Also

[_wdog32_status_flags_t](#)

- true: related status flag has been set.
- false: related status flag is not set.

28.6.10 void WDOG32_ClearStatusFlags (**WDOG_Type * *base*, **uint32_t** *mask*)**

This function clears the WDOG32 status flag.

Example to clear an interrupt flag:

```
*     WDOG32_ClearStatusFlags(wdog_base,
*                           kWDOG32_InterruptFlag);
*
```

Parameters

<i>base</i>	WDOG32 peripheral base address.
<i>mask</i>	The status flags to clear. The parameter can be any combination of the following values: <ul style="list-style-type: none"> • kWDOG32_InterruptFlag

28.6.11 void WDOG32_SetTimeoutValue (**WDOG_Type * *base*, **uint16_t** *timeoutCount*)**

This function writes a timeout value into the WDOG_TOVAL register. The WDOG_TOVAL register is a write-once register. To ensure the reconfiguration fits the timing of WCT, unlock function will be called inline.

Parameters

<i>base</i>	WDOG32 peripheral base address
<i>timeoutCount</i>	WDOG32 timeout value, count of WDOG32 clock ticks.

28.6.12 void WDOG32_SetWindowValue (WDOG_Type * *base*, uint16_t *windowValue*)

This function writes a window value into the WDOG_WIN register. The WDOG_WIN register is a write-once register. Please check the enableUpdate is set to true for calling [WDOG32_Init](#) to do wdog initialize. Before call the re-configuration APIs, ensure that the WCT window is still open and this register has not been written in this WCT while the function is called.

Parameters

<i>base</i>	WDOG32 peripheral base address.
<i>windowValue</i>	WDOG32 window value.

28.6.13 static void WDOG32_Refresh (WDOG_Type * *base*) [inline], [static]

This function feeds the WDOG32. This function should be called before the Watchdog timer is in timeout. Otherwise, a reset is asserted.

Parameters

<i>base</i>	WDOG32 peripheral base address
-------------	--------------------------------

28.6.14 static uint16_t WDOG32_GetCounterValue (WDOG_Type * *base*) [inline], [static]

This function gets the WDOG32 counter value.

Parameters

<i>base</i>	WDOG32 peripheral base address.
-------------	---------------------------------

Returns

Current WDOG32 counter value.

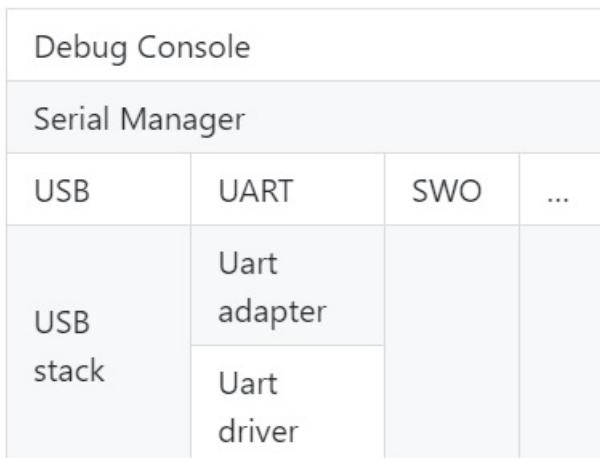
Chapter 29

Debug Console

29.1 Overview

This chapter describes the programming interface of the debug console driver.

The debug console enables debug log messages to be output via the specified peripheral with frequency of the peripheral source clock and base address at the specified baud rate. Additionally, it provides input and output functions to scan and print formatted data. The below picture shows the layout of debug console.



Debug console overview

29.2 Function groups

29.2.1 Initialization

To initialize the debug console, call the [DbgConsole_Init\(\)](#) function with these parameters. This function automatically enables the module and the clock.

```
status_t DbgConsole_Init(uint8_t instance, uint32_t baudRate,  
                         serial_port_type_t device, uint32_t clkSrcFreq);
```

Select the supported debug console hardware device type, such as

```
typedef enum _serial_port_type  
{  
    kSerialPort_Uart = 1U,  
    kSerialPort_UsbCdc,  
    kSerialPort_Swo,  
} serial_port_type_t;
```

After the initialization is successful, stdout and stdin are connected to the selected peripheral. This example shows how to call the [DbgConsole_Init\(\)](#) given the user configuration structure.

```
DbgConsole_Init(BOARD_DEBUG_UART_INSTANCE, BOARD_DEBUG_UART_BAUDRATE, BOARD_DEBUG_UART_TYPE,
                 BOARD_DEBUG_UART_CLK_FREQ);
```

29.2.2 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

- 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 *
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(char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the MCUXpresso SDK printf/scanf.

```
#if SDK_DEBUGCONSOLE == DEBUGCONSOLE_DISABLE /* Disable debug console */
#define PRINTF
#define SCANF
#define PUTCHAR
#define GETCHAR
#elif SDK_DEBUGCONSOLE == DEBUGCONSOLE_REDIRECT_TO_SDK /* Select printf, scanf, putchar, getchar of SDK
```

```

version. */
#define PRINTF DbgConsole_Printf
#define SCANF DbgConsole_Scanf
#define PUTCHAR DbgConsole_Putchar
#define GETCHAR DbgConsole_Getchar
#elif SDK_DEBUGCONSOLE == DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN /* Select printf, scanf, putchar, getchar of
toolchain. */
#define PRINTF printf
#define SCANF scanf
#define PUTCHAR putchar
#define GETCHAR getchar
#endif /* SDK_DEBUGCONSOLE */

```

29.2.3 SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_UART

There are two macros `SDK_DEBUGCONSOLE` and `SDK_DEBUGCONSOLE_UART` added to configure `PRINTF` and low level output peripheral.

- The macro `SDK_DEBUGCONSOLE` is used for frontend. Whether debug console redirect to toolchain or SDK or disabled, it decides which is the frontend of the debug console, Tool chain or SDK. The function can be set by the macro `SDK_DEBUGCONSOLE`.
- The macro `SDK_DEBUGCONSOLE_UART` is used for backend. It is used to decide whether provide low level IO implementation to toolchain printf and scanf. For example, within MCUXpresso, if the macro `SDK_DEBUGCONSOLE_UART` is defined, `_sys_write` and `_sys_read` will be used when `_REDLIB_` is defined; `_write` and `_read` will be used in other cases. The macro does not specifically refer to the peripheral "UART". It refers to the external peripheral similar to UART, like as USB CDC, UART, SWO, etc. So if the macro `SDK_DEBUGCONSOLE_UART` is not defined when tool-chain printf is calling, the semihosting will be used.

The following matrix show the effects of `SDK_DEBUGCONSOLE` and `SDK_DEBUGCONSOLE_UART` on `PRINTF` and `printf`. The green mark is the default setting of the debug console.

<code>SDK_DEBUGCONSOLE</code>	<code>SDK_DEBUGCONSOLE_UART</code>	<code>PRINTF</code>	<code>printf</code>
<code>DEBUGCONSOLE_- REDIRECT_TO_SDK</code>	defined	Low level peripheral*	Low level peripheral
<code>DEBUGCONSOLE_- REDIRECT_TO_SDK</code>	undefined	Low level peripheral*	semihost
<code>DEBUGCONSOLE_- REDIRECT_TO_TO- OLCHAIN</code>	defined	Low level peripheral*	Low level peripheral
<code>DEBUGCONSOLE_- REDIRECT_TO_TO- OLCHAIN</code>	undefined	semihost	semihost
<code>DEBUGCONSOLE_- DISABLE</code>	defined	No output	Low level peripheral
<code>DEBUGCONSOLE_- DISABLE</code>	undefined	No output	semihost

- * the **low level peripheral** could be USB CDC, UART, or SWO, and so on.

29.3 Typical use case

Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

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\n\rTime: %u ticks %2.5f milliseconds\n\rDONE\n\r", "1 day", 86400, 86.4);
```

Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

Print out failure messages using MCUXpresso SDK __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

Macros

- #define **DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN** 0U
Definition select redirect toolchain printf, scanf to uart or not.
- #define **DEBUGCONSOLE_REDIRECT_TO_SDK** 1U
Select SDK version printf, scanf.
- #define **DEBUGCONSOLE_DISABLE** 2U
Disable debugconsole function.
- #define **SDK_DEBUGCONSOLE DEBUGCONSOLE_REDIRECT_TO_SDK**
Definition to select sdk or toolchain printf, scanf.
- #define **PRINTF DbgConsole_Printf**
Definition to select redirect toolchain printf, scanf to uart or not.

Typedefs

- typedef void(* **printfCb**)(char *buf, int32_t *indicator, char val, int len)
A function pointer which is used when format printf log.

Functions

- int **StrFormatPrint** (const char *fmt, va_list ap, char *buf, **printfCb** cb)
This function outputs its parameters according to a formatted string.
- int **StrFormatScanf** (const char *line_ptr, char *format, va_list args_ptr)
Converts an input line of ASCII characters based upon a provided string format.

Variables

- **serial_handle_t g_serialHandle**
serial manager handle

Initialization

- **status_t DbgConsole_Init** (uint8_t instance, uint32_t baudRate, **serial_port_type_t** device, uint32_t clkSrcFreq)
Initializes the peripheral used for debug messages.
- **status_t DbgConsole_Deinit** (void)
De-initializes the peripheral used for debug messages.
- **status_t DbgConsole_EnterLowpower** (void)
Prepares to enter low power consumption.
- **status_t DbgConsole_ExitLowpower** (void)
Restores from low power consumption.
- int **DbgConsole_Printf** (const char *fmt_s,...)
Writes formatted output to the standard output stream.
- int **DbgConsole_Vprintf** (const char *fmt_s, va_list formatStringArg)
Writes formatted output to the standard output stream.
- int **DbgConsole_Putchar** (int ch)
Writes a character to stdout.

- int [DbgConsole_Scanf](#) (char *fmt_s,...)
Reads formatted data from the standard input stream.
- int [DbgConsole_Getchar](#) (void)
Reads a character from standard input.
- int [DbgConsole_BlockingPrintf](#) (const char *fmt_s,...)
Writes formatted output to the standard output stream with the blocking mode.
- int [DbgConsole_BlockingVprintf](#) (const char *fmt_s, va_list formatStringArg)
Writes formatted output to the standard output stream with the blocking mode.
- status_t [DbgConsole_Flush](#) (void)
Debug console flush.

29.4 Macro Definition Documentation

29.4.1 #define DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN 0U

Select toolchain printf and scanf.

29.4.2 #define DEBUGCONSOLE_REDIRECT_TO_SDK 1U

29.4.3 #define DEBUGCONSOLE_DISABLE 2U

29.4.4 #define SDK_DEBUGCONSOLE DEBUGCONSOLE_REDIRECT_TO_SDK

The macro only support to be redefined in project setting.

29.4.5 #define PRINTF DbgConsole_Printf

if SDK_DEBUGCONSOLE defined to 0,it represents select toolchain printf, scanf. if SDK_DEBUGCONSOLE defined to 1,it represents select SDK version printf, scanf. if SDK_DEBUGCONSOLE defined to 2,it represents disable debugconsole function.

29.5 Function Documentation

29.5.1 status_t DbgConsole_Init (uint8_t instance, uint32_t baudRate, serial_port_type_t device, uint32_t clkSrcFreq)

Call this function to enable debug log messages to be output via the specified peripheral initialized by the serial manager module. After this function has returned, stdout and stdin are connected to the selected peripheral.

Parameters

<i>instance</i>	The instance of the module. If the device is kSerialPort_Uart, the instance is UART peripheral instance. The UART hardware peripheral type is determined by UART adapter. For example, if the instance is 1, if the lpuart_adapter.c is added to the current project, the UART peripheral is LPUART1. If the uart_adapter.c is added to the current project, the UART peripheral is UART1.
<i>baudRate</i>	The desired baud rate in bits per second.
<i>device</i>	Low level device type for the debug console, can be one of the following. <ul style="list-style-type: none"> • kSerialPort_Uart, • kSerialPort_UsbCdc
<i>clkSrcFreq</i>	Frequency of peripheral source clock.

Returns

Indicates whether initialization was successful or not.

Return values

<i>kStatus_Success</i>	Execution successfully
------------------------	------------------------

29.5.2 status_t DbgConsole_Deinit (void)

Call this function to disable debug log messages to be output via the specified peripheral initialized by the serial manager module.

Returns

Indicates whether de-initialization was successful or not.

29.5.3 status_t DbgConsole_EnterLowpower (void)

This function is used to prepare to enter low power consumption.

Returns

Indicates whether de-initialization was successful or not.

29.5.4 status_t DbgConsole_ExitLowpower (void)

This function is used to restore from low power consumption.

Returns

Indicates whether de-initialization was successful or not.

29.5.5 int DbgConsole_Printf (const char * *fmt_s*, ...)

Call this function to write a formatted output to the standard output stream.

Parameters

<i>fmt_s</i>	Format control string.
--------------	------------------------

Returns

Returns the number of characters printed or a negative value if an error occurs.

29.5.6 int DbgConsole_Vprintf (const char * *fmt_s*, va_list *formatStringArg*)

Call this function to write a formatted output to the standard output stream.

Parameters

<i>fmt_s</i>	Format control string.
<i>formatString-Arg</i>	Format arguments.

Returns

Returns the number of characters printed or a negative value if an error occurs.

29.5.7 int DbgConsole_Putchar (int *ch*)

Call this function to write a character to stdout.

Parameters

<i>ch</i>	Character to be written.
-----------	--------------------------

Returns

Returns the character written.

29.5.8 int DbgConsole_Scanf (char * *fmt_s*, ...)

Call this function to read formatted data from the standard input stream.

Note

Due the limitation in the BM OSA environment (CPU is blocked in the function, other tasks will not be scheduled), the function cannot be used when the DEBUG_CONSOLE_TRANSFER_NON_B-LOCKING is set in the BM OSA environment. And an error is returned when the function called in this case. The suggestion is that polling the non-blocking function DbgConsole_TryGetchar to get the input char.

Parameters

<i>fmt_s</i>	Format control string.
--------------	------------------------

Returns

Returns the number of fields successfully converted and assigned.

29.5.9 int DbgConsole_Getchar (void)

Call this function to read a character from standard input.

Note

Due the limitation in the BM OSA environment (CPU is blocked in the function, other tasks will not be scheduled), the function cannot be used when the DEBUG_CONSOLE_TRANSFER_NON_B-LOCKING is set in the BM OSA environment. And an error is returned when the function called in this case. The suggestion is that polling the non-blocking function DbgConsole_TryGetchar to get the input char.

Returns

Returns the character read.

29.5.10 int DbgConsole_BlockingPrintf (const char * *fmt_s*, ...)

Call this function to write a formatted output to the standard output stream with the blocking mode. The function will send data with blocking mode no matter the DEBUG_CONSOLE_TRANSFER_NON_BLOCKING set or not. The function could be used in system ISR mode with DEBUG_CONSOLE_TRANSFER_NON_BLOCKING set.

Parameters

<i>fmt_s</i>	Format control string.
--------------	------------------------

Returns

Returns the number of characters printed or a negative value if an error occurs.

29.5.11 int DbgConsole_BlockingVprintf (const char * *fmt_s*, va_list *formatStringArg*)

Call this function to write a formatted output to the standard output stream with the blocking mode. The function will send data with blocking mode no matter the DEBUG_CONSOLE_TRANSFER_NON_BLOCKING set or not. The function could be used in system ISR mode with DEBUG_CONSOLE_TRANSFER_NON_BLOCKING set.

Parameters

<i>fmt_s</i>	Format control string.
<i>formatString-Arg</i>	Format arguments.

Returns

Returns the number of characters printed or a negative value if an error occurs.

29.5.12 status_t DbgConsole_Flush (void)

Call this function to wait the tx buffer empty. If interrupt transfer is using, make sure the global IRQ is enable before call this function This function should be called when 1, before enter power down mode 2, log is required to print to terminal immediately

Returns

Indicates whether wait idle was successful or not.

29.5.13 int StrFormatPrintf (const char * *fmt*, va_list *ap*, char * *buf*, printfCb *cb*)

Note

I/O is performed by calling given function pointer using following (*func_ptr)(c);

Parameters

in	<i>fmt</i>	Format string for printf.
in	<i>ap</i>	Arguments to printf.
in	<i>buf</i>	pointer to the buffer
	<i>cb</i>	print callbk function pointer

Returns

Number of characters to be print

29.5.14 int StrFormatScanf (const char * *line_ptr*, char * *format*, va_list *args_ptr*)

Parameters

in	<i>line_ptr</i>	The input line of ASCII data.
in	<i>format</i>	Format first points to the format string.
in	<i>args_ptr</i>	The list of parameters.

Returns

Number of input items converted and assigned.

Return values

<i>IO_EOF</i>	When line_ptr is empty string "".
---------------	-----------------------------------

29.6 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.

29.6.1 Guide Semihosting for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging, if you want use PRINTF with semihosting, please make sure the `SDK_DEBUGCONSOLE` is `DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN`.

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. Choose tab "General Options" -> "Library Configurations", select Semihosted, select Via semihosting. Please Make sure the `SDK_DEBUGCONSOLE_UART` is not defined in project settings.
4. Start the project by choosing Project>Download and Debug.
5. Choose View>Terminal I/O to display the output from the I/O operations.

29.6.2 Guide Semihosting for Keil µVision

NOTE: Semihosting is not support by MDK-ARM, use the retargeting functionality of MDK-ARM instead.

29.6.3 Guide Semihosting for MCUXpresso IDE

Step 1: Setting up the environment

1. To set debugger options, choose Project>Properties. select the setting category.
2. Select Tool Settings, unfold MCU C Compile.
3. Select Preprocessor item.
4. Set SDK_DEBUGCONSOLE=0, if set SDK_DEBUGCONSOLE=1, the log will be redirect to the UART.

Step 2: Building the project

1. Compile and link the project.

Step 3: Starting semihosting

1. Download and debug the project.
2. When the project runs successfully, the result can be seen in the Console window.

Semihosting can also be selected through the "Quick settings" menu in the left bottom window, Quick settings->SDK Debug Console->Semihost console.

29.6.4 Guide Semihosting for ARMGCC

Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
 - "Host Name (or IP address)" : localhost
 - "Port" :2333
 - "Connection type" : Telet.
 - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

Add to "CMakeLists.txt"

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__stack_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__stack_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --defsym=__heap_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} --defsym=__heap_size__=0x2000")
```

