

Document Number: MCUXSDKAPIRM
Rev 2.14.0
Jul 2023

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	17
4.2.1 struct scg_sys_clk_config_t	17
4.2.2 struct scg_sosc_config_t	18
4.2.3 struct scg_sirc_config_t	19
4.2.4 struct scg_firc_trim_config_t	19
4.2.5 struct scg_firc_config_t	20
4.2.6 struct scg_sppll_config_t	21
4.2.7 struct scg_rosc_config_t	22
4.2.8 struct scg_apll_config_t	22
4.3 Macro Definition Documentation	23
4.3.1 FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL	23
4.3.2 FSL_CLOCK_DRIVER_VERSION	24
4.3.3 SCG	24
4.3.4 DMAMUX_CLOCKS	24
4.3.5 GPIO2P_CLOCKS	24
4.3.6 SAI_CLOCKS	24
4.3.7 PCTL_CLOCKS	24
4.3.8 LPI2C_CLOCKS	25
4.3.9 FLEXIO_CLOCKS	25
4.3.10 EDMA_CLOCKS	25
4.3.11 LPUART_CLOCKS	25
4.3.12 DAC_CLOCKS	25
4.3.13 SNVS_HP_CLOCKS	26
4.3.14 LPTMR_CLOCKS	26
4.3.15 LPADC_CLOCKS	26
4.3.16 TRNG_CLOCKS	26
4.3.17 LPSPI_CLOCKS	26

Section No.	Title	Page No.
4.3.18	TPM_CLOCKS	27
4.3.19	LPIT_CLOCKS	27
4.3.20	CRC_CLOCKS	27
4.3.21	CMP_CLOCKS	27
4.3.22	XRDC_CLOCKS	27
4.3.23	MU_CLOCKS	28
4.3.24	WDOG_CLOCKS	28
4.3.25	LTC_CLOCKS	28
4.3.26	DPM_CLOCKS	28
4.3.27	SEMA42_CLOCKS	28
4.3.28	TPIU_CLOCKS	29
4.3.29	QSPI_CLOCKS	29
4.3.30	kCLOCK_Osc0ErClk	29
4.3.31	kCLOCK_Er32kClk	29
4.3.32	CLOCK_GetOsc0ErClkFreq	29
4.3.33	CLOCK_GetEr32kClkFreq	29
4.4	Enumeration Type Documentation	29
4.4.1	clock_name_t	29
4.4.2	clock_ip_src_t	30
4.4.3	clock_lptmr_src_t	31
4.4.4	clock_ip_name_t	31
4.4.5	anonymous enum	31
4.4.6	scg_sys_clk_t	31
4.4.7	scg_sys_clk_src_t	31
4.4.8	scg_sys_clk_div_t	32
4.4.9	clock_clkout_src_t	32
4.4.10	scg_async_clk_t	32
4.4.11	scg_async_clk_div_t	33
4.4.12	scg_sosc_monitor_mode_t	33
4.4.13	scg_sosc_mode_t	33
4.4.14	_scg_sosc_enable_mode	33
4.4.15	scg_sirc_range_t	33
4.4.16	_scg_sirc_enable_mode	34
4.4.17	scg_firc_trim_mode_t	34
4.4.18	scg_firc_trim_div_t	34
4.4.19	scg_firc_trim_src_t	34
4.4.20	scg_firc_range_t	35
4.4.21	_scg_firc_enable_mode	35
4.4.22	scg_spill_src_t	35
4.4.23	_scg_spill_enable_mode	35
4.4.24	scg_spill_pfd_clkout_t	35
4.4.25	scg_roscl_monitor_mode_t	36
4.4.26	scg_apll_src_t	36
4.4.27	_scg_apll_enable_mode	36

Section No.	Title	Page No.
4.4.28	<code>scg_apll_pfd_clkout_t</code>	36
4.5	Function Documentation	36
4.5.1	<code>CLOCK_EnableClock</code>	36
4.5.2	<code>CLOCK_DisableClock</code>	37
4.5.3	<code>CLOCK_IsEnabledByOtherCore</code>	37
4.5.4	<code>CLOCK_SetIpSrc</code>	37
4.5.5	<code>CLOCK_SetIpSrcDiv</code>	37
4.5.6	<code>CLOCK_GetFreq</code>	39
4.5.7	<code>CLOCK_GetCoreSysClkFreq</code>	39
4.5.8	<code>CLOCK_GetPlatClkFreq</code>	39
4.5.9	<code>CLOCK_GetExtClkFreq</code>	39
4.5.10	<code>CLOCK_GetBusClkFreq</code>	40
4.5.11	<code>CLOCK_GetSlowClkFreq</code>	40
4.5.12	<code>CLOCK_GetOsc32kClkFreq</code>	40
4.5.13	<code>CLOCK_GetErClkFreq</code>	40
4.5.14	<code>CLOCK_GetLvdsClkFreq</code>	40
4.5.15	<code>CLOCK_GetIpFreq</code>	40
4.5.16	<code>CLOCK_GetSysClkFreq</code>	41
4.5.17	<code>CLOCK_SetVlprModeSysClkConfig</code>	41
4.5.18	<code>CLOCK_SetRunModeSysClkConfig</code>	41
4.5.19	<code>CLOCK_SetHsrunModeSysClkConfig</code>	42
4.5.20	<code>CLOCK_GetCurSysClkConfig</code>	42
4.5.21	<code>CLOCK_SetClkOutSel</code>	42
4.5.22	<code>CLOCK_InitSysOsc</code>	42
4.5.23	<code>CLOCK_DeinitSysOsc</code>	43
4.5.24	<code>CLOCK_SetSysOscAsyncClkDiv</code>	43
4.5.25	<code>CLOCK_GetSysOscFreq</code>	44
4.5.26	<code>CLOCK_GetSysOscAsyncFreq</code>	44
4.5.27	<code>CLOCK_IsSysOscErr</code>	44
4.5.28	<code>CLOCK_SetSysOscMonitorMode</code>	44
4.5.29	<code>CLOCK_IsSysOscValid</code>	45
4.5.30	<code>CLOCK_InitSirc</code>	45
4.5.31	<code>CLOCK_DeinitSirc</code>	45
4.5.32	<code>CLOCK_SetSircAsyncClkDiv</code>	46
4.5.33	<code>CLOCK_EnableLpoPowerOption</code>	46
4.5.34	<code>CLOCK_GetSircFreq</code>	46
4.5.35	<code>CLOCK_GetSircAsyncFreq</code>	46
4.5.36	<code>CLOCK_IsSircValid</code>	47
4.5.37	<code>CLOCK_InitFirc</code>	47
4.5.38	<code>CLOCK_DeinitFirc</code>	47
4.5.39	<code>CLOCK_SetFircAsyncClkDiv</code>	48
4.5.40	<code>CLOCK_GetFircFreq</code>	48
4.5.41	<code>CLOCK_GetFircAsyncFreq</code>	48
4.5.42	<code>CLOCK_IsFircErr</code>	49

Section No.	Title	Page No.
4.5.43	<code>CLOCK_IsFircValid</code>	49
4.5.44	<code>CLOCK_GetRtcOscFreq</code>	49
4.5.45	<code>CLOCK_IsRtcOscErr</code>	49
4.5.46	<code>CLOCK_SetRtcOscMonitorMode</code>	49
4.5.47	<code>CLOCK_IsRtcOscValid</code>	50
4.6	Variable Documentation	50
4.6.1	<code>g_xtal0Freq</code>	50
4.6.2	<code>g_xtal32Freq</code>	50
4.6.3	<code>g_lvdsFreq</code>	50

Chapter 5 IOMUXC: IOMUX Controller

5.1	System Clock Generator (SCG)	52
5.1.1	Function description	52
5.1.2	Typical use case	54

Chapter 6 ACMP: Analog Comparator Driver

6.1	Overview	56
6.2	Typical use case	56
6.2.1	Normal Configuration	56
6.2.2	Interrupt Configuration	56
6.2.3	Round robin Configuration	56
6.3	Data Structure Documentation	59
6.3.1	<code>struct acmp_config_t</code>	59
6.3.2	<code>struct acmp_channel_config_t</code>	59
6.3.3	<code>struct acmp_filter_config_t</code>	60
6.3.4	<code>struct acmp_dac_config_t</code>	60
6.3.5	<code>struct acmp_round_robin_config_t</code>	61
6.4	Macro Definition Documentation	61
6.4.1	<code>FSL_ACMP_DRIVER_VERSION</code>	61
6.4.2	<code>CMP_C0_CFx_MASK</code>	61
6.5	Enumeration Type Documentation	62
6.5.1	<code>_acmp_interrupt_enable</code>	62
6.5.2	<code>_acmp_status_flags</code>	62
6.5.3	<code>acmp_offset_mode_t</code>	62
6.5.4	<code>acmp_hysteresis_mode_t</code>	62
6.5.5	<code>acmp_reference_voltage_source_t</code>	63
6.5.6	<code>acmp_port_input_t</code>	63
6.5.7	<code>acmp_fixed_port_t</code>	63

Section No.	Title	Page No.
6.6 Function Documentation		63
6.6.1 ACMP_Init		63
6.6.2 ACMP_Deinit		63
6.6.3 ACMP_GetDefaultConfig		64
6.6.4 ACMP_Enable		64
6.6.5 ACMP_SetChannelConfig		64
6.6.6 ACMP_EnableDMA		65
6.6.7 ACMP_EnableWindowMode		65
6.6.8 ACMP_SetFilterConfig		65
6.6.9 ACMP_SetDACConfig		66
6.6.10 ACMP_SetRoundRobinConfig		66
6.6.11 ACMP_SetRoundRobinPreState		66
6.6.12 ACMP_GetRoundRobinStatusFlags		67
6.6.13 ACMP_ClearRoundRobinStatusFlags		67
6.6.14 ACMP_GetRoundRobinResult		67
6.6.15 ACMP_EnableInterrupts		68
6.6.16 ACMP_DisableInterrupts		69
6.6.17 ACMP_GetStatusFlags		69
6.6.18 ACMP_ClearStatusFlags		69

Chapter 7 CACHE: LMEM CACHE Memory Controller

7.1 Overview		70
7.2 Function groups		70
7.2.1 L1 CACHE Operation		70
7.3 Macro Definition Documentation		71
7.3.1 FSL_CACHE_DRIVER_VERSION		71
7.3.2 L1CODEBUSCACHE_LINESIZE_BYTE		71
7.3.3 L1SYSTEMBUSCACHE_LINESIZE_BYTE		71
7.4 Function Documentation		71
7.4.1 ICACHE_InvalidateByRange		71
7.4.2 DCACHE_InvalidateByRange		71
7.4.3 DCACHE_CleanByRange		72
7.4.4 DCACHE_CleanInvalidateByRange		72

Chapter 8 Common Driver

8.1 Overview		73
8.2 Macro Definition Documentation		76
8.2.1 FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ		76
8.2.2 MAKE_STATUS		76

Section No.	Title	Page No.
8.2.3	MAKE_VERSION	76
8.2.4	FSL_COMMON_DRIVER_VERSION	77
8.2.5	DEBUG_CONSOLE_DEVICE_TYPE_NONE	77
8.2.6	DEBUG_CONSOLE_DEVICE_TYPE_UART	77
8.2.7	DEBUG_CONSOLE_DEVICE_TYPE_LPUART	77
8.2.8	DEBUG_CONSOLE_DEVICE_TYPE_LPSCI	77
8.2.9	DEBUG_CONSOLE_DEVICE_TYPE_USBCDC	77
8.2.10	DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM	77
8.2.11	DEBUG_CONSOLE_DEVICE_TYPE_IUART	77
8.2.12	DEBUG_CONSOLE_DEVICE_TYPE_VUSART	77
8.2.13	DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART	77
8.2.14	DEBUG_CONSOLE_DEVICE_TYPE_SWO	77
8.2.15	DEBUG_CONSOLE_DEVICE_TYPE_QSCI	77
8.2.16	ARRAY_SIZE	77
8.3	Typedef Documentation	77
8.3.1	status_t	77
8.4	Enumeration Type Documentation	77
8.4.1	_status_groups	77
8.4.2	anonymous enum	80
8.5	Function Documentation	80
8.5.1	SDK_Malloc	80
8.5.2	SDK_Free	81
8.5.3	SDK_DelayAtLeastUs	81
 Chapter 9 CRC: Cyclic Redundancy Check Driver		
9.1	Overview	82
9.2	CRC Driver Initialization and Configuration	82
9.3	CRC Write Data	82
9.4	CRC Get Checksum	82
9.5	Comments about API usage in RTOS	83
9.6	Data Structure Documentation	84
9.6.1	struct crc_config_t	84
9.7	Macro Definition Documentation	85
9.7.1	FSL_CRC_DRIVER_VERSION	85
9.7.2	CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT	85

Section No.	Title	Page No.
9.8 Enumeration Type Documentation		85
9.8.1 <code>crc_bits_t</code>		85
9.8.2 <code>crc_result_t</code>		85
9.9 Function Documentation		85
9.9.1 <code>CRC_Init</code>		85
9.9.2 <code>CRC_Deinit</code>		86
9.9.3 <code>CRC_GetDefaultConfig</code>		86
9.9.4 <code>CRC_WriteData</code>		86
9.9.5 <code>CRC_Get32bitResult</code>		87
9.9.6 <code>CRC_Get16bitResult</code>		87
 Chapter 10 DAC12: 12-bit Digital-to-Analog Converter Driver		
10.1 Overview		88
10.2 Typical use case		88
10.2.1 A simple use case to output the user-defined DAC12 value.		88
10.2.2 Working with the trigger		88
10.3 Data Structure Documentation		91
10.3.1 <code>struct dac12_hardware_info_t</code>		91
10.3.2 <code>struct dac12_config_t</code>		91
10.4 Macro Definition Documentation		92
10.4.1 <code>FSL_DAC12_DRIVER_VERSION</code>		92
10.4.2 <code>DAC12_CR_W1C_FLAGS_MASK</code>		92
10.4.3 <code>DAC12_CR_ALL_FLAGS_MASK</code>		92
10.5 Enumeration Type Documentation		92
10.5.1 <code>_dac12_status_flags</code>		92
10.5.2 <code>_dac12_interrupt_enable</code>		93
10.5.3 <code>dac12_fifo_size_info_t</code>		93
10.5.4 <code>dac12_fifo_work_mode_t</code>		93
10.5.5 <code>dac12_reference_voltage_source_t</code>		93
10.5.6 <code>dac12_fifo_trigger_mode_t</code>		94
10.5.7 <code>dac12_reference_current_source_t</code>		94
10.5.8 <code>dac12_speed_mode_t</code>		94
10.6 Function Documentation		94
10.6.1 <code>DAC12_GetHardwareInfo</code>		94
10.6.2 <code>DAC12_Init</code>		95
10.6.3 <code>DAC12_GetDefaultConfig</code>		95
10.6.4 <code>DAC12_Deinit</code>		95
10.6.5 <code>DAC12_Enable</code>		95
10.6.6 <code>DAC12_ResetConfig</code>		96

Section No.	Title	Page No.
10.6.7	DAC12_ResetFIFO	96
10.6.8	DAC12_GetStatusFlags	96
10.6.9	DAC12_ClearStatusFlags	96
10.6.10	DAC12_EnableInterrupts	97
10.6.11	DAC12_DisableInterrupts	97
10.6.12	DAC12_EnableDMA	97
10.6.13	DAC12_SetData	97
10.6.14	DAC12_DoSoftwareTrigger	98
10.6.15	DAC12_GetFIFOReadPointer	98
10.6.16	DAC12_GetFIFOWritePointer	98

Chapter 11 DMAMUX: Direct Memory Access Multiplexer Driver

11.1	Overview	99
11.2	Typical use case	99
11.2.1	DMAMUX Operation	99
11.3	Macro Definition Documentation	99
11.3.1	FSL_DMAMUX_DRIVER_VERSION	99
11.4	Function Documentation	99
11.4.1	DMAMUX_Init	99
11.4.2	DMAMUX_Deinit	100
11.4.3	DMAMUX_EnableChannel	100
11.4.4	DMAMUX_DisableChannel	100
11.4.5	DMAMUX_SetSource	101

Chapter 12 eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

12.1	Overview	102
12.2	Typical use case	102
12.2.1	eDMA Operation	102
12.3	Data Structure Documentation	107
12.3.1	struct edma_config_t	107
12.3.2	struct edma_transfer_config_t	108
12.3.3	struct edma_channel_Preemption_config_t	109
12.3.4	struct edma_minor_offset_config_t	109
12.3.5	struct edma_tcd_t	110
12.3.6	struct edma_handle_t	111
12.4	Macro Definition Documentation	112
12.4.1	FSL_EDMA_DRIVER_VERSION	112

Section No.	Title	Page No.
12.5 Typedef Documentation	112	
12.5.1 edma_callback	112	
12.6 Enumeration Type Documentation	113	
12.6.1 edma_transfer_size_t	113	
12.6.2 edma_modulo_t	113	
12.6.3 edma_bandwidth_t	114	
12.6.4 edma_channel_link_type_t	114	
12.6.5 anonymous enum	114	
12.6.6 anonymous enum	114	
12.6.7 edma_interrupt_enable_t	115	
12.6.8 edma_transfer_type_t	115	
12.6.9 anonymous enum	115	
12.7 Function Documentation	115	
12.7.1 EDMA_Init	115	
12.7.2 EDMA_Deinit	116	
12.7.3 EDMA_InstallTCD	116	
12.7.4 EDMA_GetDefaultConfig	116	
12.7.5 EDMA_EnableContinuousChannelLinkMode	117	
12.7.6 EDMA_EnableMinorLoopMapping	117	
12.7.7 EDMA_ResetChannel	117	
12.7.8 EDMA_SetTransferConfig	118	
12.7.9 EDMA_SetMinorOffsetConfig	118	
12.7.10 EDMA_SetChannelPreemptionConfig	119	
12.7.11 EDMA_SetChannelLink	119	
12.7.12 EDMA_SetBandWidth	120	
12.7.13 EDMA_SetModulo	120	
12.7.14 EDMA_EnableAutoStopRequest	121	
12.7.15 EDMA_EnableChannelInterrupts	121	
12.7.16 EDMA_DisableChannelInterrupts	121	
12.7.17 EDMA_SetMajorOffsetConfig	122	
12.7.18 EDMA_TcdReset	122	
12.7.19 EDMA_TcdSetTransferConfig	122	
12.7.20 EDMA_TcdSetMinorOffsetConfig	123	
12.7.21 EDMA_TcdSetChannelLink	123	
12.7.22 EDMA_TcdSetBandWidth	124	
12.7.23 EDMA_TcdSetModulo	124	
12.7.24 EDMA_TcdEnableAutoStopRequest	125	
12.7.25 EDMA_TcdEnableInterrupts	125	
12.7.26 EDMA_TcdDisableInterrupts	125	
12.7.27 EDMA_TcdSetMajorOffsetConfig	125	
12.7.28 EDMA_EnableChannelRequest	126	
12.7.29 EDMA_DisableChannelRequest	126	
12.7.30 EDMA_TriggerChannelStart	126	

Section No.	Title	Page No.
12.7.31	<code>EDMA_GetRemainingMajorLoopCount</code>	127
12.7.32	<code>EDMA_GetErrorStatusFlags</code>	127
12.7.33	<code>EDMA_GetChannelStatusFlags</code>	128
12.7.34	<code>EDMA_ClearChannelStatusFlags</code>	128
12.7.35	<code>EDMA_CreateHandle</code>	128
12.7.36	<code>EDMA_InstallTCDMemory</code>	129
12.7.37	<code>EDMA_SetCallback</code>	129
12.7.38	<code>EDMA_PrepTransferConfig</code>	130
12.7.39	<code>EDMA_PrepTransfer</code>	131
12.7.40	<code>EDMA_SubmitTransfer</code>	132
12.7.41	<code>EDMA_StartTransfer</code>	132
12.7.42	<code>EDMA_StopTransfer</code>	132
12.7.43	<code>EDMA_AbortTransfer</code>	133
12.7.44	<code>EDMA_GetUnusedTCDNumber</code>	133
12.7.45	<code>EDMA_GetNextTCDAddress</code>	133
12.7.46	<code>EDMA_HandleIRQ</code>	134

Chapter 13 EWM: External Watchdog Monitor Driver

13.1	Overview	135
13.2	Typical use case	135
13.3	Data Structure Documentation	136
13.3.1	<code>struct ewm_config_t</code>	136
13.4	Macro Definition Documentation	136
13.4.1	<code>FSL_EWM_DRIVER_VERSION</code>	136
13.5	Enumeration Type Documentation	136
13.5.1	<code>_ewm_interrupt_enable_t</code>	136
13.5.2	<code>_ewm_status_flags_t</code>	136
13.6	Function Documentation	137
13.6.1	<code>EWM_Init</code>	137
13.6.2	<code>EWM_Deinit</code>	137
13.6.3	<code>EWM_GetDefaultConfig</code>	137
13.6.4	<code>EWM_EnableInterrupts</code>	138
13.6.5	<code>EWM_DisableInterrupts</code>	138
13.6.6	<code>EWM_GetStatusFlags</code>	138
13.6.7	<code>EWM_Refresh</code>	139

Chapter 14 FlexIO: FlexIO Driver

14.1	Overview	140
-------------	-----------------	------------

Section No.	Title	Page No.
14.2 FlexIO Driver		141
14.2.1 Overview		141
14.2.2 Data Structure Documentation		145
14.2.3 Macro Definition Documentation		148
14.2.4 Typedef Documentation		148
14.2.5 Enumeration Type Documentation		148
14.2.6 Function Documentation		153
14.2.7 Variable Documentation		162
14.3 FlexIO I2C Master Driver		163
14.3.1 Overview		163
14.3.2 Typical use case		163
14.3.3 Data Structure Documentation		167
14.3.4 Macro Definition Documentation		170
14.3.5 Typedef Documentation		170
14.3.6 Enumeration Type Documentation		170
14.3.7 Function Documentation		171
14.4 FlexIO I2S Driver		181
14.4.1 Overview		181
14.4.2 Typical use case		181
14.4.3 Data Structure Documentation		186
14.4.4 Macro Definition Documentation		189
14.4.5 Enumeration Type Documentation		189
14.4.6 Function Documentation		190
14.4.7 FlexIO eDMA I2S Driver		202
14.5 FlexIO SPI Driver		209
14.5.1 Overview		209
14.5.2 Typical use case		209
14.5.3 Data Structure Documentation		216
14.5.4 Macro Definition Documentation		220
14.5.5 Typedef Documentation		220
14.5.6 Enumeration Type Documentation		220
14.5.7 Function Documentation		222
14.5.8 FlexIO eDMA SPI Driver		237
14.6 FlexIO UART Driver		245
14.6.1 Overview		245
14.6.2 Typical use case		245
14.6.3 Data Structure Documentation		254
14.6.4 Macro Definition Documentation		257
14.6.5 Typedef Documentation		257
14.6.6 Enumeration Type Documentation		257
14.6.7 Function Documentation		258