Step 2: Building the project

1. Change "CMakeLists.txt":

```
Change "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=nano.specs")"
to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=rdimon.specs")"
```

Replace paragraph

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-common")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffunction-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fdata-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -ffreestanding")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fno-builtin")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mthumb")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -mapcs")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --gc-sections")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -static")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -z")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -Xlinker")
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} muldefs")
```

To

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --specs=rdimon.specs ")
```

Remove

```
target_link_libraries(semihosting_ARMGCC.elf debug nosys)
```

2. Run "build_debug.bat" to build project

Step 3: Starting semihosting

1. 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 = MK64FN1M0xxxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

2. After the setting, press "enter". The PuTTY window now shows the printf() output.

Chapter 30

Notification Framework

30.1 Overview

This section describes the programming interface of the Notifier driver.

30.2 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)
```

```

{
    ...
    ...
    ...
}

...
...
...
...
...
...
// Main function.
int main(void)
{
    // Define a notifier handle.
    notifier_handle_t powerModeHandle;

    // Callback configuration.
    user_callback_data_t callbackData0;

    notifier_callback_config_t callbackCfg0 = {callback0,
        kNOTIFIER_CallbackBeforeAfter,
        (void *)&callbackData0};

    notifier_callback_config_t callbacks[] = {callbackCfg0};

    // Power mode configurations.
    power_user_config_t vlprConfig;
    power_user_config_t stopConfig;

    notifier_user_config_t *powerConfigs[] = {&vlprConfig, &stopConfig};

    // Definition of a transition to and out the power modes.
    vlprConfig.mode = kAPP_PowerModeVlpr;
    vlprConfig.enableLowPowerWakeUpOnInterrupt = false;

    stopConfig = vlprConfig;
    stopConfig.mode = kAPP_PowerModeStop;

    // Create Notifier handle.
    NOTIFIER_CreateHandle(&powerModeHandle, powerConfigs, 2U, callbacks, 1U,
        APP_PowerModeSwitch, NULL);
    ...

    ...
    // Power mode switch.
    NOTIFIER_switchConfig(&powerModeHandle, targetConfigIndex,
        kNOTIFIER_PolicyAgreement);
}

```

Data Structures

- struct [notifier_notification_block_t](#)
notification block passed to the registered callback function. [More...](#)
- struct [notifier_callback_config_t](#)
Callback configuration structure. [More...](#)
- struct [notifier_handle_t](#)
Notifier handle structure. [More...](#)

Typedefs

- [typedef void notifier_user_config_t](#)
Notifier user configuration type.
- [typedef status_t\(* notifier_user_function_t \)\(notifier_user_config_t *targetConfig, void *userData\)](#)

- *Notifier user function prototype Use this function to execute specific operations in configuration switch.*
typedef status_t(* notifier_callback_t)(notifier_notification_block_t *notify, void *data)
Callback prototype.

Enumerations

- **enum _notifier_status {**
kStatus_NOTIFIER_ErrorNotificationBefore,
kStatus_NOTIFIER_ErrorNotificationAfter }
Notifier error codes.
- **enum notifier_policy_t {**
kNOTIFIER_PolicyAgreement,
kNOTIFIER_PolicyForcible }
Notifier policies.
- **enum notifier_notification_type_t {**
kNOTIFIER_NotifyRecover = 0x00U,
kNOTIFIER_NotifyBefore = 0x01U,
kNOTIFIER_NotifyAfter = 0x02U }
Notification type.
- **enum notifier_callback_type_t {**
kNOTIFIER_CallbackBefore = 0x01U,
kNOTIFIER_CallbackAfter = 0x02U,
kNOTIFIER_CallbackBeforeAfter = 0x03U }
The callback type, which indicates kinds of notification the callback handles.

Functions

- **status_t NOTIFIER_CreateHandle (notifier_handle_t *notifierHandle, notifier_user_config_t **configs, uint8_t configsNumber, notifier_callback_config_t *callbacks, uint8_t callbacksNumber, notifier_user_function_t userFunction, void *userData)**
Creates a Notifier handle.
- **status_t NOTIFIER_SwitchConfig (notifier_handle_t *notifierHandle, uint8_t configIndex, notifier_policy_t policy)**
Switches the configuration according to a pre-defined structure.
- **uint8_t NOTIFIER_GetErrorCallbackIndex (notifier_handle_t *notifierHandle)**
This function returns the last failed notification callback.

30.3 Data Structure Documentation

30.3.1 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.

Field Documentation

- (1) **notifier_user_config_t* notifier_notification_block_t::targetConfig**
- (2) **notifier_policy_t notifier_notification_block_t::policy**
- (3) **notifier_notification_type_t notifier_notification_block_t::notifyType**

30.3.2 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.

Field Documentation

- (1) **notifier_callback_t notifier_callback_config_t::callback**
- (2) **notifier_callback_type_t notifier_callback_config_t::callbackType**
- (3) **void* notifier_callback_config_t::callbackData**

30.3.3 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.

Field Documentation

- (1) `notifier_user_config_t** notifier_handle_t::configsTable`
- (2) `uint8_t notifier_handle_t::configsNumber`
- (3) `notifier_callback_config_t* notifier_handle_t::callbacksTable`
- (4) `uint8_t notifier_handle_t::callbacksNumber`
- (5) `uint8_t notifier_handle_t::errorCallbackIndex`
- (6) `uint8_t notifier_handle_t::currentConfigIndex`
- (7) `notifier_user_function_t notifier_handle_t::userFunction`
- (8) `void* notifier_handle_t::userData`

30.4 Typedef Documentation

30.4.1 `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.

30.4.2 `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

<i>targetConfig</i>	target Configuration.
<i>userData</i>	Refers to other specific data passed to user function.

Returns

An error code or kStatus_Success.

30.4.3 **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.

30.5 Enumeration Type Documentation

30.5.1 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.

30.5.2 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.

30.5.3 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.

30.5.4 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.

30.6 Function Documentation

30.6.1 **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.

30.6.2 **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

<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.

30.6.3 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 31

Shell

31.1 Overview

This section describes the programming interface of the Shell middleware.

Shell controls MCUs by commands via the specified communication peripheral based on the debug console driver.

31.2 Function groups

31.2.1 Initialization

To initialize the Shell middleware, call the `SHELL_Init()` function with these parameters. This function automatically enables the middleware.

```
shell_status_t SHELL_Init(shell_handle_t shellHandle,  
    serial_handle_t serialHandle, 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(s_shellHandle, s_serialHandle, "Test@SHELL>");
```

31.2.2 Advanced Feature

- Support to get a character from standard input devices.

```
static shell_status_t SHELL_GetChar(shell_context_handle_t *shellContextHandle, uint8_t *ch);
```

Commands	Description
help	List all the registered commands.
exit	Exit program.

31.2.3 Shell Operation

```
SHELL_Init(s_shellHandle, s_serialHandle, "Test@SHELL>");  
SHELL_Task(s_shellHandle);
```

Data Structures

- struct `shell_command_t`
User command data configuration structure. More...

Macros

- #define `SHELL_NON_BLOCKING_MODE SERIAL_MANAGER_NON_BLOCKING_MODE`
Whether use non-blocking mode.
- #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_HISTORY_COUNT` (3U)
Macro to set maximum count of history commands.
- #define `SHELL_IGNORE_PARAMETER_COUNT` (0xFF)
Macro to bypass arguments check.
- #define `SHELL_HANDLE_SIZE`
The handle size of the shell module.
- #define `SHELL_USE_COMMON_TASK` (0U)
Macro to determine whether use common task.
- #define `SHELL_TASK_PRIORITY` (2U)
Macro to set shell task priority.
- #define `SHELL_TASK_STACK_SIZE` (1000U)
Macro to set shell task stack size.
- #define `SHELL_HANDLE_DEFINE`(name) uint32_t name[((`SHELL_HANDLE_SIZE` + sizeof(uint32_t) - 1U) / sizeof(uint32_t))]
Defines the shell handle.
- #define `SHELL_COMMAND_DEFINE`(command, descriptor, callback, paramInt)
Defines the shell command structure.
- #define `SHELL_COMMAND`(command) &g_shellCommand##command
Gets the shell command pointer.

Typedefs

- typedef void * `shell_handle_t`
The handle of the shell module.
- typedef `shell_status_t`(* `cmd_function_t`)(`shell_handle_t` shellHandle, int32_t argc, char **argv)
User command function prototype.

Enumerations

- enum `shell_status_t` {

`kStatus_SHELL_Success` = kStatus_Success,

`kStatus_SHELL_Error` = MAKE_STATUS(kStatusGroup_SHELL, 1),

`kStatus_SHELL_OpenWriteHandleFailed` = MAKE_STATUS(kStatusGroup_SHELL, 2),

`kStatus_SHELL_OpenReadHandleFailed` = MAKE_STATUS(kStatusGroup_SHELL, 3) }

Shell status.

Shell functional operation

- `shell_status_t SHELL_Init (shell_handle_t shellHandle, serial_handle_t serialHandle, char *prompt)`
Initializes the shell module.
- `shell_status_t SHELL_RegisterCommand (shell_handle_t shellHandle, shell_command_t *shellCommand)`
Registers the shell command.
- `shell_status_t SHELL_UnregisterCommand (shell_command_t *shellCommand)`
Unregisters the shell command.
- `shell_status_t SHELL_Write (shell_handle_t shellHandle, const char *buffer, uint32_t length)`
Sends data to the shell output stream.
- `int SHELL_Printf (shell_handle_t shellHandle, const char *formatString,...)`
Writes formatted output to the shell output stream.
- `shell_status_t SHELL_WriteSynchronization (shell_handle_t shellHandle, const char *buffer, uint32_t length)`
Sends data to the shell output stream with OS synchronization.
- `int SHELL_PrintfSynchronization (shell_handle_t shellHandle, const char *formatString,...)`
Writes formatted output to the shell output stream with OS synchronization.
- `void SHELL_ChangePrompt (shell_handle_t shellHandle, char *prompt)`
Change shell prompt.
- `void SHELL_PrintPrompt (shell_handle_t shellHandle)`
Print shell prompt.
- `void SHELL_Task (shell_handle_t shellHandle)`
The task function for Shell.
- `static bool SHELL_checkRunningInIsr (void)`
Check if code is running in ISR.

31.3 Data Structure Documentation

31.3.1 struct shell_command_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.
- `list_element_t link`
link of the element

Field Documentation

(1) `const char* shell_command_t::pcCommand`

For example "help". It must be all lower case.

(2) `char* shell_command_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".

(3) `const cmd_function_t shell_command_t::pFuncCallBack`**(4) `uint8_t shell_command_t::cExpectedNumberOfParameters`****31.4 Macro Definition Documentation****31.4.1 `#define SHELL_NON_BLOCKING_MODE SERIAL_MANAGER_NON_BLOCKING_MODE`****31.4.2 `#define SHELL_AUTO_COMPLETE (1U)`****31.4.3 `#define SHELL_BUFFER_SIZE (64U)`****31.4.4 `#define SHELL_MAX_ARGS (8U)`****31.4.5 `#define SHELL_HISTORY_COUNT (3U)`****31.4.6 `#define SHELL_HANDLE_SIZE`**

Value:

```
(160U + SHELL_HISTORY_COUNT * SHELL_BUFFER_SIZE +
    SHELL_BUFFER_SIZE + SERIAL_MANAGER_READ_HANDLE_SIZE + \
    SERIAL_MANAGER_WRITE_HANDLE_SIZE)
```

It is the sum of the SHELL_HISTORY_COUNT * SHELL_BUFFER_SIZE + SHELL_BUFFER_SIZE + SERIAL_MANAGER_READ_HANDLE_SIZE + SERIAL_MANAGER_WRITE_HANDLE_SIZE

31.4.7 `#define SHELL_USE_COMMON_TASK (0U)`**31.4.8 `#define SHELL_TASK_PRIORITY (2U)`****31.4.9 `#define SHELL_TASK_STACK_SIZE (1000U)`**

31.4.10 #define SHELL_HANDLE_DEFINE(*name*) uint32_t *name*[(**SHELL_HANDLE_SIZE** + sizeof(uint32_t) - 1U) / sizeof(uint32_t)]

This macro is used to define a 4 byte aligned shell handle. Then use "(shell_handle_t)*name*" to get the shell handle.

The macro should be global and could be optional. You could also define shell handle by yourself.

This is an example,

```
* SHELL_HANDLE_DEFINE(shellHandle);
*
```

Parameters

<i>name</i>	The name string of the shell handle.
-------------	--------------------------------------

31.4.11 #define SHELL_COMMAND_DEFINE(*command*, *descriptor*, *callback*, *paramCount*)

Value:

```
\shell_command_t g_shellCommand##command = {
    (#command), (descriptor), (callback), (paramCount), {0},      \
}
```

This macro is used to define the shell command structure [shell_command_t](#). And then uses the macro SHELL_COMMAND to get the command structure pointer. The macro should not be used in any function.

This is a example,

```
* SHELL_COMMAND_DEFINE(exit, "\r\n\"exit\": Exit program\r\n", SHELL_ExitCommand, 0);
* SHELL_RegisterCommand(s_shellHandle, SHELL_COMMAND(exit));
*
```

Parameters

<i>command</i>	The command string of the command. The double quotes do not need. Such as exit for "exit", help for "Help", read for "read".
----------------	--

<i>descriptor</i>	The description of the command is used for showing the command usage when "help" is typing.
<i>callback</i>	The callback of the command is used to handle the command line when the input command is matched.
<i>paramCount</i>	The max parameter count of the current command.

31.4.12 #define SHELL_COMMAND(*command*) &g_shellCommand##*command*

This macro is used to get the shell command pointer. The macro should not be used before the macro SHELL_COMMAND_DEFINE is used.

Parameters

<i>command</i>	The command string of the command. The double quotes do not need. Such as exit for "exit", help for "Help", read for "read".
----------------	--

31.5 Typedef Documentation

31.5.1 typedef shell_status_t(* cmd_function_t)(shell_handle_t shellHandle, int32_t argc, char **argv)

31.6 Enumeration Type Documentation

31.6.1 enum shell_status_t

Enumerator

kStatus_SHELL_Success Success.

kStatus_SHELL_Error Failed.

kStatus_SHELL_OpenWriteHandleFailed Open write handle failed.

kStatus_SHELL_OpenReadHandleFailed Open read handle failed.

31.7 Function Documentation

31.7.1 shell_status_t SHELL_Init (shell_handle_t *shellHandle*, serial_handle_t *serialHandle*, 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 Shell and how to call the SHELL_Init function by passing in these parameters. This is an example.