Section No.	Title	Page No.
14.6.8	FlexIO eDMA UART Driver	269
14.7	FlexIO Camera Driver	275
14.7.1	Overview	275
14.7.2	Typical use case	275
14.7.3	Data Structure Documentation	278
14.7.4	Macro Definition Documentation	279
14.7.5	Enumeration Type Documentation	279
14.7.6	Function Documentation	280
14.7.7	FlexIO eDMA Camera Driver	284

Chapter 15 GPIO: General-Purpose Input/Output Driver

15.1	Overview	288
15.2	Data Structure Documentation	288
15.2.1	struct gpio_pin_config_t	288
15.3	Macro Definition Documentation	289
15.3.1	FSL_GPIO_DRIVER_VERSION	289
15.4	Enumeration Type Documentation	289
15.4.1	gpio_pin_direction_t	289
15.5	GPIO Driver	290
15.5.1	Overview	290
15.5.2	Typical use case	290
15.5.3	Function Documentation	291
15.6	FGPIO Driver	294
15.6.1	Overview	294
15.6.2	Typical use case	294
15.6.3	Function Documentation	295

Chapter 16 LLWU: Low-Leakage Wakeup Unit Driver

16.1	Overview	298
16.2	External wakeup pins configurations	298
16.3	Internal wakeup modules configurations	298
16.4	Digital pin filter for external wakeup pin configurations	298
16.5	Macro Definition Documentation	299
16.5.1	FSL_LLWU_DRIVER_VERSION	299

Section No.	Title	Page No.
16.6 Enumeration Type Documentation		299
16.6.1 llwu_external_pin_mode_t		299
16.6.2 llwu_pin_filter_mode_t		299
 Chapter 17 LPADC: 12-bit SAR Analog-to-Digital Converter Driver		
17.1 Overview		300
17.2 Typical use case		300
17.2.1 Polling Configuration		300
17.2.2 Interrupt Configuration		300
17.3 Data Structure Documentation		303
17.3.1 struct lpadc_config_t		303
17.3.2 struct lpadc_conv_command_config_t		304
17.3.3 struct lpadc_conv_trigger_config_t		306
17.3.4 struct lpadc_conv_result_t		306
17.4 Macro Definition Documentation		307
17.4.1 FSL_LPADC_DRIVER_VERSION		307
17.4.2 LPADC_GET_ACTIVE_COMMAND_STATUS		307
17.4.3 LPADC_GET_ACTIVE_TRIGGER_STATUE		307
17.5 Enumeration Type Documentation		307
17.5.1 _lpadc_status_flags		307
17.5.2 _lpadc_interrupt_enable		307
17.5.3 lpadc_sample_scale_mode_t		308
17.5.4 lpadc_sample_channel_mode_t		308
17.5.5 lpadc_hardware_average_mode_t		308
17.5.6 lpadc_sample_time_mode_t		308
17.5.7 lpadc_hardware_compare_mode_t		309
17.5.8 lpadc_reference_voltage_source_t		309
17.5.9 lpadc_power_level_mode_t		309
17.5.10 lpadc_trigger_priority_policy_t		310
17.6 Function Documentation		310
17.6.1 LPADC_Init		310
17.6.2 LPADC_GetDefaultConfig		310
17.6.3 LPADC_Deinit		311
17.6.4 LPADC_Enable		311
17.6.5 LPADC_DoResetFIFO		311
17.6.6 LPADC_DoResetConfig		311
17.6.7 LPADC_GetStatusFlags		312
17.6.8 LPADC_ClearStatusFlags		312
17.6.9 LPADC_EnableInterrupts		312

Section No.	Title	Page No.
17.6.10	LPADC_DisableInterrupts	312
17.6.11	LPADC_EnableFIFOWatermarkDMA	313
17.6.12	LPADC_GetConvResultCount	313
17.6.13	LPADC_GetConvResult	313
17.6.14	LPADC_SetConvTriggerConfig	314
17.6.15	LPADC_GetDefaultConvTriggerConfig	315
17.6.16	LPADC_DoSoftwareTrigger	315
17.6.17	LPADC_SetConvCommandConfig	315
17.6.18	LPADC_GetDefaultConvCommandConfig	316

Chapter 18 LPI2C: Low Power Inter-Integrated Circuit Driver

18.1	Overview	317
18.2	Macro Definition Documentation	318
18.2.1	FSL_LPI2C_DRIVER_VERSION	318
18.2.2	I2C_RETRY_TIMES	318
18.3	Enumeration Type Documentation	318
18.3.1	anonymous enum	318
18.4	LPI2C Master Driver	319
18.4.1	Overview	319
18.4.2	Data Structure Documentation	322
18.4.3	Typedef Documentation	326
18.4.4	Enumeration Type Documentation	327
18.4.5	Function Documentation	329
18.5	LPI2C Slave Driver	343
18.5.1	Overview	343
18.5.2	Data Structure Documentation	345
18.5.3	Typedef Documentation	349
18.5.4	Enumeration Type Documentation	350
18.5.5	Function Documentation	351
18.6	LPI2C Master DMA Driver	360
18.6.1	Overview	360
18.6.2	Data Structure Documentation	360
18.6.3	Typedef Documentation	362
18.6.4	Function Documentation	363
18.7	LPI2C FreeRTOS Driver	366
18.7.1	Overview	366
18.7.2	Macro Definition Documentation	366
18.7.3	Function Documentation	366

Section No.	Title	Page No.
18.8 LPI2C CMSIS Driver		369
18.8.1 LPI2C CMSIS Driver		369
 Chapter 19 LPIT: Low-Power Interrupt Timer		
19.1 Overview		371
19.2 Function groups		371
19.2.1 Initialization and deinitialization		371
19.2.2 Timer period Operations		371
19.2.3 Start and Stop timer operations		371
19.2.4 Status		372
19.2.5 Interrupt		372
19.3 Typical use case		372
19.3.1 LPIT tick example		372
19.4 Data Structure Documentation		374
19.4.1 struct lpit_chnl_params_t		374
19.4.2 struct lpit_config_t		375
19.5 Enumeration Type Documentation		375
19.5.1 lpit_chnl_t		375
19.5.2 lpit_timer_modes_t		375
19.5.3 lpit_trigger_select_t		376
19.5.4 lpit_trigger_source_t		376
19.5.5 lpit_interrupt_enable_t		376
19.5.6 lpit_status_flags_t		377
19.6 Function Documentation		377
19.6.1 LPIT_Init		377
19.6.2 LPIT_Deinit		377
19.6.3 LPIT_GetDefaultConfig		378
19.6.4 LPIT_SetupChannel		378
19.6.5 LPIT_EnableInterrupts		378
19.6.6 LPIT_DisableInterrupts		379
19.6.7 LPIT_GetEnabledInterrupts		379
19.6.8 LPIT_GetStatusFlags		379
19.6.9 LPIT_ClearStatusFlags		379
19.6.10 LPIT_SetTimerPeriod		380
19.6.11 LPIT_SetTimerValue		380
19.6.12 LPIT_GetCurrentTimerCount		381
19.6.13 LPIT_StartTimer		381
19.6.14 LPIT_StopTimer		381
19.6.15 LPIT_Reset		382

Section No.	Title	Page No.
Chapter 20 LPSPI: Low Power Serial Peripheral Interface		
20.1 Overview		383
20.2 LPSPI Peripheral driver		384
20.2.1 Overview		384
20.2.2 Function groups		384
20.2.3 Typical use case		384
20.2.4 Data Structure Documentation		391
20.2.5 Macro Definition Documentation		398
20.2.6 Typedef Documentation		399
20.2.7 Enumeration Type Documentation		400
20.2.8 Function Documentation		405
20.2.9 Variable Documentation		421
20.3 LPSPI eDMA Driver		422
20.3.1 Overview		422
20.3.2 Data Structure Documentation		423
20.3.3 Macro Definition Documentation		428
20.3.4 Typedef Documentation		428
20.3.5 Function Documentation		429
20.4 LPSPI FreeRTOS Driver		436
20.4.1 Overview		436
20.4.2 Macro Definition Documentation		436
20.4.3 Function Documentation		436
20.5 LPSPI CMSIS Driver		439
20.5.1 Function groups		439
20.5.2 Typical use case		440
Chapter 21 LPTMR: Low-Power Timer		
21.1 Overview		441
21.2 Function groups		441
21.2.1 Initialization and deinitialization		441
21.2.2 Timer period Operations		441
21.2.3 Start and Stop timer operations		441
21.2.4 Status		442
21.2.5 Interrupt		442
21.3 Typical use case		442
21.3.1 LPTMR tick example		442
21.4 Data Structure Documentation		444

Section No.	Title	Page No.
21.4.1	struct lptmr_config_t	444
21.5	Enumeration Type Documentation	445
21.5.1	lptmr_pin_select_t	445
21.5.2	lptmr_pin_polarity_t	445
21.5.3	lptmr_timer_mode_t	445
21.5.4	lptmr_prescaler_glitch_value_t	445
21.5.5	lptmr_prescaler_clock_select_t	446
21.5.6	lptmr_interrupt_enable_t	446
21.5.7	lptmr_status_flags_t	446
21.6	Function Documentation	446
21.6.1	LPTMR_Init	446
21.6.2	LPTMR_Deinit	447
21.6.3	LPTMR_GetDefaultConfig	447
21.6.4	LPTMR_EnableInterrupts	447
21.6.5	LPTMR_DisableInterrupts	448
21.6.6	LPTMR_GetEnabledInterrupts	448
21.6.7	LPTMR_EnableTimerDMA	448
21.6.8	LPTMR_GetStatusFlags	448
21.6.9	LPTMR_ClearStatusFlags	449
21.6.10	LPTMR_SetTimerPeriod	449
21.6.11	LPTMR_GetCurrentTimerCount	449
21.6.12	LPTMR_StartTimer	450
21.6.13	LPTMR_StopTimer	450

Chapter 22 LPUART: Low Power Universal Asynchronous Receiver/Transmitter Driver

22.1	Overview	451
22.2	LPUART Driver	452
22.2.1	Overview	452
22.2.2	Typical use case	452
22.2.3	Data Structure Documentation	457
22.2.4	Macro Definition Documentation	461
22.2.5	Typedef Documentation	461
22.2.6	Enumeration Type Documentation	461
22.2.7	Function Documentation	465
22.3	LPUART eDMA Driver	482
22.3.1	Overview	482
22.3.2	Data Structure Documentation	483
22.3.3	Macro Definition Documentation	484
22.3.4	Typedef Documentation	484
22.3.5	Function Documentation	484

Section No.	Title	Page No.
22.4 LPUART FreeRTOS Driver		489
22.4.1 Overview		489
22.4.2 Data Structure Documentation		489
22.4.3 Macro Definition Documentation		490
22.4.4 Function Documentation		490
22.5 LPUART CMSIS Driver		493
22.5.1 Function groups		493
 Chapter 23 LTC: LP Trusted Cryptography		
23.1 Overview		495
23.2 LTC Driver Initialization and Configuration		495
23.3 Comments about API usage in RTOS		495
23.4 Comments about API usage in interrupt handler		495
23.5 LTC Driver Examples		496
23.5.1 Simple examples		496
23.6 Macro Definition Documentation		496
23.6.1 FSL_LTC_DRIVER_VERSION		496
23.7 Function Documentation		497
23.7.1 LTC_Init		497
23.7.2 LTC_Deinit		497
23.8 LTC Blocking APIs		499
23.8.1 Overview		499
23.8.2 LTC DES driver		500
23.8.3 LTC AES driver		501
23.8.4 LTC HASH driver		508
23.8.5 LTC PKHA driver		512
23.9 LTC Non-blocking eDMA APIs		514
23.9.1 Overview		514
23.9.2 Data Structure Documentation		514
23.9.3 Macro Definition Documentation		516
23.9.4 Typedef Documentation		517
23.9.5 Function Documentation		517
23.9.6 LTC eDMA DES driver		518
23.9.7 LTC eDMA AES driver		519

Section No.	Title	Page No.
Chapter 24 MSMC: Multicore System Mode Controller		
24.1	Overview	525
24.2	Typical use case	525
24.2.1	Set Core 0 from RUN to VLPR mode	525
24.2.2	Set Core 0 from VLPR/HSRUN to RUN mode	525
24.3	Typical use case	525
24.3.1	Set Core 0 from RUN to HSRUN mode	525
24.3.2	Enter wait or stop modes	525
24.4	Data Structure Documentation	529
24.4.1	<code>struct smc_reset_pin_filter_config_t</code>	529
24.5	Macro Definition Documentation	529
24.5.1	<code>FSL_MSMC_DRIVER_VERSION</code>	529
24.6	Enumeration Type Documentation	529
24.6.1	<code>smc_power_mode_protection_t</code>	529
24.6.2	<code>smc_power_state_t</code>	529
24.6.3	<code>smc_power_stop_entry_status_t</code>	530
24.6.4	<code>smc_run_mode_t</code>	530
24.6.5	<code>smc_stop_mode_t</code>	530
24.6.6	<code>smc_partial_stop_option_t</code>	530
24.6.7	<code>anonymous enum</code>	531
24.6.8	<code>smc_reset_source_t</code>	531
24.6.9	<code>smc_interrupt_enable_t</code>	531
24.7	Function Documentation	532
24.7.1	<code>SMC_SetPowerModeProtection</code>	532
24.7.2	<code>SMC_GetPowerModeState</code>	532
24.7.3	<code>SMC_PreEnterStopModes</code>	532
24.7.4	<code>SMC_PostExitStopModes</code>	533
24.7.5	<code>SMC_PreEnterWaitModes</code>	533
24.7.6	<code>SMC_PostExitWaitModes</code>	533
24.7.7	<code>SMC_SetPowerModeRun</code>	533
24.7.8	<code>SMC_SetPowerModeHsrun</code>	533
24.7.9	<code>SMC_SetPowerModeWait</code>	533
24.7.10	<code>SMC_SetPowerModeStop</code>	534
24.7.11	<code>SMC_SetPowerModeVlpr</code>	534
24.7.12	<code>SMC_SetPowerModeVlpw</code>	534
24.7.13	<code>SMC_SetPowerModeVlps</code>	535
24.7.14	<code>SMC_SetPowerModeLls</code>	535
24.7.15	<code>SMC_SetPowerModeVlls</code>	535
24.7.16	<code>SMC_GetPreviousResetSources</code>	536

Section No.	Title	Page No.
24.7.17	SMC_GetStickyResetSources	536
24.7.18	SMC_ClearStickyResetSources	537
24.7.19	SMC_ConfigureResetPinFilter	537
24.7.20	SMC_SetSystemResetInterruptConfig	538
24.7.21	SMC_GetResetInterruptSourcesStatus	538
24.7.22	SMC_ClearResetInterruptSourcesStatus	539
24.7.23	SMC_GetBootOptionConfig	539

Chapter 25 MU: Messaging Unit

25.1	Overview	541
25.2	Function description	541
25.2.1	MU initialization	541
25.2.2	MU message	541
25.2.3	MU flags	542
25.2.4	Status and interrupt	542
25.2.5	MU misc functions	542
25.3	Macro Definition Documentation	545
25.3.1	FSL_MU_DRIVER_VERSION	545
25.4	Enumeration Type Documentation	545
25.4.1	_mu_status_flags	545
25.4.2	_mu_interrupt_enable	545
25.4.3	_mu_interrupt_trigger	546
25.5	Function Documentation	546
25.5.1	MU_Init	546
25.5.2	MU_Deinit	546
25.5.3	MU_SendMsgNonBlocking	546
25.5.4	MU_SendMsg	547
25.5.5	MU_ReceiveMsgNonBlocking	547
25.5.6	MU_ReceiveMsg	548
25.5.7	MU_SetFlagsNonBlocking	548
25.5.8	MU_SetFlags	549
25.5.9	MU_GetFlags	549
25.5.10	MU_GetStatusFlags	549
25.5.11	MU_GetInterruptsPending	550
25.5.12	MU_ClearStatusFlags	550
25.5.13	MU_EnableInterrupts	551
25.5.14	MU_DisableInterrupts	551
25.5.15	MU_TriggerInterrupts	552
25.5.16	MU_ClearNmi	552
25.5.17	MU_BootCoreB	552

Section No.	Title	Page No.
25.5.18	MU_HoldCoreBReset	553
25.5.19	MU_BootOtherCore	553
25.5.20	MU_HoldOtherCoreReset	553
25.5.21	MU_ResetBothSides	554
25.5.22	MU_HardwareResetOtherCore	554
25.5.23	MU_SetClockOnOtherCoreEnable	555
25.5.24	MU_GetOtherCorePowerMode	555

Chapter 26 PMC0: Power Management Controller

26.1	Overview	556
26.2	Typical use case	556
26.2.1	Turn on the PMC 1 using LDO Regulator	556
26.2.2	Turn on the PMC 1 using the PMIC	556
26.2.3	Turn off the LDO Regulator	557
26.2.4	Turn on the LDO Regulator	557
26.2.5	Change the Core Regulator voltage level in PMC 0 RUN or HSRUN mode	557
26.2.6	Change the SRAMs power mode during PMC 0 RUN mode	557
26.3	Data Structure Documentation	561
26.3.1	struct pmc0_hsrn_mode_config_t	561
26.3.2	struct pmc0_rn_mode_config_t	562
26.3.3	struct pmc0_vlpr_mode_config_t	562
26.3.4	struct pmc0_stop_mode_config_t	564
26.3.5	struct pmc0_vlps_mode_config_t	564
26.3.6	struct pmc0_lls_mode_config_t	566
26.3.7	struct pmc0_vlls_mode_config_t	567
26.3.8	struct pmc0_bias_config_t	568
26.4	Enumeration Type Documentation	569
26.4.1	pmc0_high_volt_detect_monitor_select_t	569
26.4.2	pmc0_low_volt_detect_monitor_select_t	569
26.4.3	pmc0_core_regulator_select_t	569
26.4.4	pmc0_array_regulator_select_t	569
26.4.5	pmc0_vlls_array_regulator_select_t	570
26.4.6	pmc0_fbb_p_well_voltage_level_select_t	570
26.4.7	pmc0_fbb_n_well_voltage_level_select_t	570
26.4.8	pmc0_rbb_p_well_voltage_level_select_t	571
26.4.9	pmc0_rbb_n_well_voltage_level_select_t	571
26.4.10	_pmc0_status_flags	571
26.4.11	_pmc0_power_mode_status_flags	572
26.5	Function Documentation	572
26.5.1	PMC0_ConfigureHsrnMode	572

Section No.	Title	Page No.
26.5.2	PMC0_ConfigureRunMode	572
26.5.3	PMC0_ConfigureVlprMode	573
26.5.4	PMC0_ConfigureStopMode	573
26.5.5	PMC0_ConfigureVlpsMode	573
26.5.6	PMC0_ConfigureLlsMode	573
26.5.7	PMC0_ConfigureVllsMode	575
26.5.8	PMC0_GetPMC0PowerModeStatusFlags	575
26.5.9	PMC0_GetPMC0PowerTransitionStatus	575
26.5.10	PMC0_GetPMC1PowerModeStatusFlags	576
26.5.11	PMC0_GetPMC1PowerTransitionStatus	576
26.5.12	PMC0_GetStatusFlags	576
26.5.13	PMC0_EnableLowVoltDetectInterrupt	576
26.5.14	PMC0_DisableLowVoltDetectInterrupt	577
26.5.15	PMC0_ClearLowVoltDetectFlag	577
26.5.16	PMC0_EnableHighVoltDetectInterrupt	577
26.5.17	PMC0_DisableHighVoltDetectInterrupt	577
26.5.18	PMC0_ClearHighVoltDetectFlag	577
26.5.19	PMC0_EnableLowVoltDetectReset	577
26.5.20	PMC0_EnableHighVoltDetectReset	578
26.5.21	PMC0_ClearPadsIsolation	579
26.5.22	PMC0_PowerOnPmc1	579
26.5.23	PMC0_EnableWaitLdoOkSignal	579
26.5.24	PMC0_EnablePmc1LdoRegulator	579
26.5.25	PMC0_EnablePmc1RBBMode	580
26.5.26	PMC0_SetBiasConfig	580
26.5.27	PMC0_ConfigureSramBankPowerDown	580
26.5.28	PMC0_ConfigureSramBankPowerDownStopMode	580
26.5.29	PMC0_ConfigureSramBankPowerDownStandbyMode	581
26.5.30	PMC0_EnableTemperatureSensor	581
26.5.31	PMC0_SetTemperatureSensorMode	582

Chapter 27 PORT: Port Control and Interrupts

27.1	Overview	584
27.2	Data Structure Documentation	586
27.2.1	struct port_digital_filter_config_t	586
27.2.2	struct port_pin_config_t	586
27.3	Macro Definition Documentation	587
27.3.1	FSL_PORT_DRIVER_VERSION	587
27.4	Enumeration Type Documentation	587
27.4.1	_port_pull	587
27.4.2	_port_passive_filter_enable	587

Section No.	Title	Page No.
27.4.3	_port_drive_strength	587
27.4.4	_port_lock_register	587
27.4.5	port_mux_t	588
27.4.6	port_interrupt_t	588
27.4.7	port_digital_filter_clock_source_t	589
27.5	Function Documentation	589
27.5.1	PORT_SetPinConfig	589
27.5.2	PORT_SetMultiplePinsConfig	589
27.5.3	PORT_SetPinMux	590
27.5.4	PORT_EnablePinsDigitalFilter	590
27.5.5	PORT_SetDigitalFilterConfig	591
27.5.6	PORT_SetPinInterruptConfig	592
27.5.7	PORT_SetPinDriveStrength	592
27.5.8	PORT_GetPinsInterruptFlags	593
27.5.9	PORT_ClearPinsInterruptFlags	593

Chapter 28 QSPI: Quad Serial Peripheral Interface

28.1	Overview	594
28.2	Quad Serial Peripheral Interface Driver	595
28.2.1	Overview	595
28.2.2	Data Structure Documentation	600
28.2.3	Macro Definition Documentation	602
28.2.4	Enumeration Type Documentation	602
28.2.5	Function Documentation	605
28.3	Quad Serial Peripheral Interface EDMA Driver	617
28.3.1	Overview	617
28.3.2	Data Structure Documentation	618
28.3.3	Macro Definition Documentation	618
28.3.4	Function Documentation	618

Chapter 29 SAI: Serial Audio Interface

29.1	Overview	622
29.2	Typical configurations	622
29.3	Typical use case	623
29.3.1	SAI Send/receive using an interrupt method	623
29.3.2	SAI Send/receive using a DMA method	623
29.4	SAI Driver	624
29.4.1	Overview	624

Section No.	Title	Page No.
29.4.2	Data Structure Documentation	631
29.4.3	Macro Definition Documentation	634
29.4.4	Enumeration Type Documentation	634
29.4.5	Function Documentation	638
29.5	SAI EDMA Driver	670
29.5.1	Overview	670
29.5.2	Data Structure Documentation	671
29.5.3	Function Documentation	672

Chapter 30 SEMA42: Hardware Semaphores Driver

30.1	Overview	683
30.2	Typical use case	683
30.3	Macro Definition Documentation	685
30.3.1	SEMA42_GATE_NUM_RESET_ALL	685
30.3.2	SEMA42_GATEn	685
30.4	Enumeration Type Documentation	685
30.4.1	anonymous enum	685
30.4.2	sema42_gate_status_t	685
30.5	Function Documentation	686
30.5.1	SEMA42_Init	686
30.5.2	SEMA42_Deinit	686
30.5.3	SEMA42_TryLock	686
30.5.4	SEMA42_Lock	687
30.5.5	SEMA42_Unlock	688
30.5.6	SEMA42_GetGateStatus	688
30.5.7	SEMA42_ResetGate	688
30.5.8	SEMA42_ResetAllGates	689

Chapter 31 SNVS: Secure Non-Volatile Storage

31.1	Overview	690
31.2	Secure Non-Volatile Storage High-Power	691
31.2.1	Overview	691
31.2.2	Data Structure Documentation	694
31.2.3	Macro Definition Documentation	695
31.2.4	Enumeration Type Documentation	696
31.2.5	Function Documentation	697
31.3	Secure Non-Volatile Storage Low-Power	707

Section No.	Title	Page No.
31.3.1	Overview	707
31.3.2	Data Structure Documentation	710
31.3.3	Enumeration Type Documentation	711
31.3.4	Function Documentation	711
 Chapter 32 TPM: Timer PWM Module		
32.1	Overview	719
32.2	Introduction of TPM	719
32.2.1	Initialization and deinitialization	719
32.2.2	PWM Operations	719
32.2.3	Input capture operations	720
32.2.4	Output compare operations	720
32.2.5	Quad decode	720
32.2.6	Fault operation	720
32.2.7	Status	720
32.2.8	Interrupt	720
32.3	Typical use case	721
32.3.1	PWM output	721
32.4	Data Structure Documentation	724
32.4.1	struct tpm_chnl_pwm_signal_param_t	724
32.4.2	struct tpm_config_t	725
32.5	Macro Definition Documentation	725
32.5.1	FSL TPM DRIVER VERSION	725
32.6	Enumeration Type Documentation	725
32.6.1	tpm_chnl_t	725
32.6.2	tpm_pwm_mode_t	726
32.6.3	tpm_pwm_level_select_t	726
32.6.4	tpm_chnl_control_bit_mask_t	726
32.6.5	tpm_trigger_select_t	727
32.6.6	tpm_output_compare_mode_t	727
32.6.7	tpm_input_capture_edge_t	727
32.6.8	tpm_clock_source_t	727
32.6.9	tpm_clock_prescale_t	727
32.6.10	tpm_interrupt_enable_t	728
32.6.11	tpm_status_flags_t	728
32.7	Function Documentation	728
32.7.1	TPM_Init	728
32.7.2	TPM_Deinit	729
32.7.3	TPM_GetDefaultConfig	729

Section No.	Title	Page No.
32.7.4	TPM_CalculateCounterClkDiv	729
32.7.5	TPM_SetupPwm	730
32.7.6	TPM_UpdatePwmDutycycle	730
32.7.7	TPM_UpdateChnlEdgeLevelSelect	731
32.7.8	TPM_GetChannelContorlBits	731
32.7.9	TPM_DisableChannel	732
32.7.10	TPM_EnableChannel	732
32.7.11	TPM_SetupInputCapture	732
32.7.12	TPM_SetupOutputCompare	733
32.7.13	TPM_EnableInterrupts	733
32.7.14	TPM_DisableInterrupts	733
32.7.15	TPM_GetEnabledInterrupts	734
32.7.16	TPM_GetChannelValue	734
32.7.17	TPM_GetStatusFlags	734
32.7.18	TPM_ClearStatusFlags	735
32.7.19	TPM_SetTimerPeriod	735
32.7.20	TPM_GetCurrentTimerCount	735
32.7.21	TPM_StartTimer	736
32.7.22	TPM_StopTimer	736

Chapter 33 TRGMUX: Trigger Mux Driver

33.1	Overview	737
33.2	Typical use case	737
33.3	Macro Definition Documentation	737
33.3.1	FSL_TRGMUX_DRIVER_VERSION	737
33.4	Enumeration Type Documentation	738
33.4.1	anonymous enum	738
33.4.2	trgmux_trigger_input_t	738
33.5	Function Documentation	738
33.5.1	TRGMUX_LockRegister	738
33.5.2	TRGMUX_SetTriggerSource	738

Chapter 34 TRNG: True Random Number Generator

34.1	TRNG Initialization	740
34.2	Get random data from TRNG	740

Section No.	Title	Page No.
Chapter 35 TSTMR: Timestamp Timer Driver		
35.1 Overview	741	
35.2 Function Documentation	741	
35.2.1 TSTM_R_ReadOnlyStamp	741	
35.2.2 TSTM_R_DelayUs	741	
Chapter 36 WDOG32: 32-bit Watchdog Timer		
36.1 Overview	743	
36.2 Typical use case	743	
36.3 Data Structure Documentation	745	
36.3.1 struct wdog32_work_mode_t	745	
36.3.2 struct wdog32_config_t	745	
36.4 Macro Definition Documentation	745	
36.4.1 FSL_WDOG32_DRIVER_VERSION	745	
36.5 Enumeration Type Documentation	746	
36.5.1 wdog32_clock_source_t	746	
36.5.2 wdog32_clock_prescaler_t	746	
36.5.3 wdog32_test_mode_t	746	
36.5.4 _wdog32_interrupt_enable_t	746	
36.5.5 _wdog32_status_flags_t	746	
36.6 Function Documentation	747	
36.6.1 WDOG32_GetDefaultConfig	747	
36.6.2 WDOG32_Init	747	
36.6.3 WDOG32_Deinit	748	
36.6.4 WDOG32_Unlock	748	
36.6.5 WDOG32_Enable	748	
36.6.6 WDOG32_Disable	748	
36.6.7 WDOG32_EnableInterrupts	750	
36.6.8 WDOG32_DisableInterrupts	750	
36.6.9 WDOG32_GetStatusFlags	750	
36.6.10 WDOG32_ClearStatusFlags	751	
36.6.11 WDOG32_SetTimeoutValue	751	
36.6.12 WDOG32_SetWindowValue	752	
36.6.13 WDOG32_Refesh	752	
36.6.14 WDOG32_GetCounterValue	752	

Section No.	Title	Page No.
Chapter 37 XRDC: Extended Resource Domain Controller		
37.1	Overview	753
37.2	XRDC functions	753
37.3	Typical use case	753
37.3.1	Set up configurations during system initialization	753
37.3.2	XRDC error handle	753
37.3.3	Access involve SEMA42	754
37.4	Data Structure Documentation	759
37.4.1	struct xrdc_hardware_config_t	759
37.4.2	struct xrdc_processor_domain_assignment_t	760
37.4.3	struct xrdc_non_processor_domain_assignment_t	761
37.4.4	struct xrdc_pid_config_t	762
37.4.5	struct xrdc_periph_access_config_t	763
37.4.6	struct xrdc_mem_access_config_t	763
37.4.7	struct xrdc_error_t	764
37.5	Enumeration Type Documentation	765
37.5.1	anonymous enum	765
37.5.2	xrdc_pid_enable_t	765
37.5.3	xrdc_did_sel_t	765
37.5.4	xrdc_secure_attr_t	766
37.5.5	xrdc_privilege_attr_t	766
37.5.6	xrdc_pid_lock_t	766
37.5.7	xrdc_access_config_lock_t	766
37.5.8	xrdc_mem_size_t	766
37.5.9	xrdc_controller_t	767
37.5.10	xrdc_error_state_t	768
37.5.11	xrdc_error_attr_t	768
37.5.12	xrdc_error_type_t	768
37.6	Function Documentation	769
37.6.1	XRDC_Init	769
37.6.2	XRDC_Deinit	770
37.6.3	XRDC_GetHardwareConfig	770
37.6.4	XRDC_LockGlobalControl	770
37.6.5	XRDC_SetGlobalValid	770
37.6.6	XRDC_GetCurrentMasterDomainId	771
37.6.7	XRDC_GetAndClearFirstDomainError	771
37.6.8	XRDC_GetAndClearFirstSpecificDomainError	771
37.6.9	XRDC_GetPidDefaultConfig	772
37.6.10	XRDC_SetPidConfig	772
37.6.11	XRDC_SetPidLockMode	773

Section No.	Title	Page No.
37.6.12	XRDC_GetDefaultNonProcessorDomainAssignment	773
37.6.13	XRDC_GetDefaultProcessorDomainAssignment	773
37.6.14	XRDC_SetNonProcessorDomainAssignment	774
37.6.15	XRDC_SetProcessorDomainAssignment	774
37.6.16	XRDC_LockMasterDomainAssignment	775
37.6.17	XRDC_SetMasterDomainAssignmentValid	775
37.6.18	XRDC_GetMemAccessDefaultConfig	777
37.6.19	XRDC_SetMemAccessConfig	777
37.6.20	XRDC_SetMemAccessLockMode	778
37.6.21	XRDC_SetMemAccessValid	778
37.6.22	XRDC_GetPeriphAccessDefaultConfig	779
37.6.23	XRDC_SetPeriphAccessConfig	779
37.6.24	XRDC_SetPeriphAccessLockMode	780
37.6.25	XRDC_SetPeriphAccessValid	780

Chapter 38 Debug Console

38.1	Overview	782
38.2	Function groups	782
38.2.1	Initialization	782
38.2.2	Advanced Feature	783
38.2.3	SDK_DEBUGCONSOLE and SDK_DEBUGCONSOLE_UART	787
38.3	Typical use case	788
38.4	Macro Definition Documentation	790
38.4.1	DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN	790
38.4.2	DEBUGCONSOLE_REDIRECT_TO_SDK	790
38.4.3	DEBUGCONSOLE_DISABLE	790
38.4.4	SDK_DEBUGCONSOLE	790
38.4.5	PRINTF	790
38.5	Function Documentation	790
38.5.1	DbgConsole_Init	790
38.5.2	DbgConsole_Deinit	791
38.5.3	DbgConsole_EnterLowpower	791
38.5.4	DbgConsole_ExitLowpower	792
38.5.5	DbgConsole_Printf	792
38.5.6	DbgConsole_Vprintf	792
38.5.7	DbgConsole_Putchar	792
38.5.8	DbgConsole_Scanf	793
38.5.9	DbgConsole_Getchar	793
38.5.10	DbgConsole_BlockingPrintf	794
38.5.11	DbgConsole_BlockingVprintf	794

Section No.	Title	Page No.
38.5.12	DbgConsole_Flush	794
38.5.13	DbgConsole_TryGetchar	795
38.6	debug console configuration	797
38.6.1	Overview	797
38.6.2	Macro Definition Documentation	798
38.7	Semihosting	800
38.7.1	Guide Semihosting for IAR	800
38.7.2	Guide Semihosting for Keil µVision	800
38.7.3	Guide Semihosting for MCUXpresso IDE	801
38.7.4	Guide Semihosting for ARMGCC	801
38.8	SWO	804
38.8.1	Guide SWO for SDK	804
38.8.2	Guide SWO for Keil µVision	805
38.8.3	Guide SWO for MCUXpresso IDE	806
38.8.4	Guide SWO for ARMGCC	806
Chapter 39 Notification Framework		
39.1	Overview	807
39.2	Notifier Overview	807
39.3	Data Structure Documentation	809
39.3.1	struct notifier_notification_block_t	809
39.3.2	struct notifier_callback_config_t	810
39.3.3	struct notifier_handle_t	810
39.4	Typedef Documentation	811
39.4.1	notifier_user_config_t	811
39.4.2	notifier_user_function_t	811
39.4.3	notifier_callback_t	812
39.5	Enumeration Type Documentation	812
39.5.1	_notifier_status	812
39.5.2	notifier_policy_t	813
39.5.3	notifier_notification_type_t	813
39.5.4	notifier_callback_type_t	813
39.6	Function Documentation	814
39.6.1	NOTIFIER_CreateHandle	814
39.6.2	NOTIFIER_SwitchConfig	815
39.6.3	NOTIFIER_GetErrorCallbackIndex	816

Section No.	Title	Page No.
Chapter 40 Shell		
40.1 Overview	817	
40.2 Function groups	817	
40.2.1 Initialization	817	
40.2.2 Advanced Feature	817	
40.2.3 Shell Operation	817	
40.3 Data Structure Documentation	819	
40.3.1 struct shell_command_t	819	
40.4 Macro Definition Documentation	820	
40.4.1 SHELL_NON_BLOCKING_MODE	820	
40.4.2 SHELL_AUTO_COMPLETE	820	
40.4.3 SHELL_BUFFER_SIZE	820	
40.4.4 SHELL_MAX_ARGS	820	
40.4.5 SHELL_HISTORY_COUNT	820	
40.4.6 SHELL_HANDLE_SIZE	820	
40.4.7 SHELL_USE_COMMON_TASK	821	
40.4.8 SHELL_TASK_PRIORITY	821	
40.4.9 SHELL_TASK_STACK_SIZE	821	
40.4.10 SHELL_HANDLE_DEFINE	821	
40.4.11 SHELL_COMMAND_DEFINE	821	
40.4.12 SHELL_COMMAND	822	
40.5 Typedef Documentation	822	
40.5.1 cmd_function_t	822	
40.6 Enumeration Type Documentation	822	
40.6.1 shell_status_t	822	
40.7 Function Documentation	822	
40.7.1 SHELL_Init	822	
40.7.2 SHELL_RegisterCommand	823	
40.7.3 SHELL_UnregisterCommand	824	
40.7.4 SHELL_Write	824	
40.7.5 SHELL_Printf	825	
40.7.6 SHELL_WriteSynchronization	825	
40.7.7 SHELL_PrintfSynchronization	825	
40.7.8 SHELL_ChangePrompt	826	
40.7.9 SHELL_PrintPrompt	826	
40.7.10 SHELL_Task	826	
40.7.11 SHELL_checkRunningInIsr	827	

Section No.	Title	Page No.
Chapter 41 CODEC Driver		
41.1 Overview		828
41.2 CODEC Common Driver		829
41.2.1 Overview		829
41.2.2 Data Structure Documentation		834
41.2.3 Macro Definition Documentation		835
41.2.4 Enumeration Type Documentation		835
41.2.5 Function Documentation		840
41.3 CODEC I2C Driver		844
41.3.1 Overview		844
41.3.2 Data Structure Documentation		845
41.3.3 Enumeration Type Documentation		845
41.3.4 Function Documentation		845
41.4 DA7212 Driver		848
41.4.1 Overview		848
41.4.2 Data Structure Documentation		851
41.4.3 Macro Definition Documentation		852
41.4.4 Enumeration Type Documentation		852
41.4.5 Function Documentation		854
41.4.6 DA7212 Adapter		859
41.5 SGTL5000 Driver		867
41.5.1 Overview		867
41.5.2 Data Structure Documentation		869
41.5.3 Macro Definition Documentation		870
41.5.4 Enumeration Type Documentation		870
41.5.5 Function Documentation		872
41.5.6 SGTL5000 Adapter		878
41.6 WM8960 Driver		886
41.6.1 Overview		886
41.6.2 Data Structure Documentation		889
41.6.3 Macro Definition Documentation		891
41.6.4 Enumeration Type Documentation		891
41.6.5 Function Documentation		893
41.6.6 WM8960 Adapter		900
Chapter 42 Serial Manager		
42.1 Overview		908
42.2 Data Structure Documentation		911

Section No.	Title	Page No.
42.2.1	struct serial_manager_config_t	911
42.2.2	struct serial_manager_callback_message_t	912
42.3	Macro Definition Documentation	912
42.3.1	SERIAL_MANAGER_WRITE_TIME_DELAY_DEFAULT_VALUE	912
42.3.2	SERIAL_MANAGER_READ_TIME_DELAY_DEFAULT_VALUE	912
42.3.3	SERIAL_MANAGER_USE_COMMON_TASK	912
42.3.4	SERIAL_MANAGER_HANDLE_SIZE	912
42.3.5	SERIAL_MANAGER_HANDLE_DEFINE	912
42.3.6	SERIAL_MANAGER_WRITE_HANDLE_DEFINE	913
42.3.7	SERIAL_MANAGER_READ_HANDLE_DEFINE	913
42.3.8	SERIAL_MANAGER_TASK_PRIORITY	914
42.3.9	SERIAL_MANAGER_TASK_STACK_SIZE	914
42.4	Enumeration Type Documentation	914
42.4.1	serial_port_type_t	914
42.4.2	serial_manager_type_t	914
42.4.3	serial_manager_status_t	914
42.5	Function Documentation	915
42.5.1	SerialManager_Init	915
42.5.2	SerialManager_Deinit	916
42.5.3	SerialManager_OpenWriteHandle	916
42.5.4	SerialManager_CloseWriteHandle	918
42.5.5	SerialManager_OpenReadHandle	919
42.5.6	SerialManager_CloseReadHandle	920
42.5.7	SerialManager_WriteBlocking	920
42.5.8	SerialManager_ReadBlocking	921
42.5.9	SerialManager_WriteNonBlocking	922
42.5.10	SerialManager_ReadNonBlocking	922
42.5.11	SerialManager_TryRead	923
42.5.12	SerialManager_CancelWriting	924
42.5.13	SerialManager_CancelReading	924
42.5.14	SerialManager_InstallTxCallback	925
42.5.15	SerialManager_InstallRxCallback	925
42.5.16	SerialManager_needPollingIsr	927
42.5.17	SerialManager_EnterLowpower	927
42.5.18	SerialManager_ExitLowpower	927
42.5.19	SerialManager_SetLowpowerCriticalCb	928
42.6	Serial Port Uart	929
42.6.1	Overview	929
42.6.2	Enumeration Type Documentation	929
42.7	Serial Port SWO	930

Section No.	Title	Page No.
42.7.1	Overview	930
42.7.2	Data Structure Documentation	930
42.7.3	Enumeration Type Documentation	930
42.7.4	CODEC Adapter	931

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

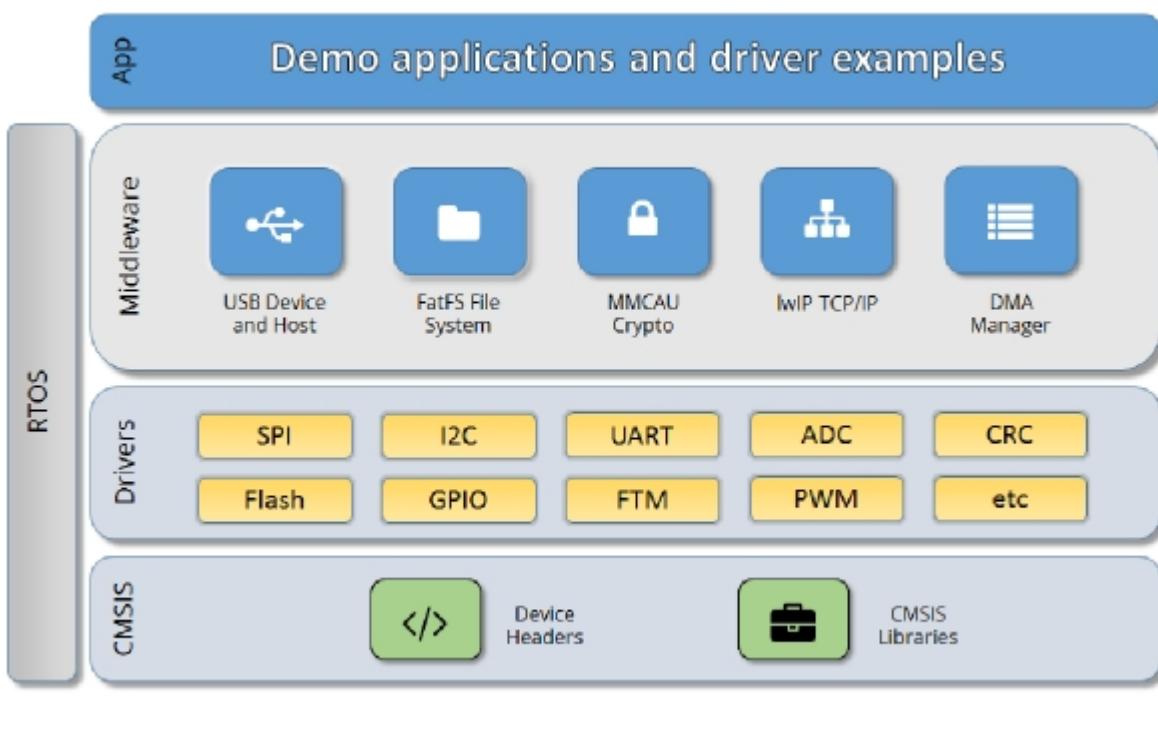
Architectural Overview

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_sppll_config_t`
SCG system PLL configuration. [More...](#)
- struct `scg_rosc_config_t`
SCG RTC OSC configuration. [More...](#)
- struct `scg_apll_config_t`
SCG auxiliary PLL configuration. [More...](#)

Macros

- `#define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0`
Configure whether driver controls clock.
- `#define SCG SCG0`
Re-map the SCG peripheral base address for i.MX 7ULP.