```
* static SHELL_HANDLE_DEFINE(s_shellHandle);
* SHELL_Init((shell_handle_t)s_shellHandle,
*             (serial_handle_t)s_serialHandle, "Test@SHELL>");
*
```

Parameters

<i>shellHandle</i>	Pointer to point to a memory space of size SHELL_HANDLE_SIZE allocated by the caller. The handle should be 4 byte aligned, because unaligned access doesn't be supported on some devices. You can define the handle in the following two ways: SHELL_HANDLE_DEFINE(shellHandle) ; or <code>uint32_t shellHandle[((SHELL_HANDLE_SIZE + sizeof(uint32_t) - 1U) / sizeof(uint32_t))];</code>
<i>serialHandle</i>	The serial manager module handle pointer.
<i>prompt</i>	The string prompt pointer of Shell. Only the global variable can be passed.

Return values

<i>kStatus_SHELL_Success</i>	The shell initialization succeed.
<i>kStatus_SHELL_Error</i>	An error occurred when the shell is initialized.
<i>kStatus_SHELL_OpenWriteHandleFailed</i>	Open the write handle failed.
<i>kStatus_SHELL_OpenReadHandleFailed</i>	Open the read handle failed.

31.7.2 **shell_status_t SHELL_RegisterCommand (shell_handle_t *shellHandle*, shell_command_t * *shellCommand*)**

This function is used to register the shell command by using the command configuration `shell_command_config_t`. This is a example,

```
* SHELL_COMMAND_DEFINE(exit, "\r\n\"exit\": Exit program\r\n", SHELL_ExitCommand, 0);
* SHELL_RegisterCommand(s_shellHandle, SHELL_COMMAND(exit));
*
```

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>shellCommand</i>	The command element.

Return values

<i>kStatus_SHELL_Success</i>	Successfully register the command.
<i>kStatus_SHELL_Error</i>	An error occurred.

31.7.3 shell_status_t SHELL_UnregisterCommand (shell_command_t * *shellCommand*)

This function is used to unregister the shell command.

Parameters

<i>shellCommand</i>	The command element.
---------------------	----------------------

Return values

<i>kStatus_SHELL_Success</i>	Successfully unregister the command.
------------------------------	--------------------------------------

31.7.4 shell_status_t SHELL_Write (shell_handle_t *shellHandle*, const char * *buffer*, uint32_t *length*)

This function is used to send data to the shell output stream.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>buffer</i>	Start address of the data to write.
<i>length</i>	Length of the data to write.

Return values

<i>kStatus_SHELL_Success</i>	Successfully send data.
<i>kStatus_SHELL_Error</i>	An error occurred.

31.7.5 int SHELL_Printf (shell_handle_t *shellHandle*, const char * *formatString*, ...)

Call this function to write a formatted output to the shell output stream.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>formatString</i>	Format string.

Returns

Returns the number of characters printed or a negative value if an error occurs.

31.7.6 **shell_status_t SHELL_WriteSynchronization (shell_handle_t *shellHandle*, const char * *buffer*, uint32_t *length*)**

This function is used to send data to the shell output stream with OS synchronization, note the function could not be called in ISR.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>buffer</i>	Start address of the data to write.
<i>length</i>	Length of the data to write.

Return values

<i>kStatus_SHELL_Success</i>	Successfully send data.
<i>kStatus_SHELL_Error</i>	An error occurred.

31.7.7 **int SHELL_PrintfSynchronization (shell_handle_t *shellHandle*, const char * *formatString*, ...)**

Call this function to write a formatted output to the shell output stream with OS synchronization, note the function could not be called in ISR.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
--------------------	----------------------------------

<i>formatString</i>	Format string.
---------------------	----------------

Returns

Returns the number of characters printed or a negative value if an error occurs.

31.7.8 void SHELL_ChangePrompt (shell_handle_t *shellHandle*, char * *prompt*)

Call this function to change shell prompt.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
<i>prompt</i>	The string which will be used for command prompt

Returns

NULL.

31.7.9 void SHELL_PrintPrompt (shell_handle_t *shellHandle*)

Call this function to print shell prompt.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
--------------------	----------------------------------

Returns

NULL.

31.7.10 void SHELL_Task (shell_handle_t *shellHandle*)

The task function for Shell; The function should be polled by upper layer. This function does not return until Shell command exit was called.

Parameters

<i>shellHandle</i>	The shell module handle pointer.
--------------------	----------------------------------

31.7.11 static bool SHELL_checkRunningInIsr(void) [inline], [static]

This function is used to check if code running in ISR.

Return values

<i>TRUE</i>	if code runing in ISR.
-------------	------------------------

Chapter 32

Serial Manager

32.1 Overview

This chapter describes the programming interface of the serial manager component.

The serial manager component provides a series of APIs to operate different serial port types. The port types it supports are UART, USB CDC and SWO.

Modules

- Serial Port Uart

Data Structures

- struct `serial_manager_config_t`
serial manager config structure. [More...](#)
- struct `serial_manager_callback_message_t`
Callback message structure. [More...](#)

Macros

- #define `SERIAL_MANAGER_NON_BLOCKING_MODE` (0U)
Enable or disable serial manager non-blocking mode (1 - enable, 0 - disable)
- #define `SERIAL_MANAGER_RING_BUFFER_FLOWCONTROL` (0U)
Enable or ring buffer flow control (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_UART` (0U)
Enable or disable uart port (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_UART_DMA` (0U)
Enable or disable uart dma port (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_USBCDC` (0U)
Enable or disable USB CDC port (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_SWO` (0U)
Enable or disable SWO port (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_VIRTUAL` (0U)
Enable or disable USB CDC virtual port (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_RPMSG` (0U)
Enable or disable rpmsg port (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_SPI_MASTER` (0U)
Enable or disable SPI Master port (1 - enable, 0 - disable)
- #define `SERIAL_PORT_TYPE_SPI_SLAVE` (0U)
Enable or disable SPI Slave port (1 - enable, 0 - disable)
- #define `SERIAL_MANAGER_TASK_HANDLE_TX` (0U)
Enable or disable SerialManager_Task() handle TX to prevent recursive calling.
- #define `SERIAL_MANAGER_WRITE_TIME_DELAY_DEFAULT_VALUE` (1U)
Set the default delay time in ms used by SerialManager_WriteTimeDelay().

- #define **SERIAL_MANAGER_READ_TIME_DELAY_DEFAULT_VALUE** (1U)
Set the default delay time in ms used by SerialManager_ReadTimeDelay().
- #define **SERIAL_MANAGER_TASK_HANDLE_RX_AVAILABLE_NOTIFY** (0U)
Enable or disable SerialManager_Task() handle RX data available notify.
- #define **SERIAL_MANAGER_WRITE_HANDLE_SIZE** (4U)
Set serial manager write handle size.
- #define **SERIAL_MANAGER_USE_COMMON_TASK** (0U)
SERIAL_PORT_UART_HANDLE_SIZE/SERIAL_PORT_USB_CDC_HANDLE_SIZE + serial manager dedicated size.
- #define **SERIAL_MANAGER_HANDLE_SIZE** (SERIAL_MANAGER_HANDLE_SIZE_TEMP + 12U)
Definition of serial manager handle size.
- #define **SERIAL_MANAGER_HANDLE_DEFINE**(name) uint32_t name[((**SERIAL_MANAGER_HANDLE_SIZE** + sizeof(uint32_t) - 1U) / sizeof(uint32_t))]
Defines the serial manager handle.
- #define **SERIAL_MANAGER_WRITE_HANDLE_DEFINE**(name) uint32_t name[((**SERIAL_MANAGER_WRITE_HANDLE_SIZE** + sizeof(uint32_t) - 1U) / sizeof(uint32_t))]
Defines the serial manager write handle.
- #define **SERIAL_MANAGER_READ_HANDLE_DEFINE**(name) uint32_t name[((**SERIAL_MANAGER_READ_HANDLE_SIZE** + sizeof(uint32_t) - 1U) / sizeof(uint32_t))]
Defines the serial manager read handle.
- #define **SERIAL_MANAGER_TASK_PRIORITY** (2U)
Macro to set serial manager task priority.
- #define **SERIAL_MANAGER_TASK_STACK_SIZE** (1000U)
Macro to set serial manager task stack size.

Typedefs

- typedef void * **serial_handle_t**
The handle of the serial manager module.
- typedef void * **serial_write_handle_t**
The write handle of the serial manager module.
- typedef void * **serial_read_handle_t**
The read handle of the serial manager module.
- typedef void(* **serial_manager_callback_t**)(void *callbackParam, **serial_manager_callback_message_t** *message, **serial_manager_status_t** status)
serial manager callback function
- typedef void(* **serial_manager_lowpower_critical_callback_t**)(void)
serial manager Lowpower Critical callback function

Enumerations

- enum `serial_port_type_t` {

 `kSerialPort_None` = 0U,

 `kSerialPort_Uart` = 1U,

 `kSerialPort_UsbCdc`,

 `kSerialPort_Swo`,

 `kSerialPort_Virtual`,

 `kSerialPort_Rpmsg`,

 `kSerialPort_UartDma`,

 `kSerialPort_SpiMaster`,

 `kSerialPort_SpiSlave` }

 serial port type
- enum `serial_manager_type_t` {

 `kSerialManager_NonBlocking` = 0x0U,

 `kSerialManager_Blocking` = 0x8F41U }

 serial manager type
- enum `serial_manager_status_t` {

 `kStatus_SerialManager_Success` = `kStatus_Success`,

 `kStatus_SerialManager_Error` = `MAKE_STATUS(kStatusGroup_SERIALMANAGER, 1)`,

 `kStatus_SerialManager_Busy` = `MAKE_STATUS(kStatusGroup_SERIALMANAGER, 2)`,

 `kStatus_SerialManager_Notify` = `MAKE_STATUS(kStatusGroup_SERIALMANAGER, 3)`,

 `kStatus_SerialManager_Canceled`,

 `kStatus_SerialManager_HandleConflict` = `MAKE_STATUS(kStatusGroup_SERIALMANAGER, 5)`,

 `kStatus_SerialManager_RingBufferOverflow`,

 `kStatus_SerialManager_NotConnected` = `MAKE_STATUS(kStatusGroup_SERIALMANAGER, 7)` }

 serial manager error code

Functions

- `serial_manager_status_t SerialManager_Init (serial_handle_t serialHandle, const serial_manager_config_t *config)`

Initializes a serial manager module with the serial manager handle and the user configuration structure.
- `serial_manager_status_t SerialManager_Deinit (serial_handle_t serialHandle)`

De-initializes the serial manager module instance.
- `serial_manager_status_t SerialManager_OpenWriteHandle (serial_handle_t serialHandle, serial_write_handle_t writeHandle)`

Opens a writing handle for the serial manager module.
- `serial_manager_status_t SerialManager_CloseWriteHandle (serial_write_handle_t writeHandle)`

Closes a writing handle for the serial manager module.
- `serial_manager_status_t SerialManager_OpenReadHandle (serial_handle_t serialHandle, serial_read_handle_t readHandle)`

Opens a reading handle for the serial manager module.
- `serial_manager_status_t SerialManager_CloseReadHandle (serial_read_handle_t readHandle)`

Closes a reading for the serial manager module.

- `serial_manager_status_t SerialManager_WriteBlocking (serial_write_handle_t writeHandle, uint8_t *buffer, uint32_t length)`
Transmits data with the blocking mode.
- `serial_manager_status_t SerialManager_ReadBlocking (serial_read_handle_t readHandle, uint8_t *buffer, uint32_t length)`
Reads data with the blocking mode.
- `serial_manager_status_t SerialManager_EnterLowpower (serial_handle_t serialHandle)`
Prepares to enter low power consumption.
- `serial_manager_status_t SerialManager_ExitLowpower (serial_handle_t serialHandle)`
Restores from low power consumption.
- `void SerialManager_SetLowpowerCriticalCb (const serial_manager_lowpower_critical_CBs_t *pfCallback)`
This function performs initialization of the callbacks structure used to disable lowpower when serial manager is active.

32.2 Data Structure Documentation

32.2.1 struct serial_manager_config_t

Data Fields

- `uint8_t * ringBuffer`
Ring buffer address, it is used to buffer data received by the hardware.
- `uint32_t ringBufferSize`
The size of the ring buffer.
- `serial_port_type_t type`
Serial port type.
- `serial_manager_type_t blockType`
Serial manager port type.
- `void * portConfig`
Serial port configuration.

Field Documentation

(1) `uint8_t* serial_manager_config_t::ringBuffer`

Besides, the memory space cannot be free during the lifetime of the serial manager module.

32.2.2 struct serial_manager_callback_message_t

Data Fields

- `uint8_t * buffer`
Transferred buffer.
- `uint32_t length`
Transferred data length.

32.3 Macro Definition Documentation

32.3.1 #define SERIAL_MANAGER_WRITE_TIME_DELAY_DEFAULT_VALUE (1U)

32.3.2 #define SERIAL_MANAGER_READ_TIME_DELAY_DEFAULT_VALUE (1U)

32.3.3 #define SERIAL_MANAGER_USE_COMMON_TASK (0U)

Macro to determine whether use common task.

32.3.4 #define SERIAL_MANAGER_HANDLE_SIZE (SERIAL_MANAGER_HANDLE_SIZE_TEMP + 12U)

**32.3.5 #define SERIAL_MANAGER_HANDLE_DEFINE(*name*) uint32_t
name[((SERIAL_MANAGER_HANDLE_SIZE + sizeof(uint32_t) - 1U) /
 sizeof(uint32_t))]**

This macro is used to define a 4 byte aligned serial manager handle. Then use "(serial_handle_t)*name*" to get the serial manager handle.

The macro should be global and could be optional. You could also define serial manager handle by yourself.

This is an example,

```
* SERIAL_MANAGER_HANDLE_DEFINE(serialManagerHandle);
*
```

Parameters

<i>name</i>	The name string of the serial manager handle.
-------------	---

**32.3.6 #define SERIAL_MANAGER_WRITE_HANDLE_DEFINE(*name*) uint32_t
name[((SERIAL_MANAGER_WRITE_HANDLE_SIZE + sizeof(uint32_t) -
 1U) / sizeof(uint32_t))]**

This macro is used to define a 4 byte aligned serial manager write handle. Then use "(serial_write_handle_t)*name*" to get the serial manager write handle.

The macro should be global and could be optional. You could also define serial manager write handle by yourself.

This is an example,

```
* SERIAL_MANAGER_WRITE_HANDLE_DEFINE(serialManagerwriteHandle);
*
```

Parameters

<i>name</i>	The name string of the serial manager write handle.
-------------	---

32.3.7 #define SERIAL_MANAGER_READ_HANDLE_DEFINE(*name*) uint32_t name[((SERIAL_MANAGER_READ_HANDLE_SIZE + sizeof(uint32_t) - 1U) / sizeof(uint32_t))]

This macro is used to define a 4 byte aligned serial manager read handle. Then use "(serial_read_handle_t)*name*" to get the serial manager read handle.

The macro should be global and could be optional. You could also define serial manager read handle by yourself.

This is an example,

```
* SERIAL_MANAGER_READ_HANDLE_DEFINE(serialManagerReadHandle);
*
```

Parameters

<i>name</i>	The name string of the serial manager read handle.
-------------	--

32.3.8 #define SERIAL_MANAGER_TASK_PRIORITY (2U)

32.3.9 #define SERIAL_MANAGER_TASK_STACK_SIZE (1000U)

32.4 Enumeration Type Documentation

32.4.1 enum serial_port_type_t

Enumerator

- kSerialPort_None* Serial port is none.
- kSerialPort_Uart* Serial port UART.
- kSerialPort_UsbCdc* Serial port USB CDC.
- kSerialPort_Swo* Serial port SWO.
- kSerialPort_Virtual* Serial port Virtual.
- kSerialPort_Rpmsg* Serial port RPMSG.
- kSerialPort_UartDma* Serial port UART DMA.

kSerialPort_SpiMaster Serial port SPIMASTER.

kSerialPort_SpiSlave Serial port SPISLAVE.

32.4.2 enum serial_manager_type_t

Enumerator

kSerialManager_NonBlocking None blocking handle.

kSerialManager_Blocking Blocking handle.

32.4.3 enum serial_manager_status_t

Enumerator

kStatus_SerialManager_Success Success.

kStatus_SerialManager_Error Failed.

kStatus_SerialManager_Busy Busy.

kStatus_SerialManager_Notify Ring buffer is not empty.

kStatus_SerialManager_Canceled the non-blocking request is canceled

kStatus_SerialManager_HandleConflict The handle is opened.

kStatus_SerialManager_RingBufferOverflow The ring buffer is overflowed.

kStatus_SerialManager_NotConnected The host is not connected.

32.5 Function Documentation

32.5.1 serial_manager_status_t SerialManager_Init (serial_handle_t *serialHandle*, const serial_manager_config_t * *config*)

This function configures the Serial Manager module with user-defined settings. The user can configure the configuration structure. The parameter *serialHandle* is a pointer to point to a memory space of size [SERIAL_MANAGER_HANDLE_SIZE](#) allocated by the caller. The Serial Manager module supports three types of serial port, UART (includes UART, USART, LPSCI, LPUART, etc), USB CDC and swo. Please refer to [serial_port_type_t](#) for serial port setting. These three types can be set by using [serial_manager_config_t](#).

Example below shows how to use this API to configure the Serial Manager. For UART,