- #define **SCG_PLLPFD_PFD_VAL**(pfдClkout, fracValue) ((uint32_t)((uint32_t)(fracValue) << (uint32_t)(pfдClkout)))
SCG (A/S)PLLPFD[PFDx] value.
- #define **SCG_PLLPFD_PFD_MASK**(pfдClkout) ((uint32_t)((uint32_t)(SCG_APPLLFD_PFD0_-MASK) << (uint32_t)(pfдClkout)))
SCG (A/S)PLLPFD[PFD] mask.
- #define **SCG_PLLPFD_PFD_VALID_MASK**(pfдClkout) ((uint32_t)((uint32_t)SCG_APPLLFD_-PFD0_VALID_MASK << (uint32_t)(pfдClkout)))
SCG (A/S)PLLPFD[PFDx_VALID] mask.
- #define **SCG_PLLPFD_PFD_CLKGATE_MASK**(pfдClkout) ((uint32_t)((uint32_t)SCG_APPLLFD_PFD0_CLKGATE_MASK << (uint32_t)(pfдClkout)))
SCG (A/S)PLLPFD[PFDx_CLKGATE] mask.
- #define **PCC_CLKCFG_PCD_MASK** (0x7U)
Re-define PCC register masks and bitfield operations to unify the different namings in the soc header file.
- #define **DMAMUX_CLOCKS**
Clock ip name array for DMAMUX.
- #define **RGPIO2P_CLOCKS**
Clock ip name array for RGPIO2P.
- #define **SAI_CLOCKS**
Clock ip name array for SAI.
- #define **PCTL_CLOCKS**
Clock ip name array for PCTL.
- #define **LPI2C_CLOCKS**
Clock ip name array for LPI2C.
- #define **FLEXIO_CLOCKS**
Clock ip name array for FLEXIO.
- #define **EDMA_CLOCKS**
Clock ip name array for EDMA.
- #define **LPUART_CLOCKS**
Clock ip name array for LPUART.
- #define **DAC_CLOCKS**
Clock ip name array for DAC.
- #define **SNVS_HP_CLOCKS**
Clock ip name array for DAC.
- #define **LPTMR_CLOCKS**
Clock ip name array for LPTMR.
- #define **LPADC_CLOCKS**
Clock ip name array for LPADC.
- #define **TRNG_CLOCKS**
Clock ip name array for TRNG.
- #define **LPSPI_CLOCKS**
Clock ip name array for LPSPI.
- #define **TPM_CLOCKS**
Clock ip name array for TPM.
- #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 **XRDC_CLOCKS**

- `#define MU_CLOCKS`
Clock ip name array for MU.
- `#define WDOG_CLOCKS`
Clock ip name array for WDOG.
- `#define LTC_CLOCKS`
Clock ip name array for LTC.
- `#define DPM_CLOCKS`
Clock ip name array for DPM.
- `#define SEMA42_CLOCKS`
Clock ip name array for SEMA42.
- `#define TPIU_CLOCKS`
Clock ip name array for TPIU.
- `#define QSPI_CLOCKS`
Clock ip name array for QSPI.
- `#define LPO_CLK_FREQ 1000U`
LPO clock frequency.
- `#define TPIU_CLK_FREQ 100000000U`
TPIU clock frequency.
- `#define kCLOCK_Osc0ErClk kCLOCK_ErClk`
For compatible with other MCG platforms.
- `#define kCLOCK_Er32kClk kCLOCK_Osc32kClk`
For compatible with other MCG platforms.
- `#define CLOCK_GetOsc0ErClkFreq CLOCK_GetErClkFreq`
For compatible with other MCG platforms.
- `#define CLOCK_GetEr32kClkFreq CLOCK_GetOsc32kClkFreq`
For compatible with other MCG platforms.

Enumerations

- enum `clock_name_t` {

kCLOCK_CoreSysClk,
 kCLOCK_PlatClk,
 kCLOCK_ExtClk,
 kCLOCK_BusClk,
 kCLOCK_SlowClk,
 kCLOCK_ScgSysOscClk,
 kCLOCK_ScgSircClk,
 kCLOCK_ScgFircClk,
 kCLOCK_ScgRtcOscClk,
 kCLOCK_ScgAuxPllClk,
 kCLOCK_ScgSysPllClk,
 kCLOCK_ScgSysOscAsyncDiv1Clk,
 kCLOCK_ScgSysOscAsyncDiv2Clk,
 kCLOCK_ScgSysOscAsyncDiv3Clk,
 kCLOCK_ScgSircAsyncDiv1Clk,
 kCLOCK_ScgSircAsyncDiv2Clk,
 kCLOCK_ScgSircAsyncDiv3Clk,
 kCLOCK_ScgFircAsyncDiv1Clk,
 kCLOCK_ScgFircAsyncDiv2Clk,
 kCLOCK_ScgFircAsyncDiv3Clk,
 kCLOCK_ScgSysPllPfd0Clk,
 kCLOCK_ScgSysPllPfd1Clk,
 kCLOCK_ScgSysPllPfd2Clk,
 kCLOCK_ScgSysPllPfd3Clk,
 kCLOCK_ScgAuxPllPfd0Clk,
 kCLOCK_ScgAuxPllPfd1Clk,
 kCLOCK_ScgAuxPllPfd2Clk,
 kCLOCK_ScgAuxPllPfd3Clk,
 kCLOCK_ScgSysPllAsyncDiv1Clk,
 kCLOCK_ScgSysPllAsyncDiv2Clk,
 kCLOCK_ScgSysPllAsyncDiv3Clk,
 kCLOCK_ScgAuxPllAsyncDiv1Clk,
 kCLOCK_ScgAuxPllAsyncDiv2Clk,
 kCLOCK_ScgAuxPllAsyncDiv3Clk,
 kCLOCK_LpoClk,
 kCLOCK_Osc32kClk,
 kCLOCK_ErClk,
 kCLOCK_LvdsClk
 }

Clock name used to get clock frequency.

- enum `clock_ip_src_t` {

```
kCLOCK_IpSrcNone = 0U,
kCLOCK_IpSrcSysOscAsync = 1U,
kCLOCK_IpSrcSircAsync = 2U,
kCLOCK_IpSrcFircAsync = 3U,
kCLOCK_IpSrcRtcAuxPllAsync = 4U,
kCLOCK_IpSrcSystem = 5U,
kCLOCK_IpSrcSysPllAsync = 6U,
kCLOCK_IpSrcPllPfdAsync = 7U }
```

Clock source for peripherals that support various clock selections.

- enum `clock_lptmr_src_t` {


```
kCLOCK_LptmrSrcSircAsync = 0U,
kCLOCK_LptmrSrcLPO1K = 1U,
kCLOCK_LptmrSrcXTAL32K = 2U,
kCLOCK_LptmrSrcExternal = 3U }
```

Clock source for LPTMR.

- 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_SysClkBus,
kSCG_SysClkExt,
kSCG_SysClkPlat,
kSCG_SysClkCore }
```

SCG system clock type.

- enum `scg_sys_clk_src_t` {


```
kSCG_SysClkSrcSysOsc = 1U,
kSCG_SysClkSrcSirc = 2U,
kSCG_SysClkSrcFirc = 3U,
kSCG_SysClkSrcRosc = 4U,
kSCG_SysClkSrcAuxPll = 5U,
kSCG_SysClkSrcSysPll = 6U }
```

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` {


```
kClockClkoutSelScgExt = 0U,
kClockClkoutSelSysOsc = 1U,
kClockClkoutSelSirc = 2U,
kClockClkoutSelFirc = 3U,
kClockClkoutSelScgRtcOsc = 4U,
kClockClkoutSelScgAuxPll = 5U,
kClockClkoutSelSysPll = 6U }
```

SCG clock out configuration (CLKOUTSEL).

- enum `scg_async_clk_t` {


```
kSCG_AsyncDiv1Clk,
kSCG_AsyncDiv2Clk,
kSCG_AsyncDiv3Clk }
```

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 `_scg_sosc_enable_mode` {

 `kSCG_SysOscEnable` = SCG_SOSCCSR_SOSCEN_MASK,

 `kSCG_SysOscEnableInStop` = SCG_SOSCCSR_SOSCSTEN_MASK,

 `kSCG_SysOscEnableInLowPower` = SCG_SOSCCSR_SOSCLPEN_MASK }

 OSC enable mode.
- enum `scg_sirc_range_t` {

 `kSCG_SircRangeLow`,

 `kSCG_SircRangeHigh` }

 SCG slow IRC clock frequency range.
- enum `_scg_sirc_enable_mode` {

 `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_FircTrimSrcUsb0` = 0U,

 `kSCG_FircTrimSrcUsb1` = 1U,

 `kSCG_FircTrimSrcSysOsc` = 2U,

 `kSCG_FircTrimSrcRtcOsc` = 3U }

 SCG fast IRC trim source.
- enum `scg_firc_range_t` {

 `kSCG_FircRange48M`,

 `kSCG_FircRange52M`,

 `kSCG_FircRange56M`,

 `kSCG_FircRange60M` }

 SCG fast IRC clock frequency range.
- enum `_scg_firc_enable_mode` {

 `kSCG_FircEnable` = SCG_FIRCCSR_FIRCEN_MASK,

- ```
kSCG_FircEnableInStop = SCG_FIRCCSR_FIRCSTEN_MASK }
```

*FIRC enable mode.*
- enum `scg_sppll_src_t` {  
`kSCG_SysPllSrcSysOsc,`  
`kSCG_SysPllSrcFirc }`
- SCG system PLL clock source.*
- enum `_scg_sppll_enable_mode` {  
`kSCG_SysPllEnable = SCG_SPLLCSR_SPLLEN_MASK,`  
`kSCG_SysPllEnableInStop = SCG_SPLLCSR_SPLLSTEN_MASK }`
- SPLL enable mode.*
- enum `scg_sppll_pfd_clkout_t` {  
`kSCG_SysPllpfd0Clk = 0U,`  
`kSCG_SysPllpfd1Clk = 8U,`  
`kSCG_SysPllpfd2Clk = 16U,`  
`kSCG_SysPllpfd3Clk = 24U }`
- SCG system PLL PFD clouk out select.*
- enum `scg_rosec_monitor_mode_t` {  
`kSCG_rtcOscMonitorDisable = 0U,`  
`kSCG_rtcOscMonitorInt = SCG_ROSCCSR_ROSCCM_MASK,`  
`kSCG_rtcOscMonitorReset }`
- SCG RTC OSC monitor mode.*
- enum `scg_apll_src_t` {  
`kSCG_AuxPllSrcSysOsc,`  
`kSCG_AuxPllSrcFirc }`
- SCG auxiliary PLL clock source.*
- enum `_scg_apll_enable_mode` {  
`kSCG_AuxPllEnable = SCG_APLLCSR_APllen_MASK,`  
`kSCG_AuxPllEnableInStop = SCG_APLLCSR_APllsten_MASK }`
- APLL enable mode.*
- enum `scg_apll_pfd_clkout_t` {  
`kSCG_AuxPllpfd0Clk = 0U,`  
`kSCG_AuxPllpfd1Clk = 8U,`  
`kSCG_AuxPllpfd2Clk = 16U,`  
`kSCG_AuxPllpfd3Clk = 24U }`
- SCG auxiliary PLL PFD clouk out select.*

## 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 bool `CLOCK_IsEnabledByOtherCore (clock_ip_name_t name)`  
*Check whether the clock is already enabled and configured by any other core.*
- 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, uint8_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_GetPlatClkFreq (void)`  
Get the platform clock frequency.
- `uint32_t CLOCK_GetExtClkFreq (void)`  
Get the external clock frequency.
- `uint32_t CLOCK_GetBusClkFreq (void)`  
Get the bus clock frequency.
- `uint32_t CLOCK_GetSlowClkFreq (void)`  
Get the slow clock frequency.
- `uint32_t CLOCK_GetOsc32kClkFreq (void)`  
Get the OSC 32K clock frequency (OSC32KCLK).
- `uint32_t CLOCK_GetErClkFreq (void)`  
Get the external reference clock frequency (ERCLK).
- `uint32_t CLOCK_GetLvdsClkFreq (void)`  
Get the external LVDS pad clock frequency (LVDS).
- `uint32_t CLOCK_GetIpFreq (clock_ip_name_t name)`  
Gets the clock frequency for a specific IP module.
- `uint32_t CLOCK_GetRtcOscFreq (void)`  
Gets the SCG RTC OSC clock frequency.
- `static bool CLOCK_IsRtcOscErr (void)`  
Checks whether the RTC OSC clock error occurs.
- `static void CLOCK_ClearRtcOscErr (void)`  
Clears the RTC OSC clock error.
- `static void CLOCK_SetRtcOscMonitorMode (scg_rose_monitor_mode_t mode)`  
Sets the RTC OSC monitor mode.
- `static bool CLOCK_IsRtcOscValid (void)`  
Checks whether the RTC OSC clock is valid.

## Variables

- `volatile uint32_t g_xtal0Freq`  
External XTAL0 (OSCO/SYROS) clock frequency.
- `volatile uint32_t g_xtal32Freq`  
External XTAL32/EXTAL32 clock frequency.
- `volatile uint32_t g_lvdsFreq`  
External LVDS pad clock frequency.

## Driver version

- `#define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))`  
*CLOCK driver version 2.3.1.*

## 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)`

- static void **CLOCK\_SetRunModeSysClkConfig** (const `scg_sys_clk_config_t` \*config)
 

*Sets the system clock configuration for VLPR mode.*
- static void **CLOCK\_SetHsrnModeSysClkConfig** (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.*
- static void **CLOCK\_EnableLpoPowerOption** (bool enable)
 

*Enables/disables the SCG slow IRC 1khz LPO clock in LLS/VLLSx modes.*
- 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_IsFircErr` (`void`)
 

*Checks whether the FIRC clock error occurs.*
- `static void CLOCK_ClearFircErr` (`void`)
 

*Clears the FIRC clock error.*
- `static bool CLOCK_IsFircValid` (`void`)
 

*Checks whether the FIRC clock is valid.*

## 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 divBus: 4`

*Bus clock divider, see `scg_sys_clk_div_t`.*
- `uint32_t __pad0__: 4`

*Reserved.*
- `uint32_t divPlat: 4`

*Platform clock divider, which can only be divided by 1.*
- `uint32_t divCore: 4`

*Core clock divider, see `scg_sys_clk_div_t`.*
- `uint32_t __pad1__: 4`

*Reserved.*
- `uint32_t src: 4`

*System clock source, see `scg_sys_clk_src_t`.*
- `uint32_t __pad2__: 4`

*reserved.*

**Field Documentation**

- (1) `uint32_t scg_sys_clk_config_t::divSlow`
- (2) `uint32_t scg_sys_clk_config_t::divBus`
- (3) `uint32_t scg_sys_clk_config_t::__pad0__`
- (4) `uint32_t scg_sys_clk_config_t::divPlat`

See [kSCG\\_SysClkDivBy1](#).

- (5) `uint32_t scg_sys_clk_config_t::divCore`
- (6) `uint32_t scg_sys_clk_config_t::__pad1__`
- (7) `uint32_t scg_sys_clk_config_t::src`
- (8) `uint32_t scg_sys_clk_config_t::__pad2__`

#### 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 div1`  
*SOSCDIV1 value.*
- `scg_async_clk_div_t div2`  
*SOSCDIV2 value.*
- `scg_async_clk_div_t div3`  
*SOSCDIV3 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::div1`
- (5) `scg_async_clk_div_t scg_sosc_config_t::div2`
- (6) `scg_async_clk_div_t scg_sosc_config_t::div3`
- (7) `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 div1`  
*SIRCDIV1 value.*
- `scg_async_clk_div_t div2`  
*SIRCDIV2 value.*
- `scg_async_clk_div_t div3`  
*SIRCDIV3 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::div1`
- (3) `scg_async_clk_div_t scg_sirc_config_t::div2`
- (4) `scg_async_clk_div_t scg_sirc_config_t::div3`
- (5) `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 div1`  
*FIRCDIV1 value.*
- `scg_async_clk_div_t div2`  
*FIRCDIV2 value.*
- `scg_async_clk_div_t div3`  
*FIRCDIV3 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::div1`
- (3) `scg_async_clk_div_t scg_firc_config_t::div2`
- (4) `scg_async_clk_div_t scg_firc_config_t::div3`
- (5) `scg_firc_range_t scg_firc_config_t::range`
- (6) `const scg_firc_trim_config_t* scg_firc_config_t::trimConfig`

**4.2.6 struct scg\_spill\_config\_t****Data Fields**

- `uint8_t enableMode`  
*Enable mode, OR'ed value of \_scg\_spill\_enable\_mode.*
- `scg_async_clk_div_t div1`  
*SPLL DIV1 value.*
- `scg_async_clk_div_t div2`  
*SPLL DIV2 value.*
- `scg_async_clk_div_t div3`  
*SPLL DIV3 value.*
- `scg_spill_src_t src`  
*Clock source.*
- `bool isPfdSelected`  
*SPLL PFD output clock selected.*
- `uint8_t prediv`  
*PLL reference clock divider.*
- `scg_spill_pfd_clkout_t pfdClkout`  
*PLL PFD clouk out select.*
- `uint8_t mult`  
*System PLL multiplier.*

**Field Documentation**

- (1) scg\_async\_clk\_div\_t scg\_spill\_config\_t::div1
- (2) scg\_async\_clk\_div\_t scg\_spill\_config\_t::div2
- (3) scg\_async\_clk\_div\_t scg\_spill\_config\_t::div3
- (4) scg\_spill\_src\_t scg\_spill\_config\_t::src
- (5) bool scg\_spill\_config\_t::isPfdSelected
- (6) uint8\_t scg\_spill\_config\_t::prediv
- (7) scg\_spill\_pfd\_clkout\_t scg\_spill\_config\_t::pfdClkout
- (8) uint8\_t scg\_spill\_config\_t::mult

**4.2.7 struct scg\_rosc\_config\_t****Data Fields**

- scg\_rosc\_monitor\_mode\_t monitorMode  
*Clock monitor mode selected.*

**Field Documentation**

- (1) scg\_rosc\_monitor\_mode\_t scg\_rosc\_config\_t::monitorMode

**4.2.8 struct scg\_apll\_config\_t****Data Fields**

- uint8\_t enableMode  
*Enable mode, OR'ed value of \_scg\_apll\_enable\_mode.*
- scg\_async\_clk\_div\_t div1  
*APLLDIV1 value.*
- scg\_async\_clk\_div\_t div2  
*APLLDIV2 value.*
- scg\_async\_clk\_div\_t div3  
*APLLDIV3 value.*
- scg\_apll\_src\_t src  
*Clock source.*
- bool isPfdSelected  
*APLL PFD output clock selected.*
- uint8\_t prediv  
*PLL reference clock divider.*
- scg\_apll\_pfd\_clkout\_t pfdClkout  
*SCG auxiliary PLL PFD clouk out select.*

- `uint8_t mult`  
*Auxiliary PLL multiplier.*
- `scg_sys_clk_div_t pllPostdiv1`  
*Auxiliary PLL Post Clock Divide1 Ratio.*
- `scg_sys_clk_div_t pllPostdiv2`  
*Auxiliary PLL Post Clock Divide2 Ratio.*
- `uint32_t num`: 30  
*30-bit numerator of the Auxiliary PLL Fractional-Loop divider.*
- `uint32_t denom`: 30  
*30-bit denominator of the Auxiliary PLL Fractional-Loop divider.*

## Field Documentation

- (1) `scg_async_clk_div_t scg_apll_config_t::div1`
- (2) `scg_async_clk_div_t scg_apll_config_t::div2`
- (3) `scg_async_clk_div_t scg_apll_config_t::div3`
- (4) `scg_apll_src_t scg_apll_config_t::src`
- (5) `bool scg_apll_config_t::isPfdSelected`
- (6) `uint8_t scg_apll_config_t::prediv`
- (7) `scg_apll_pfd_clkout_t scg_apll_config_t::pfdClkout`
- (8) `uint8_t scg_apll_config_t::mult`
- (9) `scg_sys_clk_div_t scg_apll_config_t::pllPostdiv1`
- (10) `scg_sys_clk_div_t scg_apll_config_t::pllPostdiv2`
- (11) `uint32_t scg_apll_config_t::num`
- (12) `uint32_t scg_apll_config_t::denom`

## 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, 3, 1))****4.3.3 #define SCG SCG0**

This driver is for SCG 0 on Core 0 of i.MX 7ULP only.

**4.3.4 #define DMAMUX\_CLOCKS**

**Value:**

```
{
 _kCLOCK_Dmamux0, _kCLOCK_Dmamux1 \
}
```

**4.3.5 #define GPIO2P\_CLOCKS**

**Value:**

```
{
 _kCLOCK_Rgpio2p0, _kCLOCK_Rgpio2p0, _kCLOCK_Rgpio2p1, _kCLOCK_Rgpio2p1, _kCLOCK_Rgpio2p1,
 _kCLOCK_Rgpio2p1 \
}
```

**4.3.6 #define SAI\_CLOCKS**

**Value:**

```
{
 _kCLOCK_Sai0, _kCLOCK_Sai1 \
}
```

**4.3.7 #define PCTL\_CLOCKS**

**Value:**

```
{
 _kCLOCK_PctlA, _kCLOCK_PctlB, _kCLOCK_PctlC, _kCLOCK_PctlD, _kCLOCK_PctlE, _kCLOCK_PctlF \
}
```

### 4.3.8 #define LPI2C\_CLOCKS

**Value:**

```
{
 \
 kCLOCK_Lpi2c0, kCLOCK_Lpi2c1, kCLOCK_Lpi2c2, kCLOCK_Lpi2c3, kCLOCK_Lpi2c4, kCLOCK_Lpi2c5,
 kCLOCK_Lpi2c6, \
 kCLOCK_Lpi2c7
 \
}
```

### 4.3.9 #define FLEXIO\_CLOCKS

**Value:**

```
{
 \
 kCLOCK_Flexio0, kCLOCK_Flexio1 \
}
```

### 4.3.10 #define EDMA\_CLOCKS

**Value:**

```
{
 \
 kCLOCK_Dma0, kCLOCK_Dma1 \
}
```

### 4.3.11 #define LPUART\_CLOCKS

**Value:**

```
{
 \
 kCLOCK_Lpuart0, kCLOCK_Lpuart1, kCLOCK_Lpuart2, kCLOCK_Lpuart3, kCLOCK_Lpuart4, kCLOCK_Lpuart5, \
 kCLOCK_Lpuart6, kCLOCK_Lpuart7
}
```

### 4.3.12 #define DAC\_CLOCKS

**Value:**

```
{
 \
 kCLOCK_Dac0, kCLOCK_Dac1 \
}
```

### 4.3.13 #define SNVS\_HP\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Snvs \
}
```

### 4.3.14 #define LPTMR\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Lptmr0, kCLOCK_Lptmr1 \
}
```

### 4.3.15 #define LPADC\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Adco, kCLOCK_Adcl \
}
```

### 4.3.16 #define TRNG\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Trng0 \
}
```

### 4.3.17 #define LPSPI\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Lpspi0, kCLOCK_Lpspi1, kCLOCK_Lpspi2, kCLOCK_Lpspi3 \
}
```

### 4.3.18 #define TPM\_CLOCKS

**Value:**

```
{\n \\\n kCLOCK_Tpm0, kCLOCK_Tpm1, kCLOCK_Tpm2, kCLOCK_Tpm3, kCLOCK_Tpm4, kCLOCK_Tpm5, kCLOCK_Tpm6,\n kCLOCK_Tpm7 \\\n}
```

### 4.3.19 #define LPIT\_CLOCKS

**Value:**

```
{\n \\\n kCLOCK_Lpit0, kCLOCK_Lpit1 \\\n}
```

### 4.3.20 #define CRC\_CLOCKS

**Value:**

```
{\n \\\n kCLOCK_Crc0 \\\n}
```

### 4.3.21 #define CMP\_CLOCKS

**Value:**

```
{\n \\\n kCLOCK_Cmp0, kCLOCK_Cmp1 \\\n}
```

### 4.3.22 #define XRDC\_CLOCKS

**Value:**

```
{\n \\\n kCLOCK_Xrdc0 \\\n}
```

### 4.3.23 #define MU\_CLOCKS

**Value:**

```
{ \
 kCLOCK_MuA \
}
```

### 4.3.24 #define WDOG\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Wdog0, kCLOCK_Wdog1, kCLOCK_Wdog2 \
}
```

### 4.3.25 #define LTC\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Ltc0 \
}
```

### 4.3.26 #define DPM\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Dpm \
}
```

### 4.3.27 #define SEMA42\_CLOCKS

**Value:**

```
{ \
 kCLOCK_Sema420, kCLOCK_Sema421 \
}
```

### 4.3.28 #define TPIU\_CLOCKS

**Value:**

```
{
 _kCLOCK_Tpiu \
}
```

### 4.3.29 #define QSPI\_CLOCKS

**Value:**

```
{
 _kCLOCK_Qspi \
}
```

### 4.3.30 #define kCLOCK\_Osc0ErClk kCLOCK\_ErClk

### 4.3.31 #define kCLOCK\_Er32kClk kCLOCK\_Osc32kClk

### 4.3.32 #define CLOCK\_GetOsc0ErClkFreq CLOCK\_GetErClkFreq

### 4.3.33 #define CLOCK\_GetEr32kClkFreq CLOCK\_GetOsc32kClkFreq

## 4.4 Enumeration Type Documentation

### 4.4.1 enum clock\_name\_t

Enumerator

- kCLOCK\_CoreSysClk* Core/system clock.
- kCLOCK\_PlatClk* Platform clock.
- kCLOCK\_ExtClk* External clock.
- kCLOCK\_BusClk* Bus clock.
- kCLOCK\_SlowClk* Slow clock.
- kCLOCK\_ScgSysOscClk* SCG system OSC clock. (SYSOSC)
- kCLOCK\_ScgSircClk* SCG SIRC clock.
- kCLOCK\_ScgFircClk* SCG FIRC clock.
- kCLOCK\_ScgRtcOscClk* SCG RTC OSC clock. (ROSC)
- kCLOCK\_ScgAuxPllClk* SCG auxiliary PLL clock. (AUXPLL)
- kCLOCK\_ScgSysPllClk* SCG system PLL clock. (SYSPLL)
- kCLOCK\_ScgSysOscAsyncDiv1Clk* SOSCDIV1\_CLK.

*kCLOCK\_ScgSysOscAsyncDiv2Clk* SOSCDIV2\_CLK.  
*kCLOCK\_ScgSysOscAsyncDiv3Clk* SOSCDIV3\_CLK.  
*kCLOCK\_ScgSircAsyncDiv1Clk* SIRCDIV1\_CLK.  
*kCLOCK\_ScgSircAsyncDiv2Clk* SIRCDIV2\_CLK.  
*kCLOCK\_ScgSircAsyncDiv3Clk* SIRCDIV3\_CLK.  
*kCLOCK\_ScgFircAsyncDiv1Clk* FIRCDIV1\_CLK.  
*kCLOCK\_ScgFircAsyncDiv2Clk* FIRCDIV2\_CLK.  
*kCLOCK\_ScgFircAsyncDiv3Clk* FIRCDIV3\_CLK.  
*kCLOCK\_ScgSysPllPfd0Clk* spll pfd0.  
*kCLOCK\_ScgSysPllPfd1Clk* spll pfd1.  
*kCLOCK\_ScgSysPllPfd2Clk* spll pfd2.  
*kCLOCK\_ScgSysPllPfd3Clk* spll pfd3.  
*kCLOCK\_ScgAuxPllPfd0Clk* apll pfd0.  
*kCLOCK\_ScgAuxPllPfd1Clk* apll pfd1.  
*kCLOCK\_ScgAuxPllPfd2Clk* apll pfd2.  
*kCLOCK\_ScgAuxPllPfd3Clk* apll pfd3.  
*kCLOCK\_ScgSysPllAsyncDiv1Clk* SPLLDIV1\_CLK.  
*kCLOCK\_ScgSysPllAsyncDiv2Clk* SPLLDIV2\_CLK.  
*kCLOCK\_ScgSysPllAsyncDiv3Clk* SPLLDIV3\_CLK.  
*kCLOCK\_ScgAuxPllAsyncDiv1Clk* APLL DIV1\_CLK.  
*kCLOCK\_ScgAuxPllAsyncDiv2Clk* APLL DIV2\_CLK.  
*kCLOCK\_ScgAuxPllAsyncDiv3Clk* APLL DIV3\_CLK.  
*kCLOCK\_LpoClk* LPO clock.  
*kCLOCK\_Osc32kClk* External OSC 32K clock (OSC32KCLK)  
*kCLOCK\_ErClk* ERCLK. The external reference clock from SCG.  
*kCLOCK\_LvdsClk* LVDS pad input clock frequency.

#### 4.4.2 enum clock\_ip\_src\_t

Enumerator

*kCLOCK\_IpSrcNone* Clock is off.  
*kCLOCK\_IpSrcSysOscAsync* SYSOSC platform or bus clock, depending on clock IP.  
*kCLOCK\_IpSrcSircAsync* SIRC platform or bus clock, depending on clock IP.  
*kCLOCK\_IpSrcFircAsync* FIRC platform or bus clock, depending on clock IP.  
*kCLOCK\_IpSrcRtcAuxPllAsync* RTC OSC clock or AUXPLL main clock, depending on clock IP.  
*kCLOCK\_IpSrcSystem* System platform or bus clock, depending on clock IP.  
*kCLOCK\_IpSrcSysPllAsync* SYSPLL platform or bus clock, depending on clock IP.  
*kCLOCK\_IpSrcPllPfdAsync* SYSPLL PFD3 or AUXPLL PFD0 clock, depending on clock IP.

#### 4.4.3 enum clock\_lptmr\_src\_t

Enumerator

- kCLOCK\_LptmrSrcSircAsync* SIRC clock.
- kCLOCK\_LptmrSrcLPO1K* LPO 1KHz clock.
- kCLOCK\_LptmrSrcXTAL32K* RTC XTAL clock.
- kCLOCK\_LptmrSrcExternal* External clock.

#### 4.4.4 enum clock\_ip\_name\_t

It is defined as the corresponding register address.

#### 4.4.5 anonymous enum

Enumerator

- kStatus\_SCG\_Busy* Clock is busy.
- kStatus\_SCG\_InvalidSrc* Invalid source.

#### 4.4.6 enum scg\_sys\_clk\_t

Enumerator

- kSCG\_SysClkSlow* System slow clock.
- kSCG\_SysClkBus* Bus clock.
- kSCG\_SysClkExt* External clock.
- kSCG\_SysClkPlat* Platform clock.
- kSCG\_SysClkCore* Core clock.

#### 4.4.7 enum scg\_sys\_clk\_src\_t

Enumerator

- kSCG\_SysClkSrcSysOsc* System OSC.
- kSCG\_SysClkSrcSirc* Slow IRC.
- kSCG\_SysClkSrcFirc* Fast IRC.
- kSCG\_SysClkSrcRosc* RTC OSC.
- kSCG\_SysClkSrcAuxPll* Auxiliary PLL.
- kSCG\_SysClkSrcSysPll* System PLL.

#### 4.4.8 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.

#### 4.4.9 enum clock\_clkout\_src\_t

Enumerator

- kClockClkoutSelScgExt* SCG external clock.
- kClockClkoutSelSysOsc* System OSC.
- kClockClkoutSelSirc* Slow IRC.
- kClockClkoutSelFirc* Fast IRC.
- kClockClkoutSelScgRtcOsc* SCG RTC OSC clock.
- kClockClkoutSelScgAuxPll* SCG Auxiliary PLL clock.
- kClockClkoutSelSysPll* System PLL.

#### 4.4.10 enum scg\_async\_clk\_t

Enumerator

- kSCG\_AsyncDiv1Clk* The async clock by DIV1, e.g. SOSCDIV1\_CLK, SIRCDIV1\_CLK.
- kSCG\_AsyncDiv2Clk* The async clock by DIV2, e.g. SOSCDIV2\_CLK, SIRCDIV2\_CLK.
- kSCG\_AsyncDiv3Clk* The async clock by DIV3, e.g. SOSCDIV3\_CLK, SIRCDIV3\_CLK.

#### 4.4.11 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.

#### 4.4.12 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.

#### 4.4.13 enum scg\_sosc\_mode\_t

Enumerator

- kSCG\_SysOscModeExt* Use external clock.
- kSCG\_SysOscModeOscLowPower* Oscillator low power.
- kSCG\_SysOscModeOscHighGain* Oscillator high gain.

#### 4.4.14 enum \_scg\_sosc\_enable\_mode

Enumerator

- kSCG\_SysOscEnable* Enable OSC clock.
- kSCG\_SysOscEnableInStop* Enable OSC in stop mode.
- kSCG\_SysOscEnableInLowPower* Enable OSC in low power mode.

#### 4.4.15 enum scg\_sirc\_range\_t

Enumerator

- kSCG\_SircRangeLow* Slow IRC low range clock (2 MHz, 4 MHz for i.MX 7ULP).

***kSCG\_SircRangeHigh*** Slow IRC high range clock (8 MHz, 16 MHz for i.MX 7ULP).

#### 4.4.16 enum \_scg\_sirc\_enable\_mode

Enumerator

***kSCG\_SircEnable*** Enable SIRC clock.

***kSCG\_SircEnableInStop*** Enable SIRC in stop mode.

***kSCG\_SircEnableInLowPower*** Enable SIRC in low power mode.

#### 4.4.17 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.

#### 4.4.18 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.

#### 4.4.19 enum scg\_firc\_trim\_src\_t

Enumerator

***kSCG\_FircTrimSrcUsb0*** USB0 start of frame (1kHz).

***kSCG\_FircTrimSrcUsb1*** USB1 start of frame (1kHz).

***kSCG\_FircTrimSrcSysOsc*** System OSC.

***kSCG\_FircTrimSrcRtcOsc*** RTC OSC (32.768 kHz).

#### 4.4.20 enum scg\_firc\_range\_t

Enumerator

*kSCG\_FircRange48M* Fast IRC is trimmed to 48 MHz.

*kSCG\_FircRange52M* Fast IRC is trimmed to 52 MHz.

*kSCG\_FircRange56M* Fast IRC is trimmed to 56 MHz.

*kSCG\_FircRange60M* Fast IRC is trimmed to 60 MHz.

#### 4.4.21 enum \_scg\_firc\_enable\_mode

Enumerator

*kSCG\_FircEnable* Enable FIRC clock.

*kSCG\_FircEnableInStop* Enable FIRC in stop mode.

#### 4.4.22 enum scg\_pll\_src\_t

Enumerator

*kSCG\_SysPllSrcSysOsc* System PLL clock source is system OSC.

*kSCG\_SysPllSrcFirc* System PLL clock source is fast IRC.

#### 4.4.23 enum \_scg\_pll\_enable\_mode

Enumerator

*kSCG\_SysPllEnable* Enable SPLL clock.

*kSCG\_SysPllEnableInStop* Enable SPLL in stop mode.

#### 4.4.24 enum scg\_pll\_pfd\_clkout\_t

Enumerator

*kSCG\_SysPllPfd0Clk* PFD0 output clock selected.

*kSCG\_SysPllPfd1Clk* PFD1 output clock selected.

*kSCG\_SysPllPfd2Clk* PFD2 output clock selected.

*kSCG\_SysPllPfd3Clk* PFD3 output clock selected.

#### 4.4.25 enum scg\_rosc\_monitor\_mode\_t

Enumerator

*kSCG\_rtcOscMonitorDisable* Monitor disable.

*kSCG\_rtcOscMonitorInt* Interrupt when the RTC OSC error is detected.

*kSCG\_rtcOscMonitorReset* Reset when the RTC OSC error is detected.

#### 4.4.26 enum scg\_apll\_src\_t

Enumerator

*kSCG\_AuxPllSrcSysOsc* Auxiliary PLL clock source is the system OSC.

*kSCG\_AuxPllSrcFirc* Auxiliary PLL clock source is the fast IRC.

#### 4.4.27 enum \_scg\_apll\_enable\_mode

Enumerator

*kSCG\_AuxPllEnable* Enable APLL clock.

*kSCG\_AuxPllEnableInStop* Enable APLL in stop mode.

#### 4.4.28 enum scg\_apll\_pfd\_clkout\_t

Enumerator

*kSCG\_AuxPllPfd0Clk* PFD0 output clock selected.

*kSCG\_AuxPllPfd1Clk* PFD1 output clock selected.

*kSCG\_AuxPllPfd2Clk* PFD2 output clock selected.

*kSCG\_AuxPllPfd3Clk* PFD3 output clock selected.

### 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 <a href="#">clock_ip_name_t</a> . |
|-------------|--------------------------------------------------------------|

#### 4.5.2 static void CLOCK\_DisableClock ( *clock\_ip\_name\_t name* ) [inline], [static]

Parameters

|             |                                                               |
|-------------|---------------------------------------------------------------|
| <i>name</i> | Which clock to disable, see <a href="#">clock_ip_name_t</a> . |
|-------------|---------------------------------------------------------------|

#### 4.5.3 static bool CLOCK\_IsEnabledByOtherCore ( *clock\_ip\_name\_t name* ) [inline], [static]

Parameters

|             |                                                                  |
|-------------|------------------------------------------------------------------|
| <i>name</i> | Which peripheral to check, see <a href="#">clock_ip_name_t</a> . |
|-------------|------------------------------------------------------------------|

Returns

True if clock is already enabled, otherwise false.

#### 4.5.4 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 <a href="#">clock_ip_name_t</a> . |
| <i>src</i>  | Clock source to set.                                             |

#### 4.5.5 static void CLOCK\_SetIpSrcDiv ( *clock\_ip\_name\_t name*, *clock\_ip\_src\_t src*, *uint8\_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 <a href="#">clock_ip_name_t</a> . |
| <i>src</i>       | Clock source to set.                                             |
| <i>divValue</i>  | The divider value.                                               |
| <i>fracValue</i> | The fraction multiply value.                                     |

#### 4.5.6 `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

#### 4.5.7 `uint32_t CLOCK_GetCoreSysClkFreq ( void )`

Returns

Clock frequency in Hz.

#### 4.5.8 `uint32_t CLOCK_GetPlatClkFreq ( void )`

Returns

Clock frequency in Hz.

#### 4.5.9 `uint32_t CLOCK_GetExtClkFreq ( void )`

Returns

Clock frequency in Hz.

#### 4.5.10 `uint32_t CLOCK_GetBusClkFreq( void )`

Returns

Clock frequency in Hz.

#### 4.5.11 `uint32_t CLOCK_GetSlowClkFreq( void )`

Returns

Clock frequency in Hz.

#### 4.5.12 `uint32_t CLOCK_GetOsc32kClkFreq( void )`

Returns

Clock frequency in Hz.

#### 4.5.13 `uint32_t CLOCK_GetErClkFreq( void )`

Returns

Clock frequency in Hz.

#### 4.5.14 `uint32_t CLOCK_GetLvdsClkFreq( void )`

Returns

Clock frequency in Hz.

#### 4.5.15 `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 <a href="#">clock_ip_name_t</a> . |
|-------------|----------------------------------------------------------------|

Returns

Clock frequency value in hertz

#### 4.5.16 **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.

#### 4.5.17 **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. |
|---------------|-------------------------------|

#### 4.5.18 **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. |
|---------------|-------------------------------|

#### 4.5.19 static void CLOCK\_SetHsrunModeSysClkConfig ( const scg\_sys\_clk\_config\_t \* *config* ) [inline], [static]

This function sets the system clock configuration for HSRUN mode.

Parameters

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

#### 4.5.20 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. |
|---------------|-------------------------------|

#### 4.5.21 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.

#### 4.5.22 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.

#### 4.5.23 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.

#### 4.5.24 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.

**4.5.25 `uint32_t CLOCK_GetSysOscFreq( void )`**

## Returns

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

**4.5.26 `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.

**4.5.27 `static bool CLOCK_IsSysOscErr( void ) [inline], [static]`**

## Returns

True if the error occurs, false if not.

**4.5.28 `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. |
|-------------|----------------------|

#### 4.5.29 static bool CLOCK\_IsSysOscValid( void ) [inline], [static]

Returns

True if clock is valid, false if not.

#### 4.5.30 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.

#### 4.5.31 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.

#### 4.5.32 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.

#### 4.5.33 static void CLOCK\_EnableLpoPowerOption ( *bool enable* ) [inline], [static]

This function enables/disables the SCG slow IRC 1khz LPO clock in LLS/VLLSx modes.

## Parameters

|               |                                                                                                                                                           |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>enable</i> | Switcher of LPO Power Option which controls whether the 1 kHz LPO clock is enabled in LLS/VLLSx modes. "true" means to enable, "false" means not enabled. |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|

#### 4.5.34 uint32\_t CLOCK\_GetSircFreq ( void )

## Returns

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

#### 4.5.35 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.

#### 4.5.36 static bool CLOCK\_IsSircValid ( void ) [inline], [static]

Returns

True if clock is valid, false if not.

#### 4.5.37 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.

#### 4.5.38 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.

#### 4.5.39 **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.

#### 4.5.40 **uint32\_t CLOCK\_GetFircFreq ( void )**

Returns

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

#### 4.5.41 **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.

#### 4.5.42 static bool CLOCK\_IsFircErr( void ) [inline], [static]

Returns

True if the error occurs, false if not.

#### 4.5.43 static bool CLOCK\_IsFircValid( void ) [inline], [static]

Returns

True if clock is valid, false if not.

#### 4.5.44 uint32\_t CLOCK\_GetRtcOscFreq( void )

Returns

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

#### 4.5.45 static bool CLOCK\_IsRtcOscErr( void ) [inline], [static]

Returns

True if error occurs, false if not.

#### 4.5.46 static void CLOCK\_SetRtcOscMonitorMode( scg\_rosc\_monitor\_mode\_t mode ) [inline], [static]

This function sets the RTC 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. |
|-------------|----------------------|

#### 4.5.47 static bool CLOCK\_IsRtcOscValid( void ) [inline], [static]

Returns

True if the clock is valid, false if not.

## 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.

### 4.6.2 volatile uint32\_t g\_xtal32Freq

The XTAL32/EXTAL32 clock frequency in Hz. When the clock is set up, use the function CLOCK\_SetXtal32Freq to set the value in the clock driver.

This is important for the multicore platforms where only one core needs to set up the clock. All other cores need to call the CLOCK\_SetXtal32Freq to get a valid clock frequency.

### 4.6.3 volatile uint32\_t g\_lvdsFreq

The LVDS pad clock frequency in Hz. When the clock is set up, use the function CLOCK\_SetLvdsFreq to set the value in the clock driver.

# **Chapter 5**

## **IOMUXC: IOMUX Controller**

IOMUXC driver provides APIs for pin configuration. It also supports the miscellaneous functions integrated in IOMUXC.

## 5.1 System Clock Generator (SCG)

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

### 5.1.1 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.

#### 5.1.1.1 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.

#### 5.1.1.2 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.

### 5.1.1.3 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\(\)](#).

### 5.1.1.4 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.

### 5.1.1.5 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.

### 5.1.1.6 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()`.

### 5.1.1.7 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()`.

## 5.1.2 Typical use case

### 5.1.2.1 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

### 5.1.2.2 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

### 5.1.2.3 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 6

## ACMP: Analog Comparator Driver

### 6.1 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.

### 6.2 Typical use case

#### 6.2.1 Normal Configuration

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/acmp

#### 6.2.2 Interrupt Configuration

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/acmp

#### 6.2.3 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.*

## 6.3 Data Structure Documentation

### 6.3.1 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`

### 6.3.2 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

**6.3.3 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.

**6.3.4 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.

**6.3.5 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

**6.4 Macro Definition Documentation****6.4.1 #define FSL\_ACMP\_DRIVER\_VERSION (MAKE\_VERSION(2U, 0U, 6U))****6.4.2 #define CMP\_C0\_CFx\_MASK (CMP\_C0\_CFR\_MASK | CMP\_C0\_CFF\_MASK)**

## 6.5 Enumeration Type Documentation

### 6.5.1 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.

### 6.5.2 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.

### 6.5.3 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.

### 6.5.4 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.

### 6.5.5 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.

### 6.5.6 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.

### 6.5.7 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.

## 6.6 Function Documentation

### 6.6.1 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. |

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

Parameters

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

### 6.6.3 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. |
|---------------|------------------------------------------|

### 6.6.4 void ACMP\_Enable ( CMP\_Type \* *base*, bool *enable* )

Parameters

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

### 6.6.5 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. |

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

## Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | ACMP peripheral base address. |
| <i>enable</i> | True to enable DMA.           |

**6.6.7 void ACMP\_EnableWindowMode ( CMP\_Type \* *base*, bool *enable* )**

## Parameters

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

**6.6.8 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. |

### 6.6.9 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. |

### 6.6.10 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. |

### 6.6.11 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. |

### 6.6.12 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.

### 6.6.13 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. |

### 6.6.14 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.

### 6.6.15 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". |

### 6.6.16 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". |

### 6.6.17 uint32\_t ACMP\_GetStatusFlags ( CMP\_Type \* *base* )

Parameters

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

Returns

Status flags asserted mask. See "\_acmp\_status\_flags".

### 6.6.18 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 7

## CACHE: LMEM CACHE Memory Controller

### 7.1 Overview

The MCUXpresso SDK provides a peripheral driver for the CACHE Controller of MCUXpresso SDK devices.

The CACHE driver is created to help the user more easily operate the cache memory. The APIs for basic operations are including the following three levels: 1L. The L1 cache driver API. This level provides the level 1 caches controller drivers. The L1 caches in this arch is the previous the local memory controller (LMEM).

2L. The unified cache driver API. This level provides many APIs for unified cache driver APIs for combined L1 and L2 cache maintain operations. This is provided for SDK drivers (DMA, ENET, US-DHC, etc) which should do the cache maintenance in their transactional APIs. Because in this arch, there is no L2 cache so the unified cache driver API directly calls only L1 driver APIs.

### 7.2 Function groups

#### 7.2.1 L1 CACHE Operation

The L1 CACHE has both code cache and data cache. This function group provides two independent API groups for both code cache and data cache. There are Enable/Disable APIs for code cache and data cache control and cache maintenance operations as Invalidate/Clean/CleanInvalidate by all and by address range.

#### Macros

- #define `L1CODEBUSCACHE_LINESIZE_BYTE` FSL\_FEATURE\_L1ICACHE\_LINESIZE\_BY-  
TE  
*code bus cache line size is equal to system bus line size, so the unified I/D cache line size equals too.*
- #define `L1SYSTEMBUSCACHE_LINESIZE_BYTE` `L1CODEBUSCACHE_LINESIZE_BYTE`  
*The system bus CACHE line size is 16B = 128b.*

#### Driver version

- #define `FSL_CACHE_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 6)`)  
*cache driver version.*

#### Unified Cache Control for all caches

- static void `ICACHE_InvalidateByRange` (uint32\_t address, uint32\_t size\_byte)  
*Invalidate instruction cache by range.*
- static void `DCACHE_InvalidateByRange` (uint32\_t address, uint32\_t size\_byte)  
*Invalidate data cache by range.*

- static void **DCACHE\_CleanByRange** (uint32\_t address, uint32\_t size\_byte)  
*Clean data cache by range.*
- static void **DCACHE\_CleanInvalidateByRange** (uint32\_t address, uint32\_t size\_byte)  
*Cleans and Invalidates data cache by range.*

## 7.3 Macro Definition Documentation

### 7.3.1 #define FSL\_CACHE\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 6))

### 7.3.2 #define L1CODEBUSCACHE\_LINESIZE\_BYTE FSL\_FEATURE\_L1ICACHE\_LINESIZE\_BYTE

The code bus CACHE line size is 16B = 128b.

### 7.3.3 #define L1SYSTEMBUSCACHE\_LINESIZE\_BYTE L1CODEBUSCACHE\_LINESIZE\_BYTE

## 7.4 Function Documentation

### 7.4.1 static void ICACHE\_InvalidateByRange ( uint32\_t address, uint32\_t size\_byte ) [inline], [static]

Parameters

|                  |                                       |
|------------------|---------------------------------------|
| <i>address</i>   | The physical address.                 |
| <i>size_byte</i> | size of the memory to be invalidated. |

Note

Address and size should be aligned to 16-Byte due to the cache operation unit FSL\_FEATURE\_L1ICACHE\_LINESIZE\_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size\_byte, application should make sure the alignment or make sure the right operation order if the size\_byte is not aligned.

### 7.4.2 static void DCACHE\_InvalidateByRange ( uint32\_t address, uint32\_t size\_byte ) [inline], [static]

Parameters

|                  |                                       |
|------------------|---------------------------------------|
| <i>address</i>   | The physical address.                 |
| <i>size_byte</i> | size of the memory to be invalidated. |

Note

Address and size should be aligned to 16-Byte due to the cache operation unit FSL\_FEATURE\_L1DCACHE\_LINESIZE\_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size\_byte, application should make sure the alignment or make sure the right operation order if the size\_byte is not aligned.

#### 7.4.3 static void DCACHE\_CleanByRange ( **uint32\_t address, uint32\_t size\_byte** ) [inline], [static]

Parameters

|                  |                                   |
|------------------|-----------------------------------|
| <i>address</i>   | The physical address.             |
| <i>size_byte</i> | size of the memory to be cleaned. |

Note

Address and size should be aligned to 16-Byte due to the cache operation unit FSL\_FEATURE\_L1DCACHE\_LINESIZE\_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size\_byte, application should make sure the alignment or make sure the right operation order if the size\_byte is not aligned.

#### 7.4.4 static void DCACHE\_CleanInvalidateByRange ( **uint32\_t address, uint32\_t size\_byte** ) [inline], [static]

Parameters

|                  |                                                   |
|------------------|---------------------------------------------------|
| <i>address</i>   | The physical address.                             |
| <i>size_byte</i> | size of the memory to be Cleaned and Invalidated. |

Note

Address and size should be aligned to 16-Byte due to the cache operation unit FSL\_FEATURE\_L1DCACHE\_LINESIZE\_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size\_byte, application should make sure the alignment or make sure the right operation order if the size\_byte is not aligned.

# Chapter 8

## Common Driver

### 8.1 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_EPC_USAR` = 57,  
    `kStatusGroup_DMIC` = 58,  
    `kStatusGroup_SDIF` = 59,  
}