```
* #define SERIAL_MANAGER_RING_BUFFER_SIZE (256U)
* static SERIAL_MANAGER_HANDLE_DEFINE(s_serialHandle);
* static uint8_t s_ringBuffer[SERIAL_MANAGER_RING_BUFFER_SIZE];
*
* serial_manager_config_t config;
* serial_port_uart_config_t uartConfig;
* config.type = kSerialPort_Uart;
* config.ringBuffer = &s_ringBuffer[0];
* config.ringBufferSize = SERIAL_MANAGER_RING_BUFFER_SIZE;
* uartConfig.instance = 0;
```

```

*   uartConfig.clockRate = 24000000;
*   uartConfig.baudRate = 115200;
*   uartConfig.parityMode = kSerialManager_UartParityDisabled;
*   uartConfig.stopBitCount = kSerialManager_UartOneStopBit;
*   uartConfig.enableRx = 1;
*   uartConfig.enableTx = 1;
*   uartConfig.enableRxRTS = 0;
*   uartConfig.enableTxCTS = 0;
*   config.portConfig = &uartConfig;
*   SerialManager_Init((serial_handle_t)s_serialHandle, &config);
*

```

For USB CDC,

```

*   #define SERIAL_MANAGER_RING_BUFFER_SIZE (256U)
*   static SERIAL_MANAGER_HANDLE_DEFINE(s_serialHandle);
*   static uint8_t s_ringBuffer[SERIAL_MANAGER_RING_BUFFER_SIZE];
*
*   serial_manager_config_t config;
*   serial_port_usb_cdc_config_t usbCdcConfig;
*   config.type = kSerialPort_UsbCdc;
*   config.ringBuffer = &s_ringBuffer[0];
*   config.ringBufferSize = SERIAL_MANAGER_RING_BUFFER_SIZE;
*   usbCdcConfig.controllerIndex = kSerialManager_UsbControllerKhci0;
*   config.portConfig = &usbCdcConfig;
*   SerialManager_Init((serial_handle_t)s_serialHandle, &config);
*

```

Parameters

<i>serialHandle</i>	Pointer to point to a memory space of size SERIAL_MANAGER_HANDLE_SIZE allocated by the caller. The handle should be 4 byte aligned, because unaligned access doesn't be supported on some devices. You can define the handle in the following two ways: SERIAL_MANAGER_HANDLE_DEFINE(serialHandle) ; or <code>uint32_t serialHandle[((SERIAL_MANAGER_HANDLE_SIZE + sizeof(uint32_t) - 1U) / sizeof(uint32_t))];</code>
<i>config</i>	Pointer to user-defined configuration structure.

Return values

<i>kStatus_SerialManager_Error</i>	An error occurred.
<i>kStatus_SerialManager_Success</i>	The Serial Manager module initialization succeed.

32.5.2 **serial_manager_status_t SerialManager_Deinit (serial_handle_t serialHandle)**

This function de-initializes the serial manager module instance. If the opened writing or reading handle is not closed, the function will return [kStatus_SerialManager_Busy](#).

Parameters

<i>serialHandle</i>	The serial manager module handle pointer.
---------------------	---

Return values

<i>kStatus_SerialManager_-Success</i>	The serial manager de-initialization succeed.
<i>kStatus_SerialManager_-Busy</i>	Opened reading or writing handle is not closed.

32.5.3 **serial_manager_status_t SerialManager_OpenWriteHandle (serial_handle_t *serialHandle*, serial_write_handle_t *writeHandle*)**

This function Opens a writing handle for the serial manager module. If the serial manager needs to be used in different tasks, the task should open a dedicated write handle for itself by calling [SerialManager_OpenWriteHandle](#). Since there can only one buffer for transmission for the writing handle at the same time, multiple writing handles need to be opened when the multiple transmission is needed for a task.

Parameters

<i>serialHandle</i>	The serial manager module handle pointer. The handle should be 4 byte aligned, because unaligned access doesn't be supported on some devices.
<i>writeHandle</i>	The serial manager module writing handle pointer. The handle should be 4 byte aligned, because unaligned access doesn't be supported on some devices. You can define the handle in the following two ways: SERIAL_MANAGER_WRITE_HANDLE_DEFINE(writeHandle) ; or <code>uint32_t writeHandle[((SERIAL_MANAGER_WRITE_HANDLE_SIZE + sizeof(uint32_t) - 1U) / sizeof(uint32_t))];</code>

Return values

<i>kStatus_SerialManager_-Error</i>	An error occurred.
<i>kStatus_SerialManager_-HandleConflict</i>	The writing handle was opened.

<i>kStatus_SerialManager_-Success</i>	The writing handle is opened.
---------------------------------------	-------------------------------

Example below shows how to use this API to write data. For task 1,

```
* static SERIAL_MANAGER_WRITE_HANDLE_DEFINE(s_serialWriteHandle1);
* static uint8_t s_nonBlockingWelcome1[] = "This is non-blocking writing log for task1!\r\n";
* SerialManager_OpenWriteHandle((serial_handle_t)serialHandle
    , (serial_write_handle_t)s_serialWriteHandle1);
* SerialManager_InstallTxCallback((serial_write_handle_t)s_serialWriteHandle1,
    Task1_SerialManagerTxCallback,
    s_serialWriteHandle1);
* SerialManager_WriteNonBlocking((serial_write_handle_t)s_serialWriteHandle1,
    s_nonBlockingWelcome1,
    sizeof(s_nonBlockingWelcome1) - 1U);
*
```

For task 2,

```
* static SERIAL_MANAGER_WRITE_HANDLE_DEFINE(s_serialWriteHandle2);
* static uint8_t s_nonBlockingWelcome2[] = "This is non-blocking writing log for task2!\r\n";
* SerialManager_OpenWriteHandle((serial_handle_t)serialHandle
    , (serial_write_handle_t)s_serialWriteHandle2);
* SerialManager_InstallTxCallback((serial_write_handle_t)s_serialWriteHandle2,
    Task2_SerialManagerTxCallback,
    s_serialWriteHandle2);
* SerialManager_WriteNonBlocking((serial_write_handle_t)s_serialWriteHandle2,
    s_nonBlockingWelcome2,
    sizeof(s_nonBlockingWelcome2) - 1U);
*
```

32.5.4 serial_manager_status_t SerialManager_CloseWriteHandle (serial_write_handle_t *writeHandle*)

This function Closes a writing handle for the serial manager module.

Parameters

<i>writeHandle</i>	The serial manager module writing handle pointer.
--------------------	---

Return values

<i>kStatus_SerialManager_-Success</i>	The writing handle is closed.
---------------------------------------	-------------------------------

32.5.5 serial_manager_status_t SerialManager_OpenReadHandle (serial_handle_t *serialHandle*, serial_read_handle_t *readHandle*)

This function Opens a reading handle for the serial manager module. The reading handle can not be opened multiple at the same time. The error code kStatus_SerialManager_Busy would be returned when

the previous reading handle is not closed. And there can only be one buffer for receiving for the reading handle at the same time.

Parameters

<i>serialHandle</i>	The serial manager module handle pointer. The handle should be 4 byte aligned, because unaligned access doesn't be supported on some devices.
<i>readHandle</i>	The serial manager module reading handle pointer. The handle should be 4 byte aligned, because unaligned access doesn't be supported on some devices. You can define the handle in the following two ways: SERIAL_MANAGER_READ_HANDLE_DEFINE(readHandle) ; or <code>uint32_t readHandle[((SERIAL_MANAGER_READ_HANDLE_SIZE + sizeof(uint32_t) - 1U) / sizeof(uint32_t))];</code>

Return values

<i>kStatus_SerialManager_Error</i>	An error occurred.
<i>kStatus_SerialManager_Success</i>	The reading handle is opened.
<i>kStatus_SerialManager_Busy</i>	Previous reading handle is not closed.

Example below shows how to use this API to read data.

```
* static SERIAL_MANAGER_READ_HANDLE_DEFINE(s_serialReadHandle);
* SerialManager_OpenReadHandle((serial_handle_t)serialHandle,
*     (serial_read_handle_t)s_serialReadHandle);
* static uint8_t s_nonBlockingBuffer[64];
* SerialManager_InstallRxCallback((serial_read_handle_t)s_serialReadHandle,
*     APP_SerialManagerRxCallback,
*     s_serialReadHandle);
* SerialManager_ReadNonBlocking((serial_read_handle_t)s_serialReadHandle,
*     s_nonBlockingBuffer,
*     sizeof(s_nonBlockingBuffer));
*
```

32.5.6 **serial_manager_status_t SerialManager_CloseReadHandle (serial_read_handle_t *readHandle*)**

This function Closes a reading for the serial manager module.

Parameters

<i>readHandle</i>	The serial manager module reading handle pointer.
-------------------	---

Return values

<i>kStatus_SerialManager_-Success</i>	The reading handle is closed.
---------------------------------------	-------------------------------

32.5.7 `serial_manager_status_t SerialManager_WriteBlocking (serial_write_handle_t writeHandle, uint8_t * buffer, uint32_t length)`

This is a blocking function, which polls the sending queue, waits for the sending queue to be empty. This function sends data using an interrupt method. The interrupt of the hardware could not be disabled. And There can only one buffer for transmission for the writing handle at the same time.

Note

The function `SerialManager_WriteBlocking` and the function `SerialManager_WriteNonBlocking` cannot be used at the same time. And, the function `SerialManager_CancelWriting` cannot be used to abort the transmission of this function.

Parameters

<i>writeHandle</i>	The serial manager module handle pointer.
<i>buffer</i>	Start address of the data to write.
<i>length</i>	Length of the data to write.

Return values

<i>kStatus_SerialManager_-Success</i>	Successfully sent all data.
<i>kStatus_SerialManager_-Busy</i>	Previous transmission still not finished; data not all sent yet.
<i>kStatus_SerialManager_-Error</i>	An error occurred.

32.5.8 `serial_manager_status_t SerialManager_ReadBlocking (serial_read_handle_t readHandle, uint8_t * buffer, uint32_t length)`

This is a blocking function, which polls the receiving buffer, waits for the receiving buffer to be full. This function receives data using an interrupt method. The interrupt of the hardware could not be disabled. And There can only one buffer for receiving for the reading handle at the same time.

Note

The function `SerialManager_ReadBlocking` and the function `SerialManager_ReadNonBlocking` cannot be used at the same time. And, the function `SerialManager_CancelReading` cannot be used to abort the transmission of this function.

Parameters

<code>readHandle</code>	The serial manager module handle pointer.
<code>buffer</code>	Start address of the data to store the received data.
<code>length</code>	The length of the data to be received.

Return values

<code>kStatus_SerialManager_-Success</code>	Successfully received all data.
<code>kStatus_SerialManager_-Busy</code>	Previous transmission still not finished; data not all received yet.
<code>kStatus_SerialManager_-Error</code>	An error occurred.

32.5.9 `serial_manager_status_t SerialManager_EnterLowpower (serial_handle_t serialHandle)`

This function is used to prepare to enter low power consumption.

Parameters

<code>serialHandle</code>	The serial manager module handle pointer.
---------------------------	---

Return values

<code>kStatus_SerialManager_-Success</code>	Successful operation.
---	-----------------------

32.5.10 `serial_manager_status_t SerialManager_ExitLowpower (serial_handle_t serialHandle)`

This function is used to restore from low power consumption.

Parameters

<i>serialHandle</i>	The serial manager module handle pointer.
---------------------	---

Return values

<i>kStatus_SerialManager_-Success</i>	Successful operation.
---------------------------------------	-----------------------

32.5.11 void SerialManager_SetLowpowerCriticalCb (const serial_manager_lowpower_critical_CBs_t * *pfCallback*)

Parameters

<i>pfCallback</i>	Pointer to the function structure used to allow/disable lowpower.
-------------------	---

32.6 Serial Port Uart

32.6.1 Overview

Macros

- #define **SERIAL_PORT_UART_DMA_RECEIVE_DATA_LENGTH** (64U)
serial port uart handle size
- #define **SERIAL_USE_CONFIGURE_STRUCTURE** (0U)
Enable or disable the configure structure pointer.

Enumerations

- enum **serial_port_uart_parity_mode_t** {

 kSerialManager_UartParityDisabled = 0x0U,

 kSerialManager_UartParityEven = 0x2U,

 kSerialManager_UartParityOdd = 0x3U }

serial port uart parity mode
- enum **serial_port_uart_stop_bit_count_t** {

 kSerialManager_UartOneStopBit = 0U,

 kSerialManager_UartTwoStopBit = 1U }

serial port uart stop bit count

32.6.2 Enumeration Type Documentation

32.6.2.1 enum serial_port_uart_parity_mode_t

Enumerator

- kSerialManager_UartParityDisabled*** Parity disabled.
kSerialManager_UartParityEven Parity even enabled.
kSerialManager_UartParityOdd Parity odd enabled.

32.6.2.2 enum serial_port_uart_stop_bit_count_t

Enumerator

- kSerialManager_UartOneStopBit*** One stop bit.
kSerialManager_UartTwoStopBit Two stop bits.

Chapter 33

Tsi_v5_driver

33.1 Overview

Data Structures

- struct `tsi_calibration_data_t`
TSI calibration data storage. [More...](#)
- struct `tsi_common_config_t`
TSI common configuration structure. [More...](#)
- struct `tsi_selfCap_config_t`
TSI configuration structure for self-cap mode. [More...](#)
- struct `tsi_mutualCap_config_t`
TSI configuration structure for mutual-cap mode. [More...](#)

Macros

- #define `FSL_TSI_DRIVER_VERSION` (`MAKE_VERSION(2, 3, 0)`)
TSI driver version.
- #define `ALL_FLAGS_MASK` (`TSI_GENCS_EOSF_MASK | TSI_GENCS_OUTRGF_MASK`)
TSI status flags macro collection.

Enumerations

- enum `tsi_main_clock_selection_t` {
 `kTSI_MainClockSlection_0` = 0U,
 `kTSI_MainClockSlection_1` = 1U,
 `kTSI_MainClockSlection_2` = 2U,
 `kTSI_MainClockSlection_3` = 3U }
TSI main clock selection.
- enum `tsi_sensing_mode_selection_t` {
 `kTSI_SensingModeSlection_Self` = 0U,
 `kTSI_SensingModeSlection_Mutual` = 1U }
TSI sensing mode selection.
- enum `tsi_dvolt_option_t` {
 `kTSI_DvoltOption_0` = 0U,
 `kTSI_DvoltOption_1` = 1U,
 `kTSI_DvoltOption_2` = 2U,
 `kTSI_DvoltOption_3` = 3U }
TSI DVOLT settings.
- enum `tsi_sensitivity_xdn_option_t` {

```
kTSI_SensitivityXdnOption_0 = 0U,
kTSI_SensitivityXdnOption_1 = 1U,
kTSI_SensitivityXdnOption_2 = 2U,
kTSI_SensitivityXdnOption_3 = 3U,
kTSI_SensitivityXdnOption_4 = 4U,
kTSI_SensitivityXdnOption_5 = 5U,
kTSI_SensitivityXdnOption_6 = 6U,
kTSI_SensitivityXdnOption_7 = 7U }
```

TSI sensitivity adjustment (XDN option).

- enum `tsi_shield_t` {