```

kStatusGroup_NETC = 166 }

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, 4, 0))  
*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()**

## 8.2 Macro Definition Documentation

8.2.1 `#define FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ 1`

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

8.2.3 `#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 |

8.2.4 `#define FSL_COMMON_DRIVER_VERSION (MAKE_VERSION(2, 4, 0))`

8.2.5 `#define DEBUG_CONSOLE_DEVICE_TYPE_NONE 0U`

8.2.6 `#define DEBUG_CONSOLE_DEVICE_TYPE_UART 1U`

8.2.7 `#define DEBUG_CONSOLE_DEVICE_TYPE_LPUART 2U`

8.2.8 `#define DEBUG_CONSOLE_DEVICE_TYPE_LPSCI 3U`

8.2.9 `#define DEBUG_CONSOLE_DEVICE_TYPE_USBCDC 4U`

8.2.10 `#define DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM 5U`

8.2.11 `#define DEBUG_CONSOLE_DEVICE_TYPE_IUART 6U`

8.2.12 `#define DEBUG_CONSOLE_DEVICE_TYPE_VUSART 7U`

8.2.13 `#define DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART 8U`

8.2.14 `#define DEBUG_CONSOLE_DEVICE_TYPE_SWO 9U`

8.2.15 `#define DEBUG_CONSOLE_DEVICE_TYPE_QSCI 10U`

8.2.16 `#define ARRAY_SIZE( x ) (sizeof(x) / sizeof((x)[0]))`

## 8.3 Typedef Documentation

8.3.1 `typedef int32_t status_t`

## 8.4 Enumeration Type Documentation

8.4.1 `enum _status_groups`

Enumerator

`kStatusGroup_Generic` Group number for generic status codes.

`kStatusGroup_FLASH` Group number for FLASH status codes.

`kStatusGroup_LPSCI` Group number for LPSCI 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 UART 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\_SDSPI* Group number for SDSPI status codes.  
*kStatusGroup\_FLEXIO\_I2S* Group number for FLEXIO I2S status codes.  
*kStatusGroup\_FLEXIO\_MCULCD* 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\_VBAT* Group number for VBAT 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\_HAL\_ADC\_SENSOR* Group number for HAL ADC SENSOR 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\_ELS\_PKC* Group number for ELS PKC 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.  
*kStatusGroup\_NETC* Group number for NETC status codes.

## 8.4.2 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.

## 8.5 Function Documentation

### 8.5.1 `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. |
|------------|-------------------|

### 8.5.2 void SDK\_Free ( void \* *ptr* )

Parameters

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

### 8.5.3 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 9

## CRC: Cyclic Redundancy Check Driver

### 9.1 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.

### 9.2 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.

### 9.3 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.

### 9.4 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.

## 9.5 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.*

## 9.6 Data Structure Documentation

### 9.6.1 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

## 9.7 Macro Definition Documentation

### 9.7.1 #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

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

Use CRC16-CCIT-FALSE as default.

## 9.8 Enumeration Type Documentation

### 9.8.1 enum crc\_bits\_t

Enumerator

*kCrcBits16* Generate 16-bit CRC code.

*kCrcBits32* Generate 32-bit CRC code.

### 9.8.2 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.

## 9.9 Function Documentation

### 9.9.1 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. |

### 9.9.2 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. |
|-------------|-------------------------|

### 9.9.3 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. |
|---------------|---------------------------------------|

### 9.9.4 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. |

### 9.9.5 `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.

### 9.9.6 `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 10

## DAC12: 12-bit Digital-to-Analog Converter Driver

### 10.1 Overview

The MCUXpresso SDK provides a peripheral driver for the 12-bit Digital-to-Analog Converter (DAC12) module of MCUXpresso SDK devices.

This DAC is the 12-bit resolution digital-to-analog converters with programmable reference generator output. Its output data items are loaded into a FIFO, so that various FIFO mode can be used to output the value for user-defined sequence.

The DAC driver provides a user-friendly interface to operate the DAC peripheral. The user can initialize/deinitialize the DAC driver, set data into FIFO, or enable the interrupt DMA for special events so that the hardware can process the DAC output data automatically. Also, the configuration for software and hardware trigger are also included in the driver.

### 10.2 Typical use case

#### 10.2.1 A simple use case to output the user-defined DAC12 value.

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/dac12

#### 10.2.2 Working with the trigger

Once more than one data is filled into the FIFO, the output pointer moves into configured mode when a trigger comes. This trigger can be from software or hardware, and moves one item for each trigger. Also, the interrupt/DMA event can be activated when the output pointer hits to the configured position.

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/dac12

### Files

- file [fsl\\_dac12.h](#)

### Data Structures

- struct [dac12\\_hardware\\_info\\_t](#)  
*DAC12 hardware information.* [More...](#)
- struct [dac12\\_config\\_t](#)  
*DAC12 module configuration.* [More...](#)

## Macros

- #define **DAC12\_CR\_W1C\_FLAGS\_MASK** (DAC\_CR\_OVFF\_MASK | DAC\_CR\_UDFF\_MASK)  
*Define "write 1 to clear" flags.*
- #define **DAC12\_CR\_ALL\_FLAGS\_MASK** (**DAC12\_CR\_W1C\_FLAGS\_MASK** | DAC\_CR\_WMF\_MASK | DAC\_CR\_NEMPTF\_MASK | DAC\_CR\_FULLF\_MASK)  
*Define all the flag bits in DACx\_CR register.*

## Enumerations

- enum **\_dac12\_status\_flags** {
   
    kDAC12\_OverflowFlag = DAC\_CR\_OVFF\_MASK,  
 kDAC12\_UnderflowFlag = DAC\_CR\_UDFF\_MASK,  
 kDAC12\_WatermarkFlag = DAC\_CR\_WMF\_MASK,  
 kDAC12\_NearlyEmptyFlag = DAC\_CR\_NEMPTF\_MASK,  
 kDAC12\_FullFlag = DAC\_CR\_FULLF\_MASK }
   
*DAC12 flags.*
- enum **\_dac12\_interrupt\_enable** {
   
    kDAC12\_UnderOrOverflowInterruptEnable = DAC\_CR\_UVIE\_MASK,  
 kDAC12\_WatermarkInterruptEnable = DAC\_CR\_WTMIE\_MASK,  
 kDAC12\_NearlyEmptyInterruptEnable = DAC\_CR\_EMPTIE\_MASK,  
 kDAC12\_FullInterruptEnable = DAC\_CR\_FULLIE\_MASK }
   
*DAC12 interrupts.*
- enum **dac12\_fifo\_size\_info\_t** {
   
    kDAC12\_FIFOSize2 = 0U,  
 kDAC12\_FIFOSize4 = 1U,  
 kDAC12\_FIFOSize8 = 2U,  
 kDAC12\_FIFOSize16 = 3U,  
 kDAC12\_FIFOSize32 = 4U,  
 kDAC12\_FIFOSize64 = 5U,  
 kDAC12\_FIFOSize128 = 6U,  
 kDAC12\_FIFOSize256 = 7U }
   
*DAC12 FIFO size information provided by hardware.*
- enum **dac12\_fifo\_work\_mode\_t** {
   
    kDAC12\_FIFODisabled = 0U,  
 kDAC12\_FIFOWorkAsNormalMode = 1U,  
 kDAC12\_FIFOWorkAsSwingMode = 2U }
   
*DAC12 FIFO work mode.*
- enum **dac12\_reference\_voltage\_source\_t** {
   
    kDAC12\_ReferenceVoltageSourceAlt1 = 0U,  
 kDAC12\_ReferenceVoltageSourceAlt2 = 1U }
   
*DAC12 reference voltage source.*
- enum **dac12\_fifo\_trigger\_mode\_t** {
   
    kDAC12\_FIFOTriggerByHardwareMode = 0U,  
 kDAC12\_FIFOTriggerBySoftwareMode = 1U }
   
*DAC12 FIFO trigger mode.*

- enum `dac12_reference_current_source_t` {
   
    `kDAC12_ReferenceCurrentSourceDisabled` = 0U,  
`kDAC12_ReferenceCurrentSourceAlt0` = 1U,  
`kDAC12_ReferenceCurrentSourceAlt1` = 2U,  
`kDAC12_ReferenceCurrentSourceAlt2` = 3U }
   
*DAC internal reference current source.*
- enum `dac12_speed_mode_t` {
   
    `kDAC12_SpeedLowMode` = 0U,  
`kDAC12_SpeedMiddleMode` = 1U,  
`kDAC12_SpeedHighMode` = 2U }
   
*DAC analog buffer speed mode for conversion.*

## Driver version

- #define `FSL_DAC12_DRIVER_VERSION` (`MAKE_VERSION`(2, 1, 0))
   
*DAC12 driver version 2.1.0.*

## Initialization and de-initialization

- void `DAC12_GetHardwareInfo` (`DAC_Type` \*base, `dac12_hardware_info_t` \*info)
   
*Get hardware information about this module.*
- void `DAC12_Init` (`DAC_Type` \*base, const `dac12_config_t` \*config)
   
*Initialize the DAC12 module.*
- void `DAC12_GetDefaultConfig` (`dac12_config_t` \*config)
   
*Initializes the DAC12 user configuration structure.*
- void `DAC12_Deinit` (`DAC_Type` \*base)
   
*De-initialize the DAC12 module.*
- static void `DAC12_Enable` (`DAC_Type` \*base, bool enable)
   
*Enable the DAC12's converter or not.*
- static void `DAC12_ResetConfig` (`DAC_Type` \*base)
   
*Reset all internal logic and registers.*
- static void `DAC12_ResetFIFO` (`DAC_Type` \*base)
   
*Reset the FIFO pointers.*

## Status

- static `uint32_t` `DAC12_GetStatusFlags` (`DAC_Type` \*base)
   
*Get status flags.*
- static void `DAC12_ClearStatusFlags` (`DAC_Type` \*base, `uint32_t` flags)
   
*Clear status flags.*

## Interrupts

- static void `DAC12_EnableInterrupts` (`DAC_Type` \*base, `uint32_t` mask)
   
*Enable interrupts.*
- static void `DAC12_DisableInterrupts` (`DAC_Type` \*base, `uint32_t` mask)
   
*Disable interrupts.*

## DMA control

- static void **DAC12\_EnableDMA** (DAC\_Type \*base, bool enable)  
*Enable DMA or not.*

## Functional feature

- static void **DAC12\_SetData** (DAC\_Type \*base, uint32\_t value)  
*Set data into the entry of FIFO buffer.*
- static void **DAC12\_DoSoftwareTrigger** (DAC\_Type \*base)  
*Do trigger the FIFO by software.*
- static uint32\_t **DAC12\_GetFIFOReadPointer** (DAC\_Type \*base)  
*Get the current read pointer of FIFO.*
- static uint32\_t **DAC12\_GetFIFOWritePointer** (DAC\_Type \*base)  
*Get the current write pointer of FIFO.*

## 10.3 Data Structure Documentation

### 10.3.1 struct dac12.hardware\_info\_t

#### Data Fields

- **dac12\_fifo\_size\_info\_t fifoSizeInfo**  
*The number of words in this device's DAC buffer.*

#### Field Documentation

##### (1) **dac12\_fifo\_size\_info\_t dac12.hardware\_info\_t::fifoSizeInfo**

### 10.3.2 struct dac12\_config\_t

Actually, the most fields are for FIFO buffer.

#### Data Fields

- **uint32\_t fifoWatermarkLevel**  
*FIFO's watermark, the max value can be the hardware FIFO size.*
- **dac12\_fifo\_work\_mode\_t fifoWorkMode**  
*FIFO's work mode about pointers.*
- **dac12\_reference\_voltage\_source\_t referenceVoltageSource**  
*Select the reference voltage source.*
- **dac12\_reference\_current\_source\_t referenceCurrentSource**  
*Select the trigger mode for FIFO.*
- **dac12\_speed\_mode\_t speedMode**  
*Select the speed mode for conversion.*
- **bool enableAnalogBuffer**  
*Enable analog buffer for high drive.*
- **uint32\_t currentReferenceInternalTrimValue**  
*Internal reference current trim value.*

**Field Documentation**

- (1) `uint32_t dac12_config_t::fifoWatermarkLevel`
- (2) `dac12_fifo_work_mode_t dac12_config_t::fifoWorkMode`
- (3) `dac12_reference_voltage_source_t dac12_config_t::referenceVoltageSource`
- (4) `dac12_reference_current_source_t dac12_config_t::referenceCurrentSource`

Select the reference current source.

- (5) `dac12_speed_mode_t dac12_config_t::speedMode`
- (6) `bool dac12_config_t::enableAnalogBuffer`
- (7) `uint32_t dac12_config_t::currentReferenceInternalTrimValue`

3-bit value is available.

## 10.4 Macro Definition Documentation

**10.4.1 #define FSL\_DAC12\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 0))**

**10.4.2 #define DAC12\_CR\_W1C\_FLAGS\_MASK (DAC\_CR\_OVFF\_MASK | DAC\_CR\_UDFF\_MASK)**

**10.4.3 #define DAC12\_CR\_ALL\_FLAGS\_MASK (DAC12\_CR\_W1C\_FLAGS\_MASK | DAC\_CR\_WMF\_MASK | DAC\_CR\_NEMPTF\_MASK | DAC\_CR\_FULLF\_MASK)**

## 10.5 Enumeration Type Documentation

### 10.5.1 enum \_dac12\_status\_flags

Enumerator

***kDAC12\_OverflowFlag*** FIFO overflow status flag, which indicates that more data has been written into FIFO than it can hold.

***kDAC12\_UnderflowFlag*** FIFO underflow status flag, which means that there is a new trigger after the FIFO is nearly empty.

***kDAC12\_WatermarkFlag*** FIFO watermark status flag, which indicates the remaining FIFO data is less than the watermark setting.

***kDAC12\_NearlyEmptyFlag*** FIFO nearly empty flag, which means there is only one data remaining in FIFO.

***kDAC12\_FullFlag*** FIFO full status flag, which means that the FIFO read pointer equals the write pointer, as the write pointer increase.

## 10.5.2 enum \_dac12\_interrupt\_enable

Enumerator

***kDAC12\_UnderOrOverflowInterruptEnable*** Underflow and overflow interrupt enable.

***kDAC12\_WatermarkInterruptEnable*** Watermark interrupt enable.

***kDAC12\_NearlyEmptyInterruptEnable*** Nearly empty interrupt enable.

***kDAC12\_FullInterruptEnable*** Full interrupt enable.

## 10.5.3 enum dac12\_fifo\_size\_info\_t

Enumerator

***kDAC12\_FIFOSize2*** FIFO depth is 2.

***kDAC12\_FIFOSize4*** FIFO depth is 4.

***kDAC12\_FIFOSize8*** FIFO depth is 8.

***kDAC12\_FIFOSize16*** FIFO depth is 16.

***kDAC12\_FIFOSize32*** FIFO depth is 32.

***kDAC12\_FIFOSize64*** FIFO depth is 64.

***kDAC12\_FIFOSize128*** FIFO depth is 128.

***kDAC12\_FIFOSize256*** FIFO depth is 256.

## 10.5.4 enum dac12\_fifo\_work\_mode\_t

Enumerator

***kDAC12\_FIFODisabled*** FIFO disabled and only one level buffer is enabled. Any data written from this buffer goes to conversion.

***kDAC12\_FIFOWorkAsNormalMode*** Data will first read from FIFO to buffer then go to conversion.

***kDAC12\_FIFOWorkAsSwingMode*** In Swing mode, the FIFO must be set up to be full. In Swing back mode, a trigger changes the read pointer to make it swing between the FIFO Full and Nearly Empty state. That is, the trigger increases the read pointer till FIFO is nearly empty and decreases the read pointer till the FIFO is full.

## 10.5.5 enum dac12\_reference\_voltage\_source\_t

Enumerator

***kDAC12\_ReferenceVoltageSourceAlt1*** The DAC selects DACREF\_1 as the reference voltage.

***kDAC12\_ReferenceVoltageSourceAlt2*** The DAC selects DACREF\_2 as the reference voltage.

### 10.5.6 enum dac12\_fifo\_trigger\_mode\_t

Enumerator

*kDAC12\_FIFOTriggerByHardwareMode* Buffer would be triggered by hardware.

*kDAC12\_FIFOTriggerBySoftwareMode* Buffer would be triggered by software.

### 10.5.7 enum dac12\_reference\_current\_source\_t

Analog module needs reference current to keep working . Such reference current can generated by IP itself, or by on-chip PMC's "reference part". If no current reference be selected, analog module can't working normally ,even when other register can still be assigned, DAC would waste current but no function. To make the DAC work, either kDAC12\_ReferenceCurrentSourceAlt $x$  should be selected.

Enumerator

*kDAC12\_ReferenceCurrentSourceDisabled* None of reference current source is enabled.

*kDAC12\_ReferenceCurrentSourceAlt0* Use the internal reference current generated by the module itself.

*kDAC12\_ReferenceCurrentSourceAlt1* Use the ZTC(Zero Temperature Coefficient) reference current generated by on-chip power management module.

*kDAC12\_ReferenceCurrentSourceAlt2* Use the PTAT(Proportional To Absolution Temperature) reference current generated by power management module.

### 10.5.8 enum dac12\_speed\_mode\_t

Enumerator

*kDAC12\_SpeedLowMode* Low speed mode.

*kDAC12\_SpeedMiddleMode* Middle speed mode.

*kDAC12\_SpeedHighMode* High speed mode.

## 10.6 Function Documentation

### 10.6.1 void DAC12\_GetHardwareInfo ( DAC\_Type \* *base*, dac12\_hardware\_info\_t \* *info* )

## Parameters

|             |                                                                           |
|-------------|---------------------------------------------------------------------------|
| <i>base</i> | DAC12 peripheral base address.                                            |
| <i>info</i> | Pointer to info structure, see to <a href="#">dac12_hardware_info_t</a> . |

**10.6.2 void DAC12\_Init ( DAC\_Type \* *base*, const dac12\_config\_t \* *config* )**

## Parameters

|               |                                                                             |
|---------------|-----------------------------------------------------------------------------|
| <i>base</i>   | DAC12 peripheral base address.                                              |
| <i>config</i> | Pointer to configuration structure, see to <a href="#">dac12_config_t</a> . |

**10.6.3 void DAC12\_GetDefaultConfig ( dac12\_config\_t \* *config* )**

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

```
* config->fifoWatermarkLevel = 0U;
* config->fifoWorkMode = kDAC12_FIFODisabled;
* config->referenceVoltageSource = kDAC12_ReferenceVoltageSourceAlt1;
* config->fifoTriggerMode = kDAC12_FIFOTriggerByHardwareMode;
* config->referenceCurrentSource = kDAC12_ReferenceCurrentSourceAlt0;
* config->speedMode = kDAC12_SpeedLowMode;
* config->speedMode = false;
* config->currentReferenceInternalTrimValue = 0x4;
*
```

## Parameters

|               |                                                               |
|---------------|---------------------------------------------------------------|
| <i>config</i> | Pointer to the configuration structure. See "dac12_config_t". |
|---------------|---------------------------------------------------------------|

**10.6.4 void DAC12\_Deinit ( DAC\_Type \* *base* )**

## Parameters

|             |                                |
|-------------|--------------------------------|
| <i>base</i> | DAC12 peripheral base address. |
|-------------|--------------------------------|

**10.6.5 static void DAC12\_Enable ( DAC\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

|               |                                      |
|---------------|--------------------------------------|
| <i>base</i>   | DAC12 peripheral base address.       |
| <i>enable</i> | Enable the DAC12's converter or not. |

#### 10.6.6 static void DAC12\_ResetConfig ( DAC\_Type \* *base* ) [inline], [static]

Parameters

|             |                                |
|-------------|--------------------------------|
| <i>base</i> | DAC12 peripheral base address. |
|-------------|--------------------------------|

#### 10.6.7 static void DAC12\_ResetFIFO ( DAC\_Type \* *base* ) [inline], [static]

FIFO pointers should only be reset when the DAC12 is disabled. This function can be used to configure both pointers to the same address to reset the FIFO as empty.