```
kTSI_ShieldAllOff = 0U,
kTSI_Shield0On = 1U,
kTSI_Shield1On = 2U,
kTSI_Shield1and0On = 3U,
kTSI_Shield2On = 4U,
kTSI_Shield2and0On = 5U,
kTSI_Shield2and1On = 6U,
kTSI_ShieldAllOn = 7U }
```

TSI Shield setting (S_W_SHIELD option).

- enum `tsi_sensitivity_ctrim_option_t` {


```
kTSI_SensitivityCtrimOption_0 = 0U,
kTSI_SensitivityCtrimOption_1 = 1U,
kTSI_SensitivityCtrimOption_2 = 2U,
kTSI_SensitivityCtrimOption_3 = 3U,
kTSI_SensitivityCtrimOption_4 = 4U,
kTSI_SensitivityCtrimOption_5 = 5U,
kTSI_SensitivityCtrimOption_6 = 6U,
kTSI_SensitivityCtrimOption_7 = 7U }
```

TSI sensitivity adjustment (CTRIM option).

- enum `tsi_current_multiple_input_t` {


```
kTSI_CurrentMultipleInputValue_0 = 0U,
kTSI_CurrentMultipleInputValue_1 = 1U }
```

TSI current adjustment (Input current multiple).

- enum `tsi_current_multiple_charge_t` {


```
kTSI_CurrentMultipleChargeValue_0 = 0U,
kTSI_CurrentMultipleChargeValue_1 = 1U,
kTSI_CurrentMultipleChargeValue_2 = 2U,
kTSI_CurrentMultipleChargeValue_3 = 3U,
kTSI_CurrentMultipleChargeValue_4 = 4U,
kTSI_CurrentMultipleChargeValue_5 = 5U,
kTSI_CurrentMultipleChargeValue_6 = 6U,
kTSI_CurrentMultipleChargeValue_7 = 7U }
```

TSI current adjustment (Charge/Discharge current multiple).

- enum `tsi_mutual_pre_current_t` {

```
kTSI_MutualPreCurrent_1uA = 0U,
kTSI_MutualPreCurrent_2uA = 1U,
kTSI_MutualPreCurrent_3uA = 2U,
kTSI_MutualPreCurrent_4uA = 3U,
kTSI_MutualPreCurrent_5uA = 4U,
kTSI_MutualPreCurrent_6uA = 5U,
kTSI_MutualPreCurrent_7uA = 6U,
kTSI_MutualPreCurrent_8uA = 7U }
```

TSI current used in vref generator.

- enum `tsi_mutual_pre_resistor_t` {
 kTSI_MutualPreResistor_1k = 0U,
 kTSI_MutualPreResistor_2k = 1U,
 kTSI_MutualPreResistor_3k = 2U,
 kTSI_MutualPreResistor_4k = 3U,
 kTSI_MutualPreResistor_5k = 4U,
 kTSI_MutualPreResistor_6k = 5U,
 kTSI_MutualPreResistor_7k = 6U,
 kTSI_MutualPreResistor_8k = 7U }

TSI resistor used in pre-charge.

- enum `tsi_mutual_sense_resistor_t` {
 kTSI_MutualSenseResistor_2k5 = 0U,
 kTSI_MutualSenseResistor_5k = 1U,
 kTSI_MutualSenseResistor_7k5 = 2U,
 kTSI_MutualSenseResistor_10k = 3U,
 kTSI_MutualSenseResistor_12k5 = 4U,
 kTSI_MutualSenseResistor_15k = 5U,
 kTSI_MutualSenseResistor_17k5 = 6U,
 kTSI_MutualSenseResistor_20k = 7U,
 kTSI_MutualSenseResistor_22k5 = 8U,
 kTSI_MutualSenseResistor_25k = 9U,
 kTSI_MutualSenseResistor_27k5 = 10U,
 kTSI_MutualSenseResistor_30k = 11U,
 kTSI_MutualSenseResistor_32k5 = 12U,
 kTSI_MutualSenseResistor_35k = 13U,
 kTSI_MutualSenseResistor_37k5 = 14U,
 kTSI_MutualSenseResistor_40k = 15U }

TSI resistor used in I-sense generator.

- enum `tsi_mutual_tx_channel_t` {
 kTSI_MutualTxChannel_0 = 0U,
 kTSI_MutualTxChannel_1 = 1U,
 kTSI_MutualTxChannel_2 = 2U,
 kTSI_MutualTxChannel_3 = 3U,
 kTSI_MutualTxChannel_4 = 4U,
 kTSI_MutualTxChannel_5 = 5U }

TSI TX channel selection in mutual-cap mode.

- enum `tsi_mutual_rx_channel_t` {

```
kTSI_MutualRxChannel_6 = 0U,
kTSI_MutualRxChannel_7 = 1U,
kTSI_MutualRxChannel_8 = 2U,
kTSI_MutualRxChannel_9 = 3U,
kTSI_MutualRxChannel_10 = 4U,
kTSI_MutualRxChannel_11 = 5U }
```

TSI RX channel selection in mutual-cap mode.

- enum `tsi_mutual_sense_boost_current_t` {


```
kTSI_MutualSenseBoostCurrent_0uA = 0U,
kTSI_MutualSenseBoostCurrent_2uA = 1U,
kTSI_MutualSenseBoostCurrent_4uA = 2U,
kTSI_MutualSenseBoostCurrent_6uA = 3U,
kTSI_MutualSenseBoostCurrent_8uA = 4U,
kTSI_MutualSenseBoostCurrent_10uA = 5U,
kTSI_MutualSenseBoostCurrent_12uA = 6U,
kTSI_MutualSenseBoostCurrent_14uA = 7U,
kTSI_MutualSenseBoostCurrent_16uA = 8U,
kTSI_MutualSenseBoostCurrent_18uA = 9U,
kTSI_MutualSenseBoostCurrent_20uA = 10U,
kTSI_MutualSenseBoostCurrent_22uA = 11U,
kTSI_MutualSenseBoostCurrent_24uA = 12U,
kTSI_MutualSenseBoostCurrent_26uA = 13U,
kTSI_MutualSenseBoostCurrent_28uA = 14U,
kTSI_MutualSenseBoostCurrent_30uA = 15U,
kTSI_MutualSenseBoostCurrent_32uA = 16U,
kTSI_MutualSenseBoostCurrent_34uA = 17U,
kTSI_MutualSenseBoostCurrent_36uA = 18U,
kTSI_MutualSenseBoostCurrent_38uA = 19U,
kTSI_MutualSenseBoostCurrent_40uA = 20U,
kTSI_MutualSenseBoostCurrent_42uA = 21U,
kTSI_MutualSenseBoostCurrent_44uA = 22U,
kTSI_MutualSenseBoostCurrent_46uA = 23U,
kTSI_MutualSenseBoostCurrent_48uA = 24U,
kTSI_MutualSenseBoostCurrent_50uA = 25U,
kTSI_MutualSenseBoostCurrent_52uA = 26U,
kTSI_MutualSenseBoostCurrent_54uA = 27U,
kTSI_MutualSenseBoostCurrent_56uA = 28U,
kTSI_MutualSenseBoostCurrent_58uA = 29U,
kTSI_MutualSenseBoostCurrent_60uA = 30U,
kTSI_MutualSenseBoostCurrent_62uA = 31U }
```

TSI sensitivity boost current settings.

- enum `tsi_mutual_tx_drive_mode_t` {


```
kTSI_MutualTxDriveModeOption_0 = 0U,
kTSI_MutualTxDriveModeOption_1 = 1U }
```

TSI TX drive mode control.

- enum `tsi_mutual_pmos_current_left_t` {

kTSI_MutualPmosCurrentMirrorLeft_4 = 0U,
 kTSI_MutualPmosCurrentMirrorLeft_8 = 1U,
 kTSI_MutualPmosCurrentMirrorLeft_12 = 2U,
 kTSI_MutualPmosCurrentMirrorLeft_16 = 3U,
 kTSI_MutualPmosCurrentMirrorLeft_20 = 4U,
 kTSI_MutualPmosCurrentMirrorLeft_24 = 5U,
 kTSI_MutualPmosCurrentMirrorLeft_28 = 6U,
 kTSI_MutualPmosCurrentMirrorLeft_32 = 7U }

TSI Pmos current mirror selection on the left side.

- enum `tsi_mutual_pmos_current_right_t` {

kTSI_MutualPmosCurrentMirrorRight_1 = 0U,
 kTSI_MutualPmosCurrentMirrorRight_2 = 1U,
 kTSI_MutualPmosCurrentMirrorRight_3 = 2U,
 kTSI_MutualPmosCurrentMirrorRight_4 = 3U }

TSI Pmos current mirror selection on the right side.

- enum `tsi_mutual_nmos_current_t` {

kTSI_MutualNmosCurrentMirror_1 = 0U,
 kTSI_MutualNmosCurrentMirror_2 = 1U,
 kTSI_MutualNmosCurrentMirror_3 = 2U,
 kTSI_MutualNmosCurrentMirror_4 = 3U }

TSI Nmos current mirror selection.

- enum `tsi_sinc_cutoff_div_t` {

kTSI_SincCutoffDiv_1 = 0U,
 kTSI_SincCutoffDiv_2 = 1U,
 kTSI_SincCutoffDiv_4 = 2U,
 kTSI_SincCutoffDiv_8 = 3U,
 kTSI_SincCutoffDiv_16 = 4U,
 kTSI_SincCutoffDiv_32 = 5U,
 kTSI_SincCutoffDiv_64 = 6U,
 kTSI_SincCutoffDiv_128 = 7U }

TSI SINC cutoff divider setting.

- enum `tsi_sinc_filter_order_t` {

kTSI_SincFilterOrder_1 = 0U,
 kTSI_SincFilterOrder_2 = 1U }

TSI SINC filter order setting.

- enum `tsi_sinc_decimation_value_t` {

```
kTSI_SincDecimationValue_1 = 0U,  
kTSI_SincDecimationValue_2 = 1U,  
kTSI_SincDecimationValue_3 = 2U,  
kTSI_SincDecimationValue_4 = 3U,  
kTSI_SincDecimationValue_5 = 4U,  
kTSI_SincDecimationValue_6 = 5U,  
kTSI_SincDecimationValue_7 = 6U,  
kTSI_SincDecimationValue_8 = 7U,  
kTSI_SincDecimationValue_9 = 8U,  
kTSI_SincDecimationValue_10 = 9U,  
kTSI_SincDecimationValue_11 = 10U,  
kTSI_SincDecimationValue_12 = 11U,  
kTSI_SincDecimationValue_13 = 12U,  
kTSI_SincDecimationValue_14 = 13U,  
kTSI_SincDecimationValue_15 = 14U,  
kTSI_SincDecimationValue_16 = 15U,  
kTSI_SincDecimationValue_17 = 16U,  
kTSI_SincDecimationValue_18 = 17U,  
kTSI_SincDecimationValue_19 = 18U,  
kTSI_SincDecimationValue_20 = 19U,  
kTSI_SincDecimationValue_21 = 20U,  
kTSI_SincDecimationValue_22 = 21U,  
kTSI_SincDecimationValue_23 = 22U,  
kTSI_SincDecimationValue_24 = 23U,  
kTSI_SincDecimationValue_25 = 24U,  
kTSI_SincDecimationValue_26 = 25U,  
kTSI_SincDecimationValue_27 = 26U,  
kTSI_SincDecimationValue_28 = 27U,  
kTSI_SincDecimationValue_29 = 28U,  
kTSI_SincDecimationValue_30 = 29U,  
kTSI_SincDecimationValue_31 = 30U,  
kTSI_SincDecimationValue_32 = 31U }
```

TSI SINC decimation value setting.

- enum `tsi_ssc_charge_num_t` {

```
kTSI_SscChargeNumValue_1 = 0U,
kTSI_SscChargeNumValue_2 = 1U,
kTSI_SscChargeNumValue_3 = 2U,
kTSI_SscChargeNumValue_4 = 3U,
kTSI_SscChargeNumValue_5 = 4U,
kTSI_SscChargeNumValue_6 = 5U,
kTSI_SscChargeNumValue_7 = 6U,
kTSI_SscChargeNumValue_8 = 7U,
kTSI_SscChargeNumValue_9 = 8U,
kTSI_SscChargeNumValue_10 = 9U,
kTSI_SscChargeNumValue_11 = 10U,
kTSI_SscChargeNumValue_12 = 11U,
kTSI_SscChargeNumValue_13 = 12U,
kTSI_SscChargeNumValue_14 = 13U,
kTSI_SscChargeNumValue_15 = 14U,
kTSI_SscChargeNumValue_16 = 15U }
```

TSI SSC output bit0's period setting(SSC0[CHARGE_NUM])

- enum `tsi_ssc_nocharge_num_t` {


```
kTSI_SscNoChargeNumValue_1 = 0U,
kTSI_SscNoChargeNumValue_2 = 1U,
kTSI_SscNoChargeNumValue_3 = 2U,
kTSI_SscNoChargeNumValue_4 = 3U,
kTSI_SscNoChargeNumValue_5 = 4U,
kTSI_SscNoChargeNumValue_6 = 5U,
kTSI_SscNoChargeNumValue_7 = 6U,
kTSI_SscNoChargeNumValue_8 = 7U,
kTSI_SscNoChargeNumValue_9 = 8U,
kTSI_SscNoChargeNumValue_10,
kTSI_SscNoChargeNumValue_11,
kTSI_SscNoChargeNumValue_12,
kTSI_SscNoChargeNumValue_13,
kTSI_SscNoChargeNumValue_14,
kTSI_SscNoChargeNumValue_15,
kTSI_SscNoChargeNumValue_16 }
```

TSI SSC output bit1's period setting(SSC0[BASE_NOCHARGE_NUM])

- enum `tsi_ssc_prbs_outsel_t` {

```
kTSI_SscPrbsOutsel_2 = 2U,
kTSI_SscPrbsOutsel_3 = 3U,
kTSI_SscPrbsOutsel_4 = 4U,
kTSI_SscPrbsOutsel_5 = 5U,
kTSI_SscPrbsOutsel_6 = 6U,
kTSI_SscPrbsOutsel_7 = 7U,
kTSI_SscPrbsOutsel_8 = 8U,
kTSI_SscPrbsOutsel_9 = 9U,
kTSI_SscPrbsOutsel_10 = 10U,
kTSI_SscPrbsOutsel_11 = 11U,
kTSI_SscPrbsOutsel_12 = 12U,
kTSI_SscPrbsOutsel_13 = 13U,
kTSI_SscPrbsOutsel_14 = 14U,
kTSI_SscPrbsOutsel_15 = 15U }
```

*TSI SSC outsel choosing the length of the PRBS (Pseudo-RandomBinarySequence) method setting(SSC0[-
TSI_SSC0_PRBS_OUTSEL])*

- enum **tsi_status_flags_t** {


```
kTSI_EndOfScanFlag = TSI_GENCS_EOSF_MASK,
kTSI_OutOfRangeFlag = (int)TSI_GENCS_OUTRGF_MASK }
```

TSI status flags.
- enum **tsi_interrupt_enable_t** {


```
kTSI_GlobalInterruptEnable = 1U,
kTSI_OutOfRangeInterruptEnable = 2U,
kTSI_EndOfScanInterruptEnable = 4U }
```

TSI feature interrupt source.
- enum **tsi_ssc_mode_t** {


```
kTSI_ssc_prbs_method = 0U,
kTSI_ssc_up_down_counter = 1U,
kTSI_ssc_dissable = 2U }
```

TSI SSC mode selection.
- enum **tsi_ssc_prescaler_t** {