Parameters

|             |                                |
|-------------|--------------------------------|
| <i>base</i> | DAC12 peripheral base address. |
|-------------|--------------------------------|

#### 10.6.8 static uint32\_t DAC12\_GetStatusFlags ( DAC\_Type \* *base* ) [inline], [static]

Parameters

|             |                                |
|-------------|--------------------------------|
| <i>base</i> | DAC12 peripheral base address. |
|-------------|--------------------------------|

Returns

Mask of current status flags. See to [\\_dac12\\_status\\_flags](#).

#### 10.6.9 static void DAC12\_ClearStatusFlags ( DAC\_Type \* *base*, uint32\_t *flags* ) [inline], [static]

Note: Not all the flags can be cleared by this API. Several flags need special condition to clear them according to target chip's reference manual document.

Parameters

|              |                                                                                  |
|--------------|----------------------------------------------------------------------------------|
| <i>base</i>  | DAC12 peripheral base address.                                                   |
| <i>flags</i> | Mask of status flags to be cleared. See to <a href="#">_dac12_status_flags</a> . |

#### 10.6.10 static void DAC12\_EnableInterrupts ( **DAC\_Type** \* *base*, **uint32\_t** *mask* ) [**inline**], [**static**]

Parameters

|             |                                                                                          |
|-------------|------------------------------------------------------------------------------------------|
| <i>base</i> | DAC12 peripheral base address.                                                           |
| <i>mask</i> | Mask value of interrupts to be enabled. See to <a href="#">_dac12_interrupt_enable</a> . |

#### 10.6.11 static void DAC12\_DisableInterrupts ( **DAC\_Type** \* *base*, **uint32\_t** *mask* ) [**inline**], [**static**]

Parameters

|             |                                                                                           |
|-------------|-------------------------------------------------------------------------------------------|
| <i>base</i> | DAC12 peripheral base address.                                                            |
| <i>mask</i> | Mask value of interrupts to be disabled. See to <a href="#">_dac12_interrupt_enable</a> . |

#### 10.6.12 static void DAC12\_EnableDMA ( **DAC\_Type** \* *base*, **bool** *enable* ) [**inline**], [**static**]

When DMA is enabled, the DMA request will be generated by original interrupts. The interrupts will not be presented on this module at the same time.

#### 10.6.13 static void DAC12\_SetData ( **DAC\_Type** \* *base*, **uint32\_t** *value* ) [**inline**], [**static**]

When the DAC FIFO is disabled, and the one entry buffer is enabled, the DAC converts the data in the buffer to analog output voltage. Any write to the DATA register will replace the data in the buffer and push data to analog conversion without trigger support. When the DAC FIFO is enabled. Writing data would increase the write pointer of FIFO. Also, the data would be restored into the FIFO buffer.

Parameters

|              |                                 |
|--------------|---------------------------------|
| <i>base</i>  | DAC12 peripheral base address.  |
| <i>value</i> | Setting value into FIFO buffer. |

#### 10.6.14 static void DAC12\_DoSoftwareTrigger ( **DAC\_Type** \* *base* ) [inline], [static]

When the DAC FIFO is enabled, and software trigger is used. Doing trigger would increase the read pointer, and the data in the entry pointed by read pointer would be converted as new output.

Parameters

|             |                                |
|-------------|--------------------------------|
| <i>base</i> | DAC12 peripheral base address. |
|-------------|--------------------------------|

#### 10.6.15 static uint32\_t DAC12\_GetFIFOReadPointer ( **DAC\_Type** \* *base* ) [inline], [static]

Parameters

|             |                                |
|-------------|--------------------------------|
| <i>base</i> | DAC12 peripheral base address. |
|-------------|--------------------------------|

Returns

Read pointer index of FIFO buffer.

#### 10.6.16 static uint32\_t DAC12\_GetFIFOWritePointer ( **DAC\_Type** \* *base* ) [inline], [static]

Parameters

|             |                                |
|-------------|--------------------------------|
| <i>base</i> | DAC12 peripheral base address. |
|-------------|--------------------------------|

Returns

Write pointer index of FIFO buffer

# Chapter 11

## DMAMUX: Direct Memory Access Multiplexer Driver

### 11.1 Overview

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

### 11.2 Typical use case

#### 11.2.1 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, 1, 0)`)  
*DMAMUX driver version 2.1.0.*

### 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, int32\_t source)  
*Configures the DMAMUX channel source.*

### 11.3 Macro Definition Documentation

#### 11.3.1 #define `FSL_DMAMUX_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 0)`)

### 11.4 Function Documentation

#### 11.4.1 void `DMAMUX_Init` ( `DMAMUX_Type * base` )

This function ungates the DMAMUX clock.

Parameters

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

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

This function gates the DMAMUX clock.

Parameters

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

#### 11.4.3 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.          |

#### 11.4.4 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. |
|----------------|------------------------|

#### 11.4.5 static void DMAMUX\_SetSource ( **DMAMUX\_Type** \* *base*, **uint32\_t** *channel*, **int32\_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. User need to use the dma_request_source_t type as the input parameter. |

# Chapter 12

## eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

### 12.1 Overview

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

### 12.2 Typical use case

#### 12.2.1 eDMA Operation

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_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_InstallTCMDMemory** (**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_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)
Prepares the eDMA transfer structure configurations.
- 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)
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.

12.3 Data Structure Documentation

12.3.1 struct edma_config_t

Data Fields

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

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.

12.3.2 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`

12.3.3 struct edma_channel_Preemption_config_t**Data Fields**

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

12.3.4 struct edma_minor_offset_config_t**Data Fields**

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

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**

12.3.5 struct edma_tcd_t

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

Data Fields

- **_IO uint32_t SADDR**
SADDR register, used to save source address.
- **_IO uint16_t SOFF**
SOFF register, save offset bytes every transfer.
- **_IO uint16_t ATTR**
ATTR register, source/destination transfer size and modulo.
- **_IO uint32_t NBYTES**
Nbytes register, minor loop length in bytes.
- **_IO uint32_t SLAST**
SLAST register.
- **_IO uint32_t DADDR**
DADDR register, used for destination address.
- **_IO uint16_t DOFF**
DOFF register, used for destination offset.
- **_IO uint16_t CITER**
CITER register, current minor loop numbers, for unfinished minor loop.
- **_IO uint32_t DLAST_SGA**
DLASTSGA register, next 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`

12.3.6 struct edma_handle_t**Data Fields**

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

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****12.4 Macro Definition Documentation****12.4.1 #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 4, 3))**

Version 2.4.3.

12.5 Typedef Documentation**12.5.1 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.
-------------	--

12.6 Enumeration Type Documentation

12.6.1 enum edma_transfer_size_t

Enumerator

- kEDMA_TransferSize1Bytes* Source/Destination data transfer size is 1 byte every time.
- kEDMA_TransferSize2Bytes* Source/Destination data transfer size is 2 bytes every time.
- kEDMA_TransferSize4Bytes* Source/Destination data transfer size is 4 bytes every time.
- kEDMA_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.

12.6.2 enum edma_modulo_t

Enumerator

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

12.6.3 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.

12.6.4 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.

12.6.5 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

12.6.6 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.

12.6.7 enum edma_interrupt_enable_t

Enumerator

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

12.6.8 enum edma_transfer_type_t

Enumerator

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

12.6.9 anonymous enum

Enumerator

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

12.7 Function Documentation

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

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

Parameters

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

Note

This function enables the minor loop map feature.

12.7.2 void EDMA_Deinit (DMA_Type * *base*)

This function gates the eDMA clock.

Parameters

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

12.7.3 void EDMA_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.

12.7.4 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.
---------------	--

12.7.5 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.

12.7.6 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.

12.7.7 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.

12.7.8 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.

12.7.9 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.

12.7.10 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.

12.7.11 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.

12.7.12 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

12.7.13 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.

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

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

Parameters

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

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

Parameters

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

12.7.16 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.
-------------	---

12.7.17 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.

12.7.18 void EDMA_TcdReset (edma_tcd_t * *tcd*)

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

Parameters

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

Note

This function enables the auto stop request feature.

12.7.19 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.

12.7.20 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.

12.7.21 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.

12.7.22 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

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

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

Parameters

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

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

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

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

Parameters

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

12.7.25 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 <i>edma_interrupt_enable_t</i> type.

12.7.26 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 <i>edma_interrupt_enable_t</i> type.

12.7.27 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.

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

This function enables the hardware channel request.

Parameters

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

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

This function disables the hardware channel request.

Parameters

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

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

This function starts a minor loop transfer.

Parameters

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

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

12.7.31 **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)

12.7.32 **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.

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

Parameters

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

Returns

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

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

Parameters

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

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

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

Parameters

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

12.7.36 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 tclds 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.

12.7.37 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.

**12.7.38 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 edma_transfer_t.
<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).

12.7.39 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 edma_transfer_t.
<i>srcAddr</i>	eDMA transfer source address.
<i>srcWidth</i>	eDMA transfer source address width(bytes).
<i>destAddr</i>	eDMA transfer destination address.
<i>destWidth</i>	eDMA transfer destination address width(bytes).
<i>bytesEachRequest</i>	eDMA transfer bytes per channel request.

<i>transferBytes</i>	eDMA transfer bytes to be transferred.
<i>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).

12.7.40 **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.

12.7.41 **void EDMA_StartTransfer (edma_handle_t * *handle*)**

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

Parameters

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

12.7.42 **void EDMA_StopTransfer (edma_handle_t * *handle*)**

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

Parameters

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

12.7.43 void EDMA_AbortTransfer(**edma_handle_t * handle**)

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

Parameters

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

12.7.44 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.

12.7.45 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.

12.7.46 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 13

EWM: External Watchdog Monitor Driver

13.1 Overview

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

13.2 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.

13.3 Data Structure Documentation

13.3.1 struct ewm_config_t

Data structure for EWM configuration.

This structure is used to configure the EWM.

Data Fields

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

13.4 Macro Definition Documentation

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

13.5 Enumeration Type Documentation

13.5.1 enum _ewm_interrupt_enable_t

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

Enumerator

kEWM InterruptEnable Enable the EWM to generate an interrupt.

13.5.2 enum _ewm_status_flags_t

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

Enumerator

kEWM_RunningFlag Running flag, set when EWM is enabled.

13.6 Function Documentation

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

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

This is an example.

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

Parameters

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

13.6.2 void EWM_Deinit (EWM_Type * *base*)

This function is used to shut down the EWM.

Parameters

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

13.6.3 void EWM_GetDefaultConfig (ewm_config_t * *config*)

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

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

Parameters

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

See Also

[ewm_config_t](#)

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

This function enables the EWM interrupt.

Parameters

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

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

This function enables the EWM interrupt.

Parameters

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

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

This function gets all status flags.

This is an example for getting the running flag.

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

Parameters

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

Returns

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

See Also

[_ewm_status_flags_t](#)

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

13.6.7 void EWM_Refresh (EWM_Type * *base*)

This function resets the EWM counter to zero.

Parameters

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

Chapter 14

FlexIO: FlexIO Driver

14.1 Overview

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

Modules

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

14.2 FlexIO Driver

14.2.1 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`,

 `kFLEXIO_TimerDecSrcOnPinInputShiftPinInput`,

 `kFLEXIO_TimerDecSrcOnTriggerInputShiftTriggerInput` }

 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_timer_output_state_t` {

kFLEXIO_PwmLow = 0,

kFLEXIO_PwmHigh }

- FlexIO as PWM channel output state.*
- 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, 2, 0))

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 **FLEXIO.GetInstance** (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.
- 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.
- static void **FLEXIO_SetClockMode** (FLEXIO_Type *base, uint8_t index, flexio_timer_decrement_source_t clocksource)

This function set the value of the prescaler on flexio channels.

FlexIO Interrupt Operation

- static void **FLEXIO_EnableShifterStatusInterrupts** (FLEXIO_Type *base, uint32_t mask)

Enables the shifter status interrupt.
- static void **FLEXIO_DisableShifterStatusInterrupts** (FLEXIO_Type *base, uint32_t mask)

Disables the shifter status interrupt.

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

FlexIO Status Operation

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

14.2.2 Data Structure Documentation

14.2.2.1 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`

14.2.2.2 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`

14.2.2.3 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`

14.2.3 Macro Definition Documentation

14.2.3.1 `#define FSL_FLEXIO_DRIVER_VERSION (MAKE_VERSION(2, 2, 0))`

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

14.2.4 Typedef Documentation

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

14.2.5 Enumeration Type Documentation

14.2.5.1 `enum flexio_timer_trigger_polarity_t`

Enumerator

kFLEXIO_TimerTriggerPolarityActiveHigh Active high.

kFLEXIO_TimerTriggerPolarityActiveLow Active low.

14.2.5.2 enum flexio_timer_trigger_source_t

Enumerator

kFLEXIO_TimerTriggerSourceExternal External trigger selected.

kFLEXIO_TimerTriggerSourceInternal Internal trigger selected.

14.2.5.3 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.

14.2.5.4 enum flexio_pin_polarity_t

Enumerator

kFLEXIO_PinActiveHigh Active high.

kFLEXIO_PinActiveLow Active low.

14.2.5.5 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.

14.2.5.6 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.

14.2.5.7 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.

14.2.5.8 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.

14.2.5.9 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.

14.2.5.10 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.

14.2.5.11 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.

14.2.5.12 enum flexio_timer_start_bit_condition_t

Enumerator

kFLEXIO_TimerStartBitDisabled Start bit disabled.

kFLEXIO_TimerStartBitEnabled Start bit enabled.

14.2.5.13 enum flexio_timer_output_state_t

Enumerator

kFLEXIO_PwmLow The output state of PWM channel is low.

kFLEXIO_PwmHigh The output state of PWM channel is high.

14.2.5.14 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.

14.2.5.15 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.

14.2.5.16 enum flexio_shifter_input_source_t

Enumerator

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

14.2.5.17 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.

14.2.5.18 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.

14.2.5.19 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.

14.2.6 Function Documentation

14.2.6.1 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
-------------------	---

14.2.6.2 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

14.2.6.3 void FLEXIO_Deinit (FLEXIO_Type * *base*)

Call this API to stop the FlexIO clock.

Note

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

Parameters

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

14.2.6.4 `uint32_t FLEXIO_GetInstance (FLEXIO_Type * base)`

Parameters

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

14.2.6.5 `void FLEXIO_Reset (FLEXIO_Type * base)`

Parameters

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

14.2.6.6 `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.

14.2.6.7 `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

14.2.6.8 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

14.2.6.9 static void FLEXIO_SetClockMode (FLEXIO_Type * *base*, uint8_t *index*, flexio_timer_decrement_source_t *clocksource*) [inline], [static]

Parameters

<i>base</i>	Pointer to the FlexIO simulated peripheral type.
<i>clocksource</i>	Set clock value

14.2.6.10 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 ($1 << \text{shifter index}$)

Note

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

14.2.6.11 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 ($1 << \text{shifter index}$)

Note

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

14.2.6.12 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 ($1 << \text{shifter index}$)

Note

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

14.2.6.13 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 ($1 << \text{shifter index}$)

Note

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

14.2.6.14 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 ($1 << \text{timer index}$)

Note

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

**14.2.6.15 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 $((1 << \text{timer index}0) | (1 << \text{timer index}1))$

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

Parameters

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

Returns

Shifter status flags

14.2.6.17 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 $((1 << \text{shifter index}0) | (1 << \text{shifter index}1))$

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

Parameters

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

Returns

Shifter error flags

14.2.6.19 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))

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

Parameters

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

Returns

Timer status flags

14.2.6.21 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 $((1 << \text{timer index}0) | (1 << \text{timer index}1))$

14.2.6.22 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.

14.2.6.23 **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

14.2.6.24 **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.

14.2.6.25 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.

14.2.7 Variable Documentation

14.2.7.1 FLEXIO_Type* const s_flexioBases[]

14.2.7.2 const clock_ip_name_t s_flexioClocks[]

14.3 FlexIO I2C Master Driver

14.3.1 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.

14.3.2 Typical use case

14.3.2.1 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, 5, 0))

## Initialization and deinitialization

- 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.*

### 14.3.3 Data Structure Documentation

#### 14.3.3.1 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**

#### 14.3.3.2 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`

### 14.3.3.3 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`

#### 14.3.3.4 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`

#### 14.3.4 Macro Definition Documentation

**14.3.4.1 `#define I2C_RETRY_TIMES 0U /* Define to zero means keep waiting until the flag is assert/deassert. */`**

#### 14.3.5 Typedef Documentation

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

#### 14.3.6 Enumeration Type Documentation

##### 14.3.6.1 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.

##### 14.3.6.2 enum \_flexio\_i2c\_master\_interrupt

Enumerator

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

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

##### 14.3.6.3 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.

#### 14.3.6.4 enum flexio\_i2c\_direction\_t

Enumerator

- kFLEXIO\_I2C\_Write*** Master send to slave.
- kFLEXIO\_I2C\_Read*** Master receive from slave.

#### 14.3.7 Function Documentation

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

Return values

|                                |                                                |
|--------------------------------|------------------------------------------------|
| <i>kStatus_Success</i>         | Initialization successful                      |
| <i>kStatus_InvalidArgument</i> | The source clock exceed upper range limitation |

##### 14.3.7.2 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. |
|-------------|-------------------------------------------------------|

#### 14.3.7.3 void [FLEXIO\\_I2C\\_MasterGetDefaultConfig](#) ( [flexio\\_i2c\\_master\\_config\\_t](#) \* *masterConfig* )

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

Example:

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

Parameters

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

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

Parameters

|               |                                                             |
|---------------|-------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.       |
| <i>enable</i> | Pass true to enable module, false does not have any effect. |

#### 14.3.7.5 [uint32\\_t FLEXIO\\_I2C\\_MasterGetStatusFlags](#) ( [FLEXIO\\_I2C\\_Type](#) \* *base* )

Parameters

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

Returns

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

|                    |                    |
|--------------------|--------------------|
| <i>srcClock_Hz</i> | source clock in HZ |
|--------------------|--------------------|

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

Note

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

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> |

#### 14.3.7.11 void FLEXIO\_I2C\_MasterStop ( FLEXIO\_I2C\_Type \* *base* )

Parameters

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

#### 14.3.7.12 void FLEXIO\_I2C\_MasterRepeatedStart ( FLEXIO\_I2C\_Type \* *base* )

Parameters

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

#### 14.3.7.13 void FLEXIO\_I2C\_MasterAbortStop ( FLEXIO\_I2C\_Type \* *base* )

Parameters

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

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

Parameters

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

#### 14.3.7.15 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.         |

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

Note

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

Parameters

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

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

Note

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

Parameters

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

Returns

data byte read.

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

Note

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

Parameters

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

Return values

---

|                                   |                                  |
|-----------------------------------|----------------------------------|
| <i>kStatus_Success</i>            | Successfully write data.         |
| <i>kStatus_FLEXIO_I2C_Nak</i>     | Receive NAK during writing data. |
| <i>kStatus_FLEXIO_I2C_Timeout</i> | Timeout polling status flags.    |

#### 14.3.7.19 **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_Timeout</i> | Timeout polling status flags. |

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

Note

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

Parameters

|             |                                                       |
|-------------|-------------------------------------------------------|
| <i>base</i> | pointer to <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.

#### 14.3.7.21 status\_t FLEXIO\_I2C\_MasterTransferCreateHandle ( ***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. |

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

Note

The API returns immediately after the transfer initiates. Call [FLEXIO\\_I2C\\_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. |

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

Parameters

|               |                                                                                                  |
|---------------|--------------------------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2C_Type</a> structure.                                            |
| <i>handle</i> | Pointer to <a href="#">flexio_i2c_master_handle_t</a> structure which stores the transfer state. |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction.                              |

Return values

|                                     |                                                                |
|-------------------------------------|----------------------------------------------------------------|
| <i>kStatus_InvalidArgument</i>      | count is Invalid.                                              |
| <i>kStatus_NoTransferInProgress</i> | There is not a non-blocking transaction currently in progress. |
| <i>kStatus_Success</i>              | Successfully return the count.                                 |

#### 14.3.7.24 **void FLEXIO\_I2C\_MasterTransferAbort ( *FLEXIO\_I2C\_Type \* base, flexio\_i2c\_master\_handle\_t \* handle* )**

Note

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

Parameters

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

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| <i>handle</i> | Pointer to flexio_i2c_master_handle_t structure which stores the transfer state |
|---------------|---------------------------------------------------------------------------------|

#### 14.3.7.25 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 |

## 14.4 FlexIO I2S Driver

### 14.4.1 Overview

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

The FlexIO I2S driver includes functional APIs and transactional APIs.

Functional APIs are feature/property target low level APIs.

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

Transactional APIs are transaction target high level APIs. The 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 sai\_handle\_t as the first parameter. Initialize the handle by calling the FlexIO\_I2S\_TransferTxCreateHandle() or FlexIO\_I2S\_TransferRxCreateHandle() API.

Transactional APIs support asynchronous transfer. This means that the functions [FLEXIO\\_I2S\\_TransferSendNonBlocking\(\)](#) and [FLEXIO\\_I2S\\_TransferReceiveNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the kStatus\_FLEXIO\_I2S\_TxIdle and kStatus\_FLEXIO\_I2S\_RxIdle status.