```
kTSI_ssc_div_by_1 = 0x0U,
kTSI_ssc_div_by_2 = 0x1U,
kTSI_ssc_div_by_4 = 0x3U,
kTSI_ssc_div_by_8 = 0x7U,
kTSI_ssc_div_by_16 = 0xfU,
kTSI_ssc_div_by_32 = 0x1fU,
kTSI_ssc_div_by_64 = 0x3fU,
kTSI_ssc_div_by_128 = 0x7fU,
kTSI_ssc_div_by_256 = 0xffU }
```

TSI main clock selection.

Functions

- uint32_t **TSI_GetInstance** (TSI_Type *base)

Get the TSI instance from peripheral base address.
- void **TSI_InitSelfCapMode** (TSI_Type *base, const **tsi_selfCap_config_t** *config)

- **`void TSI_InitMutualCapMode (TSI_Type *base, const tsi_mutualCap_config_t *config)`**
 - Initialize hardware to Self-cap mode.*
- **`void TSI_Deinit (TSI_Type *base)`**
 - De-initialize hardware.*
- **`void TSI_GetSelfCapModeDefaultConfig (tsi_selfCap_config_t *userConfig)`**
 - Get TSI self-cap mode user configure structure.*
- **`void TSI_GetMutualCapModeDefaultConfig (tsi_mutualCap_config_t *userConfig)`**
 - Get TSI mutual-cap mode default user configure structure.*
- **`void TSI_SelfCapCalibrate (TSI_Type *base, tsi_calibration_data_t *calBuff)`**
 - Hardware base counter value for calibration.*
- **`void TSI_EnableInterrupts (TSI_Type *base, uint32_t mask)`**
 - Enables TSI interrupt requests.*
- **`void TSI_DisableInterrupts (TSI_Type *base, uint32_t mask)`**
 - Disables TSI interrupt requests.*
- **`static uint32_t TSI_GetStatusFlags (TSI_Type *base)`**
 - Get interrupt flag.*
- **`void TSI_ClearStatusFlags (TSI_Type *base, uint32_t mask)`**
 - Clear interrupt flag.*
- **`static uint32_t TSI_GetScanTriggerMode (TSI_Type *base)`**
 - Get TSI scan trigger mode.*
- **`static bool TSI_IsScanInProgress (TSI_Type *base)`**
 - Get scan in progress flag.*
- **`static void TSI_EnableModule (TSI_Type *base, bool enable)`**
 - Enables the TSI Module or not.*
- **`static void TSI_EnableLowPower (TSI_Type *base, bool enable)`**
 - Sets the TSI low power STOP mode enable or not.*
- **`static void TSI_EnableHardwareTriggerScan (TSI_Type *base, bool enable)`**
 - Enable the hardware trigger scan or not.*
- **`static void TSI_StartSoftwareTrigger (TSI_Type *base)`**
 - Start one software trigger measurement (trigger a new measurement).*
- **`static void TSI_SetSelfCapMeasuredChannel (TSI_Type *base, uint8_t channel)`**
 - Set the measured channel number for self-cap mode.*
- **`static uint8_t TSI_GetSelfCapMeasuredChannel (TSI_Type *base)`**
 - Get the current measured channel number, in self-cap mode.*
- **`static void TSI_EnableDmaTransfer (TSI_Type *base, bool enable)`**
 - Enable DMA transfer or not.*
- **`static void TSI_EnableEndOfScanDmaTransferOnly (TSI_Type *base, bool enable)`**
 - Decide whether to enable End of Scan DMA transfer request only.*
- **`static uint16_t TSI_GetCounter (TSI_Type *base)`**
 - Gets the conversion counter value.*
- **`static void TSI_SetLowThreshold (TSI_Type *base, uint16_t low_threshold)`**
 - Set the TSI wake-up channel low threshold.*
- **`static void TSI_SetHighThreshold (TSI_Type *base, uint16_t high_threshold)`**
 - Set the TSI wake-up channel high threshold.*
- **`static void TSI_SetMainClock (TSI_Type *base, tsi_main_clock_selection_t mainClock)`**
 - Set the main clock of the TSI module.*
- **`static void TSI_SetSensingMode (TSI_Type *base, tsi_sensing_mode_selection_t mode)`**
 - Set the sensing mode of the TSI module.*
- **`static tsi_sensing_mode_selection_t TSI_GetSensingMode (TSI_Type *base)`**
 - Get the sensing mode of the TSI module.*

- static void [TSI_SetDvolt](#) (TSI_Type *base, [tsi_dvolt_option_t](#) dvolt)
Set the DVOLT settings.
- static void [TSI_EnableNoiseCancellation](#) (TSI_Type *base, bool enableCancellation)
Enable self-cap mode noise cancellation function or not.
- static void [TSI_SetMutualCapTxChannel](#) (TSI_Type *base, [tsi_mutual_tx_channel_t](#) txChannel)
Set the mutual-cap mode TX channel.
- static [tsi_mutual_tx_channel_t](#) [TSI_GetTxMutualCapMeasuredChannel](#) (TSI_Type *base)
Get the current measured TX channel number, in mutual-cap mode.
- static void [TSI_SetMutualCapRxChannel](#) (TSI_Type *base, [tsi_mutual_rx_channel_t](#) rxChannel)
Set the mutual-cap mode RX channel.
- static [tsi_mutual_rx_channel_t](#) [TSI_GetRxMutualCapMeasuredChannel](#) (TSI_Type *base)
Get the current measured RX channel number, in mutual-cap mode.
- static void [TSI_SetSscMode](#) (TSI_Type *base, [tsi_ssc_mode_t](#) mode)
Set the SSC clock mode of the TSI module.
- static void [TSI_SetSscPrescaler](#) (TSI_Type *base, [tsi_ssc_prescaler_t](#) prescaler)
Set the SSC prescaler of the TSI module.
- static void [TSI_SetUsedTxChannel](#) (TSI_Type *base, [tsi_mutual_tx_channel_t](#) txChannel)
Set used mutual-cap TX channel.
- static void [TSI_ClearUsedTxChannel](#) (TSI_Type *base, [tsi_mutual_tx_channel_t](#) txChannel)
Clear used mutual-cap TX channel.

33.2 Data Structure Documentation

33.2.1 struct tsi_calibration_data_t

Data Fields

- uint16_t [calibratedData](#) [FSL FEATURE TSI CHANNEL COUNT]
TSI calibration data storage buffer.

33.2.2 struct tsi_common_config_t

This structure contains the common settings for TSI self-cap or mutual-cap mode, configurations including the TSI module main clock, sensing mode, DVOLT options, SINC and SSC configurations.

Data Fields

- [tsi_main_clock_selection_t](#) [mainClock](#)
Set main clock.
- [tsi_sensing_mode_selection_t](#) [mode](#)
Choose sensing mode.
- [tsi_dvolt_option_t](#) [dvolt](#)
DVOLT option value.
- [tsi_sinc_cutoff_div_t](#) [cutoff](#)
Cutoff divider.
- [tsi_sinc_filter_order_t](#) [order](#)
SINC filter order.

- **tsi_sinc_decimation_value_t decimation**
SINC decimation value.
- **tsi_ssc_charge_num_t chargeNum**
SSC High Width (t1), SSC output bit0's period setting.
- **tsi_ssc_prbs_outsel_t prbsOutsel**
SSC High Random Width (t2), length of PRBS(Pseudo-RandomBinarySequence),SSC output bit2's period setting.
- **tsi_ssc_nocharge_num_t noChargeNum**
SSC Low Width (t3), SSC output bit1's period setting.
- **tsi_ssc_mode_t ssc_mode**
Clock mode selection (basic - from main clock by divider,advanced - using SSC(Switching Speed Clock) by three configurable intervals.
- **tsi_ssc_prescaler_t ssc_prescaler**
Set clock divider for basic mode.

Field Documentation

- (1) **tsi_main_clock_selection_t tsi_common_config_t::mainClock**
- (2) **tsi_sensing_mode_selection_t tsi_common_config_t::mode**
- (3) **tsi_dvolt_option_t tsi_common_config_t::dvolt**
- (4) **tsi_sinc_cutoff_div_t tsi_common_config_t::cutoff**
- (5) **tsi_sinc_filter_order_t tsi_common_config_t::order**
- (6) **tsi_sinc_decimation_value_t tsi_common_config_t::decimation**
- (7) **tsi_ssc_charge_num_t tsi_common_config_t::chargeNum**
- (8) **tsi_ssc_prbs_outsel_t tsi_common_config_t::prbsOutsel**
- (9) **tsi_ssc_nocharge_num_t tsi_common_config_t::noChargeNum**
- (10) **tsi_ssc_mode_t tsi_common_config_t::ssc_mode**
- (11) **tsi_ssc_prescaler_t tsi_common_config_t::ssc_prescaler**

33.2.3 struct tsi_selfCap_config_t

This structure contains the settings for the most common TSI self-cap configurations including the TSI module charge currents, sensitivity configuration and so on.

Data Fields

- **tsi_common_config_t commonConfig**
Common settings.
- **bool enableSensitivity**

- [tsi_shield_t enableShield](#)
Enable sensitivity boost of self-cap or not.
- [tsi_sensitivity_xdn_option_t xdn](#)
Sensitivity XDN option.
- [tsi_sensitivity_ctrim_option_t ctrim](#)
Sensitivity CTRIM option.
- [tsi_current_multiple_input_t inputCurrent](#)
Input current multiple.
- [tsi_current_multiple_charge_t chargeCurrent](#)
Charge/Discharge current multiple.

Field Documentation

- (1) [tsi_common_config_t tsi_selfCap_config_t::commonConfig](#)
- (2) [bool tsi_selfCap_config_t::enableSensitivity](#)
- (3) [tsi_shield_t tsi_selfCap_config_t::enableShield](#)
- (4) [tsi_sensitivity_xdn_option_t tsi_selfCap_config_t::xdn](#)
- (5) [tsi_sensitivity_ctrim_option_t tsi_selfCap_config_t::ctrim](#)
- (6) [tsi_current_multiple_input_t tsi_selfCap_config_t::inputCurrent](#)
- (7) [tsi_current_multiple_charge_t tsi_selfCap_config_t::chargeCurrent](#)

33.2.4 struct tsi_mutualCap_config_t

This structure contains the settings for the most common TSI mutual-cap configurations including the TSI module generator settings, sensitivity related current settings and so on.

Data Fields

- [tsi_common_config_t commonConfig](#)
Common settings.
- [tsi_mutual_pre_current_t preCurrent](#)
Vref generator current.
- [tsi_mutual_pre_resistor_t preResistor](#)
Vref generator resistor.
- [tsi_mutual_sense_resistor_t senseResistor](#)
I-sense generator resistor.
- [tsi_mutual_sense_boost_current_t boostCurrent](#)
Sensitivity boost current setting.
- [tsi_mutual_tx_drive_mode_t txDriveMode](#)
TX drive mode control setting.
- [tsi_mutual_pmos_current_left_t pmosLeftCurrent](#)
Pmos current mirror on the left side.

- `tsi_mutual_pmos_current_right_t pmosRightCurrent`
Pmos current mirror on the right side.
- `bool enableNmosMirror`
Enable Nmos current mirror setting or not.
- `tsi_mutual_nmos_current_t nmosCurrent`
Nmos current mirror setting.

Field Documentation

- (1) `tsi_common_config_t tsi_mutualCap_config_t::commonConfig`
- (2) `tsi_mutual_pre_current_t tsi_mutualCap_config_t::preCurrent`
- (3) `tsi_mutual_pre_resistor_t tsi_mutualCap_config_t::preResistor`
- (4) `tsi_mutual_sense_resistor_t tsi_mutualCap_config_t::senseResistor`
- (5) `tsi_mutual_sense_boost_current_t tsi_mutualCap_config_t::boostCurrent`
- (6) `tsi_mutual_tx_drive_mode_t tsi_mutualCap_config_t::txDriveMode`
- (7) `tsi_mutual_pmos_current_left_t tsi_mutualCap_config_t::pmosLeftCurrent`
- (8) `tsi_mutual_pmos_current_right_t tsi_mutualCap_config_t::pmosRightCurrent`
- (9) `bool tsi_mutualCap_config_t::enableNmosMirror`
- (10) `tsi_mutual_nmos_current_t tsi_mutualCap_config_t::nmosCurrent`

33.3 Enumeration Type Documentation**33.3.1 enum tsi_main_clock_selection_t**

These constants set the tsi main clock.

Enumerator

- `kTSI_MainClockSelection_0` Set TSI main clock frequency to 20.72MHz.
- `kTSI_MainClockSelection_1` Set TSI main clock frequency to 16.65MHz.
- `kTSI_MainClockSelection_2` Set TSI main clock frequency to 13.87MHz.
- `kTSI_MainClockSelection_3` Set TSI main clock frequency to 11.91MHz.

33.3.2 enum tsi_sensing_mode_selection_t

These constants set the tsi sensing mode.

Enumerator

- `kTSI_SensingModeSelection_Self` Set TSI sensing mode to self-cap mode.

kTSI_SensingModeSelection_Mutual Set TSI sensing mode to mutual-cap mode.

33.3.3 enum tsi_dvolt_option_t

These bits indicate the comparator vp, vm and dvolt voltage.

Enumerator

- kTSI_DvoltOption_0* DVOLT value option 0, the value may differ on different platforms.
- kTSI_DvoltOption_1* DVOLT value option 1, the value may differ on different platforms.
- kTSI_DvoltOption_2* DVOLT value option 2, the value may differ on different platforms.
- kTSI_DvoltOption_3* DVOLT value option 3, the value may differ on different platforms.

33.3.4 enum tsi_sensitivity_xdn_option_t

These constants define the tsi sensitivity adjustment in self-cap mode, when TSI_MODE[S_SEN] = 1.

Enumerator

- kTSI_SensitivityXdnOption_0* Adjust sensitivity in self-cap mode, 1/16.
- kTSI_SensitivityXdnOption_1* Adjust sensitivity in self-cap mode, 1/8.
- kTSI_SensitivityXdnOption_2* Adjust sensitivity in self-cap mode, 1/4.
- kTSI_SensitivityXdnOption_3* Adjust sensitivity in self-cap mode, 1/2.
- kTSI_SensitivityXdnOption_4* Adjust sensitivity in self-cap mode, 1/1.
- kTSI_SensitivityXdnOption_5* Adjust sensitivity in self-cap mode, 2/1.
- kTSI_SensitivityXdnOption_6* Adjust sensitivity in self-cap mode, 4/1.
- kTSI_SensitivityXdnOption_7* Adjust sensitivity in self-cap mode, 8/1.

33.3.5 enum tsi_shield_t

These constants define the shield pin used for HW shielding functionality. One or more shield pin can be selected. The involved bitfield is not fix can change from device to device (KE16Z7 and KE17Z7 support 3 shield pins, other KE serials only support 1 shield pin).

Enumerator

- kTSI_shieldAllOff* No pin used.
- kTSI_shield0On* Shield 0 pin used.
- kTSI_shield1On* Shield 1 pin used.
- kTSI_shield1and0On* Shield 0,1 pins used.
- kTSI_shield2On* Shield 2 pin used.
- kTSI_shield2and0On* Shield 2,0 pins used.

kTSI_shield2and1On Shield 2,1 pins used.

kTSI_shieldAllOn Shield 2,1,0 pins used.

33.3.6 enum tsi_sensitivity_ctrim_option_t

These constants define the tsi sensitivity adjustment in self-cap mode, when TSI_MODE[S_SEN] = 1.

Enumerator

- kTSI_SensitivityCtrimOption_0* Adjust sensitivity in self-cap mode, 2.5p.
- kTSI_SensitivityCtrimOption_1* Adjust sensitivity in self-cap mode, 5.0p.
- kTSI_SensitivityCtrimOption_2* Adjust sensitivity in self-cap mode, 7.5p.
- kTSI_SensitivityCtrimOption_3* Adjust sensitivity in self-cap mode, 10.0p.
- kTSI_SensitivityCtrimOption_4* Adjust sensitivity in self-cap mode, 12.5p.
- kTSI_SensitivityCtrimOption_5* Adjust sensitivity in self-cap mode, 15.0p.
- kTSI_SensitivityCtrimOption_6* Adjust sensitivity in self-cap mode, 17.5p.
- kTSI_SensitivityCtrimOption_7* Adjust sensitivity in self-cap mode, 20.0p.

33.3.7 enum tsi_current_multiple_input_t

These constants set the tsi input current multiple in self-cap mode.

Enumerator

- kTSI_CurrentMultipleInputValue_0* Adjust input current multiple in self-cap mode, 1/8.
- kTSI_CurrentMultipleInputValue_1* Adjust input current multiple in self-cap mode, 1/4.

33.3.8 enum tsi_current_multiple_charge_t

These constants set the tsi charge/discharge current multiple in self-cap mode.

Enumerator

- kTSI_CurrentMultipleChargeValue_0* Adjust charge/discharge current multiple in self-cap mode, 1/16.
- kTSI_CurrentMultipleChargeValue_1* Adjust charge/discharge current multiple in self-cap mode, 1/8.
- kTSI_CurrentMultipleChargeValue_2* Adjust charge/discharge current multiple in self-cap mode, 1/4.
- kTSI_CurrentMultipleChargeValue_3* Adjust charge/discharge current multiple in self-cap mode, 1/2.

kTSI_CurrentMultipleChargeValue_4 Adjust charge/discharge current multiple in self-cap mode, 1/1.

kTSI_CurrentMultipleChargeValue_5 Adjust charge/discharge current multiple in self-cap mode, 2/1.

kTSI_CurrentMultipleChargeValue_6 Adjust charge/discharge current multiple in self-cap mode, 4/1.

kTSI_CurrentMultipleChargeValue_7 Adjust charge/discharge current multiple in self-cap mode, 8/1.

33.3.9 enum tsi_mutual_pre_current_t

These constants Choose the current used in vref generator.

Enumerator

kTSI_MutualPreCurrent_1uA Vref generator current is 1uA, used in mutual-cap mode.

kTSI_MutualPreCurrent_2uA Vref generator current is 2uA, used in mutual-cap mode.

kTSI_MutualPreCurrent_3uA Vref generator current is 3uA, used in mutual-cap mode.

kTSI_MutualPreCurrent_4uA Vref generator current is 4uA, used in mutual-cap mode.

kTSI_MutualPreCurrent_5uA Vref generator current is 5uA, used in mutual-cap mode.

kTSI_MutualPreCurrent_6uA Vref generator current is 6uA, used in mutual-cap mode.

kTSI_MutualPreCurrent_7uA Vref generator current is 7uA, used in mutual-cap mode.

kTSI_MutualPreCurrent_8uA Vref generator current is 8uA, used in mutual-cap mode.

33.3.10 enum tsi_mutual_pre_resistor_t

These constants Choose the resistor used in pre-charge.

Enumerator

kTSI_MutualPreResistor_1k Vref generator resistor is 1k, used in mutual-cap mode.

kTSI_MutualPreResistor_2k Vref generator resistor is 2k, used in mutual-cap mode.

kTSI_MutualPreResistor_3k Vref generator resistor is 3k, used in mutual-cap mode.

kTSI_MutualPreResistor_4k Vref generator resistor is 4k, used in mutual-cap mode.

kTSI_MutualPreResistor_5k Vref generator resistor is 5k, used in mutual-cap mode.

kTSI_MutualPreResistor_6k Vref generator resistor is 6k, used in mutual-cap mode.

kTSI_MutualPreResistor_7k Vref generator resistor is 7k, used in mutual-cap mode.

kTSI_MutualPreResistor_8k Vref generator resistor is 8k, used in mutual-cap mode.

33.3.11 enum tsi_mutual_sense_resistor_t

These constants Choose the resistor used in I-sense generator.

Enumerator

- kTSI_MutualSenseResistor_2k5* I-sense resistor is 2.5k , used in mutual-cap mode.
- kTSI_MutualSenseResistor_5k* I-sense resistor is 5.0k , used in mutual-cap mode.
- kTSI_MutualSenseResistor_7k5* I-sense resistor is 7.5k , used in mutual-cap mode.
- kTSI_MutualSenseResistor_10k* I-sense resistor is 10.0k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_12k5* I-sense resistor is 12.5k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_15k* I-sense resistor is 15.0k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_17k5* I-sense resistor is 17.5k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_20k* I-sense resistor is 20.0k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_22k5* I-sense resistor is 22.5k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_25k* I-sense resistor is 25.0k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_27k5* I-sense resistor is 27.5k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_30k* I-sense resistor is 30.0k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_32k5* I-sense resistor is 32.5k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_35k* I-sense resistor is 35.0k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_37k5* I-sense resistor is 37.5k, used in mutual-cap mode.
- kTSI_MutualSenseResistor_40k* I-sense resistor is 40.0k, used in mutual-cap mode.

33.3.12 enum tsi_mutual_tx_channel_t

These constants Choose the TX channel used in mutual-cap mode.

Enumerator

- kTSI_MutualTxChannel_0* Select channel 0 as tx0, used in mutual-cap mode.
- kTSI_MutualTxChannel_1* Select channel 1 as tx1, used in mutual-cap mode.
- kTSI_MutualTxChannel_2* Select channel 2 as tx2, used in mutual-cap mode.
- kTSI_MutualTxChannel_3* Select channel 3 as tx3, used in mutual-cap mode.
- kTSI_MutualTxChannel_4* Select channel 4 as tx4, used in mutual-cap mode.
- kTSI_MutualTxChannel_5* Select channel 5 as tx5, used in mutual-cap mode.

33.3.13 enum tsi_mutual_rx_channel_t

These constants Choose the RX channel used in mutual-cap mode.

Enumerator

- kTSI_MutualRxChannel_6* Select channel 6 as rx6, used in mutual-cap mode.
- kTSI_MutualRxChannel_7* Select channel 7 as rx7, used in mutual-cap mode.
- kTSI_MutualRxChannel_8* Select channel 8 as rx8, used in mutual-cap mode.
- kTSI_MutualRxChannel_9* Select channel 9 as rx9, used in mutual-cap mode.
- kTSI_MutualRxChannel_10* Select channel 10 as rx10, used in mutual-cap mode.
- kTSI_MutualRxChannel_11* Select channel 11 as rx11, used in mutual-cap mode.

33.3.14 enum tsi_mutual_sense_boost_current_t

These constants set the sensitivity boost current.

Enumerator

kTSI_MutualSenseBoostCurrent_0uA Sensitivity boost current is 0uA , used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_2uA Sensitivity boost current is 2uA , used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_4uA Sensitivity boost current is 4uA , used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_6uA Sensitivity boost current is 6uA , used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_8uA Sensitivity boost current is 8uA , used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_10uA Sensitivity boost current is 10uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_12uA Sensitivity boost current is 12uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_14uA Sensitivity boost current is 14uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_16uA Sensitivity boost current is 16uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_18uA Sensitivity boost current is 18uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_20uA Sensitivity boost current is 20uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_22uA Sensitivity boost current is 22uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_24uA Sensitivity boost current is 24uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_26uA Sensitivity boost current is 26uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_28uA Sensitivity boost current is 28uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_30uA Sensitivity boost current is 30uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_32uA Sensitivity boost current is 32uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_34uA Sensitivity boost current is 34uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_36uA Sensitivity boost current is 36uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_38uA Sensitivity boost current is 38uA, used in mutual-cap

mode.

kTSI_MutualSenseBoostCurrent_40uA Sensitivity boost current is 40uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_42uA Sensitivity boost current is 42uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_44uA Sensitivity boost current is 44uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_46uA Sensitivity boost current is 46uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_48uA Sensitivity boost current is 48uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_50uA Sensitivity boost current is 50uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_52uA Sensitivity boost current is 52uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_54uA Sensitivity boost current is 54uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_56uA Sensitivity boost current is 56uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_58uA Sensitivity boost current is 58uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_60uA Sensitivity boost current is 60uA, used in mutual-cap mode.

kTSI_MutualSenseBoostCurrent_62uA Sensitivity boost current is 62uA, used in mutual-cap mode.

33.3.15 enum tsi_mutual_tx_drive_mode_t

These constants Choose the TX drive mode control setting.

Enumerator

kTSI_MutualTxDriveModeOption_0 TX drive mode is -5v ~ +5v, used in mutual-cap mode.

kTSI_MutualTxDriveModeOption_1 TX drive mode is 0v ~ +5v, used in mutual-cap mode.

33.3.16 enum tsi_mutual_pmos_current_left_t

These constants set the Pmos current mirror on the left side used in mutual-cap mode.

Enumerator

kTSI_MutualPmosCurrentMirrorLeft_4 Set Pmos current mirror left value as 4, used in mutual-cap mode.

- kTSI_MutualPmosCurrentMirrorLeft_8*** Set Pmos current mirror left value as 8, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorLeft_12*** Set Pmos current mirror left value as 12, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorLeft_16*** Set Pmos current mirror left value as 16, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorLeft_20*** Set Pmos current mirror left value as 20, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorLeft_24*** Set Pmos current mirror left value as 24, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorLeft_28*** Set Pmos current mirror left value as 28, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorLeft_32*** Set Pmos current mirror left value as 32, used in mutual-cap mode.

33.3.17 enum tsi_mutual_pmos_current_right_t

These constants set the Pmos current mirror on the right side used in mutual-cap mode.

Enumerator

- kTSI_MutualPmosCurrentMirrorRight_1*** Set Pmos current mirror right value as 1, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorRight_2*** Set Pmos current mirror right value as 2, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorRight_3*** Set Pmos current mirror right value as 3, used in mutual-cap mode.
- kTSI_MutualPmosCurrentMirrorRight_4*** Set Pmos current mirror right value as 4, used in mutual-cap mode.

33.3.18 enum tsi_mutual_nmos_current_t

These constants set the Nmos current mirror used in mutual-cap mode.

Enumerator

- kTSI_MutualNmosCurrentMirror_1*** Set Nmos current mirror value as 1, used in mutual-cap mode.
- kTSI_MutualNmosCurrentMirror_2*** Set Nmos current mirror value as 2, used in mutual-cap mode.
- kTSI_MutualNmosCurrentMirror_3*** Set Nmos current mirror value as 3, used in mutual-cap mode.
- kTSI_MutualNmosCurrentMirror_4*** Set Nmos current mirror value as 4, used in mutual-cap mode.

33.3.19 enum tsi_sinc_cutoff_div_t

These bits set the SINC cutoff divider.

Enumerator

- kTSI_SincCutoffDiv_1* Set SINC cutoff divider as 1.
- kTSI_SincCutoffDiv_2* Set SINC cutoff divider as 2.
- kTSI_SincCutoffDiv_4* Set SINC cutoff divider as 4.
- kTSI_SincCutoffDiv_8* Set SINC cutoff divider as 8.
- kTSI_SincCutoffDiv_16* Set SINC cutoff divider as 16.
- kTSI_SincCutoffDiv_32* Set SINC cutoff divider as 32.
- kTSI_SincCutoffDiv_64* Set SINC cutoff divider as 64.
- kTSI_SincCutoffDiv_128* Set SINC cutoff divider as 128.

33.3.20 enum tsi_sinc_filter_order_t

These bits set the SINC filter order.

Enumerator

- kTSI_SincFilterOrder_1* Use 1 order SINC filter.
- kTSI_SincFilterOrder_2* Use 1 order SINC filter.

33.3.21 enum tsi_sinc_decimation_value_t

These bits set the SINC decimation value.

Enumerator

- kTSI_SincDecimationValue_1* The TSI_DATA[TSICH] bits is the counter value of 1 trigger period.
- kTSI_SincDecimationValue_2* The TSI_DATA[TSICH] bits is the counter value of 2 trigger period.
- kTSI_SincDecimationValue_3* The TSI_DATA[TSICH] bits is the counter value of 3 trigger period.
- kTSI_SincDecimationValue_4* The TSI_DATA[TSICH] bits is the counter value of 4 trigger period.
- kTSI_SincDecimationValue_5* The TSI_DATA[TSICH] bits is the counter value of 5 trigger period.
- kTSI_SincDecimationValue_6* The TSI_DATA[TSICH] bits is the counter value of 6 trigger period.
- kTSI_SincDecimationValue_7* The TSI_DATA[TSICH] bits is the counter value of 7 trigger period.

kTSI_SincDecimationValue_8 The TSI_DATA[TSICH] bits is the counter value of 8 trigger period.

kTSI_SincDecimationValue_9 The TSI_DATA[TSICH] bits is the counter value of 9 trigger period.

kTSI_SincDecimationValue_10 The TSI_DATA[TSICH] bits is the counter value of 10 trigger period.

kTSI_SincDecimationValue_11 The TSI_DATA[TSICH] bits is the counter value of 11 trigger period.

kTSI_SincDecimationValue_12 The TSI_DATA[TSICH] bits is the counter value of 12 trigger period.

kTSI_SincDecimationValue_13 The TSI_DATA[TSICH] bits is the counter value of 13 trigger period.

kTSI_SincDecimationValue_14 The TSI_DATA[TSICH] bits is the counter value of 14 trigger period.

kTSI_SincDecimationValue_15 The TSI_DATA[TSICH] bits is the counter value of 15 trigger period.

kTSI_SincDecimationValue_16 The TSI_DATA[TSICH] bits is the counter value of 16 trigger period.

kTSI_SincDecimationValue_17 The TSI_DATA[TSICH] bits is the counter value of 17 trigger period.

kTSI_SincDecimationValue_18 The TSI_DATA[TSICH] bits is the counter value of 18 trigger period.

kTSI_SincDecimationValue_19 The TSI_DATA[TSICH] bits is the counter value of 19 trigger period.

kTSI_SincDecimationValue_20 The TSI_DATA[TSICH] bits is the counter value of 20 trigger period.

kTSI_SincDecimationValue_21 The TSI_DATA[TSICH] bits is the counter value of 21 trigger period.

kTSI_SincDecimationValue_22 The TSI_DATA[TSICH] bits is the counter value of 22 trigger period.

kTSI_SincDecimationValue_23 The TSI_DATA[TSICH] bits is the counter value of 23 trigger period.

kTSI_SincDecimationValue_24 The TSI_DATA[TSICH] bits is the counter value of 24 trigger period.

kTSI_SincDecimationValue_25 The TSI_DATA[TSICH] bits is the counter value of 25 trigger period.

kTSI_SincDecimationValue_26 The TSI_DATA[TSICH] bits is the counter value of 26 trigger period.

kTSI_SincDecimationValue_27 The TSI_DATA[TSICH] bits is the counter value of 27 trigger period.

kTSI_SincDecimationValue_28 The TSI_DATA[TSICH] bits is the counter value of 28 trigger period.

kTSI_SincDecimationValue_29 The TSI_DATA[TSICH] bits is the counter value of 29 trigger period.

kTSI_SincDecimationValue_30 The TSI_DATA[TSICH] bits is the counter value of 30 trigger period.

period.

kTSI_SincDecimationValue_31 The TSI_DATA[TSICH] bits is the counter value of 31 trigger period.

kTSI_SincDecimationValue_32 The TSI_DATA[TSICH] bits is the counter value of 32 trigger period.

33.3.22 enum tsi_ssc_charge_num_t

These bits set the SSC output bit0's period setting.

Enumerator

kTSI_SscChargeNumValue_1 The SSC output bit 0's period will be 1 clock cycle of system clock.

kTSI_SscChargeNumValue_2 The SSC output bit 0's period will be 2 clock cycle of system clock.

kTSI_SscChargeNumValue_3 The SSC output bit 0's period will be 3 clock cycle of system clock.

kTSI_SscChargeNumValue_4 The SSC output bit 0's period will be 4 clock cycle of system clock.

kTSI_SscChargeNumValue_5 The SSC output bit 0's period will be 5 clock cycle of system clock.

kTSI_SscChargeNumValue_6 The SSC output bit 0's period will be 6 clock cycle of system clock.

kTSI_SscChargeNumValue_7 The SSC output bit 0's period will be 7 clock cycle of system clock.

kTSI_SscChargeNumValue_8 The SSC output bit 0's period will be 8 clock cycle of system clock.

kTSI_SscChargeNumValue_9 The SSC output bit 0's period will be 9 clock cycle of system clock.

kTSI_SscChargeNumValue_10 The SSC output bit 0's period will be 10 clock cycle of system clock.

kTSI_SscChargeNumValue_11 The SSC output bit 0's period will be 11 clock cycle of system clock.

kTSI_SscChargeNumValue_12 The SSC output bit 0's period will be 12 clock cycle of system clock.

kTSI_SscChargeNumValue_13 The SSC output bit 0's period will be 13 clock cycle of system clock.

kTSI_SscChargeNumValue_14 The SSC output bit 0's period will be 14 clock cycle of system clock.

kTSI_SscChargeNumValue_15 The SSC output bit 0's period will be 15 clock cycle of system clock.

kTSI_SscChargeNumValue_16 The SSC output bit 0's period will be 16 clock cycle of system clock.

33.3.23 enum tsi_ssc_nocharge_num_t

These bits set the SSC output bit1's period setting.

Enumerator

kTSI_SscNoChargeNumValue_1 The SSC output bit 1's basic period will be 1 clock cycle of system clock.

- kTSI_SscNoChargeNumValue_2*** The SSC output bit 1's basic period will be 2 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_3*** The SSC output bit 1's basic period will be 3 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_4*** The SSC output bit 1's basic period will be 4 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_5*** The SSC output bit 1's basic period will be 5 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_6*** The SSC output bit 1's basic period will be 6 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_7*** The SSC output bit 1's basic period will be 7 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_8*** The SSC output bit 1's basic period will be 8 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_9*** The SSC output bit 1's basic period will be 9 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_10*** The SSC output bit 1's basic period will be 10 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_11*** The SSC output bit 1's basic period will be 11 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_12*** The SSC output bit 1's basic period will be 12 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_13*** The SSC output bit 1's basic period will be 13 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_14*** The SSC output bit 1's basic period will be 14 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_15*** The SSC output bit 1's basic period will be 15 clock cycle of system clock.
- kTSI_SscNoChargeNumValue_16*** The SSC output bit 1's basic period will be 16 clock cycle of system clock.

33.3.24 enum tsi_ssc_prbs_outsel_t

These bits set the SSC PRBS length.

Enumerator

- kTSI_SscPrbsOutsel_2*** The length of the PRBS is 2.
- kTSI_SscPrbsOutsel_3*** The length of the PRBS is 3.
- kTSI_SscPrbsOutsel_4*** The length of the PRBS is 4.
- kTSI_SscPrbsOutsel_5*** The length of the PRBS is 5.
- kTSI_SscPrbsOutsel_6*** The length of the PRBS is 6.
- kTSI_SscPrbsOutsel_7*** The length of the PRBS is 7.
- kTSI_SscPrbsOutsel_8*** The length of the PRBS is 8.

- kTSI_SscPrbsOutsel_9*** The length of the PRBS is 9.
- kTSI_SscPrbsOutsel_10*** The length of the PRBS is 10.
- kTSI_SscPrbsOutsel_11*** The length of the PRBS is 11.
- kTSI_SscPrbsOutsel_12*** The length of the PRBS is 12.
- kTSI_SscPrbsOutsel_13*** The length of the PRBS is 13.
- kTSI_SscPrbsOutsel_14*** The length of the PRBS is 14.
- kTSI_SscPrbsOutsel_15*** The length of the PRBS is 15.

33.3.25 enum tsi_status_flags_t

Enumerator

- kTSI_EndOfScanFlag*** End-Of-Scan flag.
- kTSI_OutOfRangeFlag*** Out-Of-Range flag.

33.3.26 enum tsi_interrupt_enable_t

Enumerator

- kTSI_GlobalInterruptEnable*** TSI module global interrupt.
- kTSI_OutOfRangeInterruptEnable*** Out-Of-Range interrupt.
- kTSI_EndOfScanInterruptEnable*** End-Of-Scan interrupt.

33.3.27 enum tsi_ssc_mode_t

These constants set the SSC mode.

Enumerator

- kTSI_ssc_prbs_method*** Using PRBS method generating SSC output bit.
- kTSI_ssc_up_down_counter*** Using up-down counter generating SSC output bit.
- kTSI_ssc_dissable*** SSC function is disabled.

33.3.28 enum tsi_ssc_prescaler_t

These constants set select the divider ratio for the clock used for generating the SSC output bit.

Enumerator

- kTSI_ssc_div_by_1*** Set SSC divider to 00000000 div1(2^0)

kTSI_ssc_div_by_2 Set SSC divider to 00000001 div2(2^1)
kTSI_ssc_div_by_4 Set SSC divider to 00000011 div4(2^2)
kTSI_ssc_div_by_8 Set SSC divider to 00000111 div8(2^3)
kTSI_ssc_div_by_16 Set SSC divider to 00001111 div16(2^4)
kTSI_ssc_div_by_32 Set SSC divider to 00011111 div32(2^5)
kTSI_ssc_div_by_64 Set SSC divider to 00111111 div64(2^6)
kTSI_ssc_div_by_128 Set SSC divider to 01111111 div128(2^7)
kTSI_ssc_div_by_256 Set SSC divider to 11111111 div256(2^8)

33.4 Function Documentation

33.4.1 `uint32_t TSI_GetInstance (TSI_Type * base)`

Parameters

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

Returns

TSI instance.

33.4.2 `void TSI_InitSelfCapMode (TSI_Type * base, const tsi_selfCap_config_t * config)`

Initialize the peripheral to the targeted state specified by parameter config, such as sets sensitivity adjustment, current settings.

Parameters

<i>base</i>	TSI peripheral base address.
<i>config</i>	Pointer to TSI self-cap configuration structure.

Returns

none

33.4.3 `void TSI_InitMutualCapMode (TSI_Type * base, const tsi_mutualCap_config_t * config)`

Initialize the peripheral to the targeted state specified by parameter config, such as sets Vref generator setting, sensitivity boost settings, Pmos/Nmos settings.

Parameters

<i>base</i>	TSI peripheral base address.
<i>config</i>	Pointer to TSI mutual-cap configuration structure.

Returns

none

33.4.4 void TSI_Deinit (TSI_Type * *base*)

De-initialize the peripheral to default state.

Parameters

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

Returns

none

33.4.5 void TSI_GetSelfCapModeDefaultConfig (tsi_selfCap_config_t * *userConfig*)

This interface sets *userConfig* structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to a value:

```

userConfig->commonConfig.mainClock      = kTSI_MainClockSelection_0;
userConfig->commonConfig.mode           = kTSI_SensingModeSelection_Self;
userConfig->commonConfig.dvolt          = kTSI_DvoltOption_2;
userConfig->commonConfig.cutoff          = kTSI_SincCutoffDiv_1;
userConfig->commonConfig.order          = kTSI_SincFilterOrder_1;
userConfig->commonConfig.decimation     = kTSI_SincDecimationValue_8;
userConfig->commonConfig.chargeNum       = kTSI_SscChargeNumValue_3;
userConfig->commonConfig.prbsOutsel     = kTSI_SscPrbsOutsel_2;
userConfig->commonConfig.noChargeNum    = kTSI_SscNoChargeNumValue_2;
userConfig->commonConfig.ssc_mode         = kTSI_ssc_prbs_method;
userConfig->commonConfig.ssc_prescaler   = kTSI_ssc_div_by_1;
userConfig->enableSensitivity          = true;
userConfig->enableShield               = false;
userConfig->xndn                      = kTSI_SensitivityXdnOption_1;
userConfig->ctrim                     = kTSI_SensitivityCtrrimOption_7;
userConfig->inputCurrent              = kTSI_CurrentMultipleInputValue_0;
userConfig->chargeCurrent             = kTSI_CurrentMultipleChargeValue_1
;
```

Parameters

<i>userConfig</i>	Pointer to TSI user configure structure.
-------------------	--

33.4.6 void TSI_GetMutualCapModeDefaultConfig (*tsi_mutualCap_config_t* * *userConfig*)

This interface sets *userConfig* structure to a default value. The configuration structure only includes the settings for the whole TSI. The user configure is set to a value:

```

userConfig->commonConfig.mainClock          = kTSI_MainClockSelection_1;
userConfig->commonConfig.mode                = kTSI_SensingModeSelection_Mutual;
userConfig->commonConfig.dvolt              = kTSI_DvoltOption_0;
userConfig->commonConfig.cutoff              = kTSI_SincCutoffDiv_1;
userConfig->commonConfig.order              = kTSI_SincFilterOrder_1;
userConfig->commonConfig.decimation         = kTSI_SincDecimationValue_8;
userConfig->commonConfig.chargeNum          = kTSI_SscChargeNumValue_4;
userConfig->commonConfig.prbsOutsel        = kTSI_SscPrbsOutsel_2;
userConfig->commonConfig.noChargeNum        = kTSI_SscNoChargeNumValue_5;
userConfig->commonConfig.ssc_mode           = kTSI_ssc_prbs_method;
userConfig->commonConfig.ssc_prescaler      = kTSI_ssc_div_by_1;
userConfig->preCurrent                     = kTSI_MutualPreCurrent_4uA;
userConfig->preResistor                    = kTSI_MutualPreResistor_4k;
userConfig->senseResistor                  = kTSI_MutualSenseResistor_10k;
userConfig->boostCurrent                  = kTSI_MutualSenseBoostCurrent_0uA;
userConfig->txDriveMode                   = kTSI_MutualTxDriveModeOption_0;
userConfig->pmosLeftCurrent               = kTSI_MutualPmosCurrentMirrorLeft_32
;
userConfig->pmosRightCurrent              = kTSI_MutualPmosCurrentMirrorRight_1
;
userConfig->enableNmosMirror              = true;
userConfig->nmosCurrent                  = kTSI_MutualNmosCurrentMirror_1;

```

Parameters

<i>userConfig</i>	Pointer to TSI user configure structure.
-------------------	--

33.4.7 void TSI_SelfCapCalibrate (*TSI_Type* * *base*, *tsi_calibration_data_t* * *calBuff*)

Calibrate the peripheral to fetch the initial counter value of the enabled channels. This API is mostly used at initial application setup, it shall be called after the TSI_Init API, then user can use the calibrated counter values to setup applications(such as to determine under which counter value we can confirm a touch event occurs).

Parameters

<i>base</i>	TSI peripheral base address.
<i>calBuff</i>	Data buffer that store the calibrated counter value.

Returns

none

Note

This API is mainly used for self-cap mode;

The calibration work in mutual-cap mode shall be done in applications due to different board layout.

33.4.8 void TSI_EnableInterrupts (*TSI_Type* * *base*, *uint32_t* *mask*)

Parameters

<i>base</i>	TSI peripheral base address.
<i>mask</i>	interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none">• kTSI_GlobalInterruptEnable• kTSI_EndOfScanInterruptEnable• kTSI_OutOfRangeInterruptEnable

33.4.9 void TSI_DisableInterrupts (*TSI_Type* * *base*, *uint32_t* *mask*)

Parameters

<i>base</i>	TSI peripheral base address.
<i>mask</i>	interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none">• kTSI_GlobalInterruptEnable• kTSI_EndOfScanInterruptEnable• kTSI_OutOfRangeInterruptEnable

33.4.10 static *uint32_t* TSI_GetStatusFlags (*TSI_Type* * *base*) [inline], [static]

This function get tsi interrupt flags.

Parameters

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

Returns

The mask of these status flags combination.

33.4.11 void TSI_ClearStatusFlags (**TSI_Type** * *base*, **uint32_t** *mask*)

This function clear tsi interrupt flag, automatically cleared flags can not be cleared by this function.

Parameters

<i>base</i>	TSI peripheral base address.
<i>mask</i>	The status flags to clear.

33.4.12 static **uint32_t** TSI_GetScanTriggerMode (**TSI_Type** * *base*) [inline], [static]

Parameters

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

Returns

Scan trigger mode.

33.4.13 static **bool** TSI_IsScanInProgress (**TSI_Type** * *base*) [inline], [static]

Parameters

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

Returns

True - scan is in progress. False - scan is not in progress.

33.4.14 **static void TSI_EnableModule (TSI_Type * *base*, bool *enable*)**
[**inline**], [**static**]

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose whether to enable or disable module; <ul style="list-style-type: none"> • true Enable TSI module; • false Disable TSI module;

Returns

none.

33.4.15 static void TSI_EnableLowPower (TSI_Type * *base*, bool *enable*) [inline], [static]

This enables TSI module function in low power modes.

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose to enable or disable STOP mode. <ul style="list-style-type: none"> • true Enable module in STOP mode; • false Disable module in STOP mode;

Returns

none.

33.4.16 static void TSI_EnableHardwareTriggerScan (TSI_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose to enable hardware trigger or software trigger scan. <ul style="list-style-type: none">• true Enable hardware trigger scan;• false Enable software trigger scan;

Returns

none.

33.4.17 static void TSI_StartSoftwareTrigger (**TSI_Type** * *base*) [inline], [static]

Parameters

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

Returns

none.

33.4.18 static void TSI_SetSelfCapMeasuredChannel (**TSI_Type** * *base*, **uint8_t** *channel*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>channel</i>	Channel number 0 ... 24.

Returns

none.

Note

This API can only be used in self-cap mode!

33.4.19 static **uint8_t** TSI_GetSelfCapMeasuredChannel (**TSI_Type** * *base*) [inline], [static]

Parameters

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

Returns

uint8_t Channel number 0 ... 24.

Note

This API can only be used in self-cap mode!

33.4.20 static void TSI_EnableDmaTransfer (**TSI_Type** * *base*, **bool** *enable*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose to enable DMA transfer or not. <ul style="list-style-type: none"> • true Enable DMA transfer; • false Disable DMA transfer;

Returns

none.

33.4.21 static void TSI_EnableEndOfScanDmaTransferOnly (**TSI_Type** * *base*, **bool** *enable*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable</i>	Choose whether to enable End of Scan DMA transfer request only. <ul style="list-style-type: none"> • true Enable End of Scan DMA transfer request only; • false Both End-of-Scan and Out-of-Range can generate DMA transfer request.

Returns

none.

33.4.22 **static uint16_t TSI_GetCounter(TSI_Type * *base*) [inline],
[static]**

Parameters

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

Returns

Accumulated scan counter value ticked by the reference clock.

33.4.23 static void TSI_SetLowThreshold (*TSI_Type* * *base*, *uint16_t* *low_threshold*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>low_threshold</i>	Low counter threshold.

Returns

none.

33.4.24 static void TSI_SetHighThreshold (*TSI_Type* * *base*, *uint16_t* *high_threshold*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>high_threshold</i>	High counter threshold.

Returns

none.

33.4.25 static void TSI_SetMainClock (*TSI_Type* * *base*, *tsi_main_clock_selection_t* *mainClock*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>mainClock</i>	clock option value.

Returns

none.

33.4.26 static void TSI_SetSensingMode (TSI_Type * *base*, tsi_sensing_mode_selection_t *mode*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>mode</i>	Mode value.

Returns

none.

33.4.27 static tsi_sensing_mode_selection_t TSI_GetSensingMode (TSI_Type * *base*) [inline], [static]

Parameters

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

Returns

Currently selected sensing mode.

33.4.28 static void TSI_SetDvolt (TSI_Type * *base*, tsi_dvolt_option_t *dvolt*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>dvolt</i>	The voltage rails.

Returns

none.

33.4.29 static void TSI_EnableNoiseCancellation (*TSI_Type* * *base*, *bool enableCancellation*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>enable-Cancellation</i>	Choose whether to enable noise cancellation in self-cap mode <ul style="list-style-type: none"> • true Enable noise cancellation; • false Disable noise cancellation;

Returns

none.

33.4.30 static void TSI_SetMutualCapTxChannel (*TSI_Type* * *base*, *tsi_mutual_tx_channel_t txChannel*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>txChannel</i>	Mutual-cap mode TX channel number

Returns

none.

33.4.31 static *tsi_mutual_tx_channel_t* TSI_GetTxMutualCapMeasuredChannel (*TSI_Type* * *base*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address;
-------------	------------------------------

Returns

Tx Channel number 0 ... 5;

Note

This API can only be used in mutual-cap mode!

33.4.32 static void TSI_SetMutualCapRxChannel (**TSI_Type** * *base*, **tsi_mutual_rx_channel_t** *rxChannel*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>rxChannel</i>	Mutual-cap mode RX channel number

Returns

none.

33.4.33 static **tsi_mutual_rx_channel_t** TSI_GetRxMutualCapMeasuredChannel (**TSI_Type** * *base*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address;
-------------	------------------------------

Returns

Rx Channel number 6 ... 11;

Note

This API can only be used in mutual-cap mode!

33.4.34 static void TSI_SetSscMode (**TSI_Type** * *base*, **tsi_ssc_mode_t** *mode*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>mode</i>	SSC mode option value.

Returns

none.

33.4.35 static void TSI_SetSscPrescaler (*TSI_Type* * *base*, *tsi_ssc_prescaler_t prescaler*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>prescaler</i>	SSC prescaler option value.

Returns

none.

33.4.36 static void TSI_SetUsedTxChannel (*TSI_Type* * *base*, *tsi_mutual_tx_channel_t txChannel*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>txChannel</i>	Mutual-cap mode TX channel number

Returns

none.

33.4.37 static void TSI_ClearUsedTxChannel (*TSI_Type* * *base*, *tsi_mutual_tx_channel_t txChannel*) [inline], [static]

Parameters

<i>base</i>	TSI peripheral base address.
<i>txChannel</i>	Mutual-cap mode TX channel number

Returns

none.

How to Reach Us:

Home Page:

nxp.com

Web Support:

nxp.com/support

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

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

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, Freescale, the Freescale logo, Kinetis, Processor Expert, and Tower are trademarks of NXP B.V. All other product or service names are the property of their respective owners. Arm, Cortex, Keil, Mbed, Mbed Enabled, and Vision are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2021 NXP B.V.