### 14.4.2 Typical use case

#### 14.4.2.1 FlexIO I2S send/receive using an interrupt method

```
sai_handle_t g_saiTxHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
volatile bool rxFinished;
const uint8_t sendData[] = [.....];

void FLEXIO_I2S_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
 userData = userData;

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

void main(void)
{
 //...

 FLEXIO_I2S_TxGetDefaultConfig(&user_config);
```

```

FLEXIO_I2S_TxInit(FLEXIO_I2S0, &user_config);
FLEXIO_I2S_TransferTxCreateHandle(FLEXIO_I2S0, &g_saiHandle,
 FLEXIO_I2S_UserCallback, NULL);

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

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

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

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

// ...
}

```

#### 14.4.2.2 FLEXIO\_I2S send/receive using a DMA method

```

sai_handle_t g_saiHandle;
dma_handle_t g_saiTxDmaHandle;
dma_handle_t g_saiRxDmaHandle;
sai_config_t user_config;
sai_transfer_t sendXfer;
volatile bool txFinished;
uint8_t sendData[] = ...;

void FLEXIO_I2S_UserCallback(sai_handle_t *handle, status_t status, void *userData)
{
 userData = userData;

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

void main(void)
{
 //...

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

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

 DMA_Init(DMA0);

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

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

 // Prepares to send.
 sendXfer.data = sendData
}

```

```

sendXfer.dataSize = sizeof(sendData)/sizeof(sendData[0]);
txFinished = false;

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

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

// ...
}

```

## Modules

- FlexIO eDMA I2S Driver

## Data Structures

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

## Macros

- #define **I2S\_RETRY\_TIMES** 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/  
*Retry times for waiting flag.*
- #define **FLEXIO\_I2S\_XFER\_QUEUE\_SIZE** (4U)  
*FlexIO I2S transfer queue size, user can refine it according to use case.*

## Typedefs

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

## Enumerations

- enum {
   
kStatus\_FLEXIO\_I2S\_Idle = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 0),
   
kStatus\_FLEXIO\_I2S\_TxBusy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 1),
   
kStatus\_FLEXIO\_I2S\_RxBusy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 2),
   
kStatus\_FLEXIO\_I2S\_Error = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 3),
   
kStatus\_FLEXIO\_I2S\_QueueFull = MAKE\_STATUS(kStatusGroup\_FLEXIO\_I2S, 4),
   
kStatus\_FLEXIO\_I2S\_Timeout }
   
*FlexIO I2S transfer status.*
- enum **flexio\_i2s\_master\_slave\_t** {
   
kFLEXIO\_I2S\_Master = 0x0U,
   
kFLEXIO\_I2S\_Slave = 0x1U }
   
*Master or slave mode.*
- enum {
   
kFLEXIO\_I2S\_TxDataRegEmptyInterruptEnable = 0x1U,
   
kFLEXIO\_I2S\_RxDataRegFullInterruptEnable = 0x2U }
   
*\_flexio\_i2s\_interrupt\_enable Define FlexIO I2S interrupt mask.*
- enum {
   
kFLEXIO\_I2S\_TxDataRegEmptyFlag = 0x1U,
   
kFLEXIO\_I2S\_RxDataRegFullFlag = 0x2U }
   
*\_flexio\_i2s\_status\_flags Define FlexIO I2S status mask.*
- enum **flexio\_i2s\_sample\_rate\_t** {
   
kFLEXIO\_I2S\_SampleRate8KHz = 8000U,
   
kFLEXIO\_I2S\_SampleRate11025Hz = 11025U,
   
kFLEXIO\_I2S\_SampleRate12KHz = 12000U,
   
kFLEXIO\_I2S\_SampleRate16KHz = 16000U,
   
kFLEXIO\_I2S\_SampleRate22050Hz = 22050U,
   
kFLEXIO\_I2S\_SampleRate24KHz = 24000U,
   
kFLEXIO\_I2S\_SampleRate32KHz = 32000U,
   
kFLEXIO\_I2S\_SampleRate44100Hz = 44100U,
   
kFLEXIO\_I2S\_SampleRate48KHz = 48000U,
   
kFLEXIO\_I2S\_SampleRate96KHz = 96000U }
   
*Audio sample rate.*
- enum **flexio\_i2s\_word\_width\_t** {
   
kFLEXIO\_I2S\_WordWidth8bits = 8U,
   
kFLEXIO\_I2S\_WordWidth16bits = 16U,
   
kFLEXIO\_I2S\_WordWidth24bits = 24U,
   
kFLEXIO\_I2S\_WordWidth32bits = 32U }
   
*Audio word width.*

## Driver version

- #define **FSL\_FLEXIO\_I2S\_DRIVER\_VERSION** (MAKE\_VERSION(2, 2, 0))
   
*FlexIO I2S driver version 2.2.0.*

## Initialization and deinitialization

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

## Status

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

## Interrupts

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

## DMA Control

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

## Bus Operations

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

- Writes data into a data register.  
 • `status_t FLEXIO_I2S_ReadBlocking (FLEXIO_I2S_Type *base, uint8_t bitWidth, uint8_t *rxData, size_t size)`  
*Receives a piece of data using a blocking method.*
- static `uint32_t FLEXIO_I2S_ReadData (FLEXIO_I2S_Type *base)`  
*Reads a data from the data register.*

## Transactional

- void `FLEXIO_I2S_TransferTxCreateHandle (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, flexio_i2s_callback_t callback, void *userData)`  
*Initializes the FlexIO I2S handle.*
- void `FLEXIO_I2S_TransferSetFormat (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, flexio_i2s_format_t *format, uint32_t srcClock_Hz)`  
*Configures the FlexIO I2S audio format.*
- void `FLEXIO_I2S_TransferRxCreateHandle (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, flexio_i2s_callback_t callback, void *userData)`  
*Initializes the FlexIO I2S receive handle.*
- `status_t FLEXIO_I2S_TransferSendNonBlocking (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, flexio_i2s_transfer_t *xfer)`  
*Performs an interrupt non-blocking send transfer on FlexIO I2S.*
- `status_t FLEXIO_I2S_TransferReceiveNonBlocking (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, flexio_i2s_transfer_t *xfer)`  
*Performs an interrupt non-blocking receive transfer on FlexIO I2S.*
- void `FLEXIO_I2S_TransferAbortSend (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle)`  
*Aborts the current send.*
- void `FLEXIO_I2S_TransferAbortReceive (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle)`  
*Aborts the current receive.*
- `status_t FLEXIO_I2S_TransferGetSendCount (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, size_t *count)`  
*Gets the remaining bytes to be sent.*
- `status_t FLEXIO_I2S_TransferGetReceiveCount (FLEXIO_I2S_Type *base, flexio_i2s_handle_t *handle, size_t *count)`  
*Gets the remaining bytes to be received.*
- void `FLEXIO_I2S_TransferTxHandleIRQ (void *i2sBase, void *i2sHandle)`  
*Tx interrupt handler.*
- void `FLEXIO_I2S_TransferRxHandleIRQ (void *i2sBase, void *i2sHandle)`  
*Rx interrupt handler.*

### 14.4.3 Data Structure Documentation

#### 14.4.3.1 struct FLEXIO\_I2S\_Type

##### Data Fields

- `FLEXIO_Type * flexioBase`  
*FlexIO base pointer.*

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

#### 14.4.3.2 struct flexio\_i2s\_config\_t

##### Data Fields

- `bool enableI2S`  
*Enable FlexIO I2S.*
- `flexio_i2s_master_slave_t masterSlave`  
*Master or slave.*
- `flexio_pin_polarity_t txPinPolarity`  
*Tx data pin polarity, active high or low.*
- `flexio_pin_polarity_t rxPinPolarity`  
*Rx data pin polarity.*
- `flexio_pin_polarity_t bclkPinPolarity`  
*Bit clock pin polarity.*
- `flexio_pin_polarity_t fsPinPolarity`  
*Frame sync pin polarity.*
- `flexio_shifter_timer_polarity_t txTimerPolarity`  
*Tx data valid on bclk rising or falling edge.*
- `flexio_shifter_timer_polarity_t rxTimerPolarity`  
*Rx data valid on bclk rising or falling edge.*

#### 14.4.3.3 struct flexio\_i2s\_format\_t

##### Data Fields

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

#### 14.4.3.4 struct flexio\_i2s\_transfer\_t

##### Data Fields

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

##### Field Documentation

(1) `size_t flexio_i2s_transfer_t::dataSize`

#### 14.4.3.5 struct \_flexio\_i2s\_handle

##### Data Fields

- `uint32_t state`  
*Internal state.*
- `flexio_i2s_callback_t callback`  
*Callback function called at transfer event.*
- `void * userData`  
*Callback parameter passed to callback function.*
- `uint8_t bitWidth`  
*Bit width for transfer, 8/16/24/32bits.*
- `flexio_i2s_transfer_t queue [FLEXIO_I2S_XFER_QUEUE_SIZE]`  
*Transfer queue storing queued transfer.*
- `size_t transferSize [FLEXIO_I2S_XFER_QUEUE_SIZE]`  
*Data bytes need to transfer.*
- `volatile uint8_t queueUser`  
*Index for user to queue transfer.*
- `volatile uint8_t queueDriver`  
*Index for driver to get the transfer data and size.*

#### 14.4.4 Macro Definition Documentation

**14.4.4.1 #define FSL\_FLEXIO\_I2S\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 0))**

**14.4.4.2 #define I2S\_RETRY\_TIMES 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/**

**14.4.4.3 #define FLEXIO\_I2S\_XFER\_QUEUE\_SIZE (4U)**

#### 14.4.5 Enumeration Type Documentation

##### 14.4.5.1 anonymous enum

Enumerator

*kStatus\_FLEXIO\_I2S\_Idle* FlexIO I2S is in idle state.

*kStatus\_FLEXIO\_I2S\_TxBusy* FlexIO I2S Tx is busy.

*kStatus\_FLEXIO\_I2S\_RxBusy* FlexIO I2S Rx is busy.

*kStatus\_FLEXIO\_I2S\_Error* FlexIO I2S error occurred.

*kStatus\_FLEXIO\_I2S\_QueueFull* FlexIO I2S transfer queue is full.

*kStatus\_FLEXIO\_I2S\_Timeout* FlexIO I2S timeout polling status flags.

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

Enumerator

*kFLEXIO\_I2S\_Master* Master mode.

*kFLEXIO\_I2S\_Slave* Slave mode.

##### 14.4.5.3 anonymous enum

Enumerator

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

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

##### 14.4.5.4 anonymous enum

Enumerator

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

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

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

Enumerator

*kFLEXIO\_I2S\_SampleRate8KHz* Sample rate 8000Hz.  
*kFLEXIO\_I2S\_SampleRate11025Hz* Sample rate 11025Hz.  
*kFLEXIO\_I2S\_SampleRate12KHz* Sample rate 12000Hz.  
*kFLEXIO\_I2S\_SampleRate16KHz* Sample rate 16000Hz.  
*kFLEXIO\_I2S\_SampleRate22050Hz* Sample rate 22050Hz.  
*kFLEXIO\_I2S\_SampleRate24KHz* Sample rate 24000Hz.  
*kFLEXIO\_I2S\_SampleRate32KHz* Sample rate 32000Hz.  
*kFLEXIO\_I2S\_SampleRate44100Hz* Sample rate 44100Hz.  
*kFLEXIO\_I2S\_SampleRate48KHz* Sample rate 48000Hz.  
*kFLEXIO\_I2S\_SampleRate96KHz* Sample rate 96000Hz.

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

Enumerator

*kFLEXIO\_I2S\_WordWidth8bits* Audio data width 8 bits.  
*kFLEXIO\_I2S\_WordWidth16bits* Audio data width 16 bits.  
*kFLEXIO\_I2S\_WordWidth24bits* Audio data width 24 bits.  
*kFLEXIO\_I2S\_WordWidth32bits* Audio data width 32 bits.

### 14.4.6 Function Documentation

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

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

Note

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

Parameters

---

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

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

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

Parameters

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

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

Calling this API resets the FlexIO I2S shifter and timer config. After calling this API, call the [FLEXO\\_I2S\\_Init](#) to use the FlexIO I2S module.

Parameters

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

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

Parameters

|               |                                                 |
|---------------|-------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a>      |
| <i>enable</i> | True to enable, false dose not have any effect. |

#### 14.4.6.5 uint32\_t FLEXIO\_I2S\_GetStatusFlags ( FLEXIO\_I2S\_Type \* *base* )

Parameters

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

Returns

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

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

This function enables the FlexIO UART interrupt.

Parameters

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

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

This function disables the FlexIO UART interrupt.

Parameters

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

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

Parameters

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

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

Parameters

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

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

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

Parameters

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

Returns

FlexIO i2s send data register address.

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

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

Parameters

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

Returns

FlexIO i2s receive data register address.

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

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

Parameters

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

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

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

Parameters

|               |                                                      |
|---------------|------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_I2S_Type</a> structure |
| <i>format</i> | Pointer to FlexIO I2S audio data format structure.   |

#### 14.4.6.14 status\_t [FLEXIO\\_I2S\\_WriteBlocking](#) ( [FLEXIO\\_I2S\\_Type](#) \* *base*, [uint8\\_t](#) *bitWidth*, [uint8\\_t](#) \* *txData*, [size\\_t](#) *size* )

Note

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

Parameters

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

Return values

|                                   |                               |
|-----------------------------------|-------------------------------|
| <i>kStatus_Success</i>            | Successfully write data.      |
| <i>kStatus_FLEXIO_I2C_Timeout</i> | Timeout polling status flags. |

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

Parameters

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

#### 14.4.6.16 status\_t FLEXIO\_I2S\_ReadBlocking ( **FLEXIO\_I2S\_Type** \* *base*, **uint8\_t** *bitWidth*, **uint8\_t** \* *rxData*, **size\_t** *size* )

Note

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

Parameters

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

Return values

|                                   |                               |
|-----------------------------------|-------------------------------|
| <i>kStatus_Success</i>            | Successfully read data.       |
| <i>kStatus_FLEXIO_I2C_Timeout</i> | Timeout polling status flags. |

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

Parameters

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

Returns

Data read from data register.

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

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

Parameters

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

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

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

Parameters

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

**14.4.6.20 void FLEXIO\_I2S\_TransferRxCreateHandle ( [FLEXIO\\_I2S\\_Type](#) \* *base*, [flexio\\_i2s\\_handle\\_t](#) \* *handle*, [flexio\\_i2s\\_callback\\_t](#) *callback*, [void](#) \* *userData* )**

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

Parameters

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

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

## Note

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

## Parameters

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

## Return values

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

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

## Note

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

## Parameters

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

## Return values

|                        |                                      |
|------------------------|--------------------------------------|
| <i>kStatus_Success</i> | Successfully start the data receive. |
|------------------------|--------------------------------------|

|                                   |                                      |
|-----------------------------------|--------------------------------------|
| <i>kStatus_FLEXIO_I2S_-RxBusy</i> | Previous receive still not finished. |
| <i>kStatus_InvalidArgument</i>    | The input parameter is invalid.      |

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

Note

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

Parameters

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

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

Note

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

Parameters

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

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

Parameters

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

Return values

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

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

Parameters

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

Returns

*count* Bytes received.

Return values

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

#### 14.4.6.27 void FLEXIO\_I2S\_TransferTxHandleIRQ ( **void** \* *i2sBase*, **void** \* *i2sHandle* )

Parameters

|                  |                                                 |
|------------------|-------------------------------------------------|
| <i>i2sBase</i>   | Pointer to <b>FLEXIO_I2S_Type</b> structure.    |
| <i>i2sHandle</i> | Pointer to <b>flexio_i2s_handle_t</b> structure |

#### 14.4.6.28 void FLEXIO\_I2S\_TransferRxHandleIRQ ( **void** \* *i2sBase*, **void** \* *i2sHandle* )

## Parameters

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

## 14.4.7 FlexIO eDMA I2S Driver

### 14.4.7.1 Overview

#### Data Structures

- struct `flexio_i2s_edma_handle_t`

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

#### TypeDefs

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

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

#### Driver version

- #define `FSL_FLEXIO_I2S_EDMA_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 7)`)  
*FlexIO I2S EDMA driver version 2.1.7.*

#### eDMA Transactional

- void `FLEXIO_I2S_TransferTxCreateHandleEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_edma_callback_t` callback, void \*userData, `edma_handle_t` \*dmaHandle)  
*Initializes the FlexIO I2S eDMA handle.*
- void `FLEXIO_I2S_TransferRxCreateHandleEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_edma_callback_t` callback, void \*userData, `edma_handle_t` \*dmaHandle)  
*Initializes the FlexIO I2S Rx eDMA handle.*
- void `FLEXIO_I2S_TransferSetFormatEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_format_t` \*format, uint32\_t srcClock\_Hz)  
*Configures the FlexIO I2S Tx audio format.*
- `status_t FLEXIO_I2S_TransferSendEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO I2S transfer using DMA.*
- `status_t FLEXIO_I2S_TransferReceiveEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle, `flexio_i2s_transfer_t` \*xfer)  
*Performs a non-blocking FlexIO I2S receive using eDMA.*
- void `FLEXIO_I2S_TransferAbortSendEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle)  
*Aborts a FlexIO I2S transfer using eDMA.*
- void `FLEXIO_I2S_TransferAbortReceiveEDMA` (`FLEXIO_I2S_Type` \*base, `flexio_i2s_edma_handle_t` \*handle)  
*Aborts a FlexIO I2S receive using eDMA.*

- **status\_t FLEXIO\_I2S\_TransferGetSendCountEDMA** (**FLEXIO\_I2S\_Type** \*base, **flexio\_i2s\_edma\_handle\_t** \*handle, **size\_t** \*count)  
*Gets the remaining bytes to be sent.*
- **status\_t FLEXIO\_I2S\_TransferGetReceiveCountEDMA** (**FLEXIO\_I2S\_Type** \*base, **flexio\_i2s\_edma\_handle\_t** \*handle, **size\_t** \*count)  
*Get the remaining bytes to be received.*

#### 14.4.7.2 Data Structure Documentation

##### 14.4.7.2.1 struct \_flexio\_i2s\_edma\_handle

###### Data Fields

- **edma\_handle\_t \* dmaHandle**  
*DMA handler for FlexIO I2S send.*
- **uint8\_t bytesPerFrame**  
*Bytes in a frame.*
- **uint8\_t nbytes**  
*eDMA minor byte transfer count initially configured.*
- **uint32\_t state**  
*Internal state for FlexIO I2S eDMA transfer.*
- **flexio\_i2s\_edma\_callback\_t callback**  
*Callback for users while transfer finish or error occurred.*
- **void \* userData**  
*User callback parameter.*
- **edma\_tcd\_t tcd** [**FLEXIO\_I2S\_XFER\_QUEUE\_SIZE+1U**]  
*TCD pool for eDMA transfer.*
- **flexio\_i2s\_transfer\_t queue** [**FLEXIO\_I2S\_XFER\_QUEUE\_SIZE**]  
*Transfer queue storing queued transfer.*
- **size\_t transferSize** [**FLEXIO\_I2S\_XFER\_QUEUE\_SIZE**]  
*Data bytes need to transfer.*
- **volatile uint8\_t queueUser**  
*Index for user to queue transfer.*
- **volatile uint8\_t queueDriver**  
*Index for driver to get the transfer data and size.*

## Field Documentation

- (1) `uint8_t flexio_i2s_edma_handle_t::nbytes`
- (2) `edma_tcd_t flexio_i2s_edma_handle_t::tcd[FLEXIO_I2S_XFER_QUEUE_SIZE+1U]`
- (3) `flexio_i2s_transfer_t flexio_i2s_edma_handle_t::queue[FLEXIO_I2S_XFER_QUEUE_SIZE]`
- (4) `volatile uint8_t flexio_i2s_edma_handle_t::queueUser`

### 14.4.7.3 Macro Definition Documentation

**14.4.7.3.1 #define FSL\_FLEXIO\_I2S\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 7))**

### 14.4.7.4 Function Documentation

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

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

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

Note

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

Parameters

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

Return values

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

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

Note

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

Parameters

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

Return values

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

Return values

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

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

Parameters

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

Return values

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

## 14.5 FlexIO SPI Driver

### 14.5.1 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.

### 14.5.2 Typical use case

#### 14.5.2.1 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)
 {
 // ...
 }
}

```

#### 14.5.2.2 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** (0x00U)  
*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, 2))  
*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.*

### 14.5.3 Data Structure Documentation

#### 14.5.3.1 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]

#### 14.5.3.2 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

#### 14.5.3.3 struct flexio\_spi\_slave\_config\_t

##### Data Fields

- bool enableSlave

- **bool enableInDoze**  
*Enable/disable FlexIO SPI slave after configuration.*
- **bool enableInDebug**  
*Enable/disable FlexIO operation in doze mode.*
- **bool enableFastAccess**  
*Enable/disable FlexIO operation in debug mode.*
- **flexio\_spi\_clock\_phase\_t phase**  
*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\_data\_bitcount\_mode\_t dataMode**  
*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**

### 14.5.3.4 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`

### 14.5.3.5 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`

## 14.5.4 Macro Definition Documentation

**14.5.4.1 #define FSL\_FLEXIO\_SPI\_DRIVER\_VERSION (MAKE\_VERSION(2, 3, 2))**

**14.5.4.2 #define FLEXIO\_SPI\_DUMMYDATA (0x00U)**

**14.5.4.3 #define SPI\_RETRY\_TIMES 0U /\* Define to zero means keep waiting until the flag is assert/deassert. \*/**

**14.5.4.4 #define FLEXIO\_SPI\_XFER\_DATA\_FORMAT( *flag* ) ((*flag*) & (0x7U))**

## 14.5.5 Typedef Documentation

**14.5.5.1 typedef flexio\_spi\_master\_handle\_t flexio\_spi\_slave\_handle\_t**

## 14.5.6 Enumeration Type Documentation

### 14.5.6.1 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.

#### 14.5.6.2 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.

#### 14.5.6.3 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.

#### 14.5.6.4 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.

#### 14.5.6.5 enum \_flexio\_spi\_interrupt\_enable

Enumerator

***kFLEXIO\_SPI\_TxEmptyInterruptEnable*** Transmit buffer empty interrupt enable.

***kFLEXIO\_SPI\_RxFullInterruptEnable*** Receive buffer full interrupt enable.

#### 14.5.6.6 enum \_flexio\_spi\_status\_flags

Enumerator

***kFLEXIO\_SPI\_TxBufferEmptyFlag*** Transmit buffer empty flag.

***kFLEXIO\_SPI\_RxBufferFullFlag*** Receive buffer full flag.

#### 14.5.6.7 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.

#### 14.5.6.8 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.

### 14.5.7 Function Documentation

#### 14.5.7.1 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 <a href="#">FLEXIO_SPI_Type</a> structure.            |
| <i>masterConfig</i> | Pointer to the <a href="#">flexio_spi_master_config_t</a> structure. |
| <i>srcClock_Hz</i>  | FlexIO source clock in Hz.                                           |

#### 14.5.7.2 void FLEXIO\_SPI\_MasterDeinit ( [FLEXIO\\_SPI\\_Type](#) \* *base* )

#### Parameters

|             |                                                  |
|-------------|--------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> . |
|-------------|--------------------------------------------------|

#### 14.5.7.3 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. |
|---------------------|----------------------------------------------------------------------|

#### 14.5.7.4 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. |

**14.5.7.5 void FLEXIO\_SPI\_SlaveDeinit ( FLEXIO\_SPI\_Type \* *base* )**

## Parameters

|             |                                                  |
|-------------|--------------------------------------------------|
| <i>base</i> | Pointer to the <a href="#">FLEXIO_SPI_Type</a> . |
|-------------|--------------------------------------------------|

**14.5.7.6 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. |
|--------------------|---------------------------------------------------------------------|

#### 14.5.7.7 `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`

#### 14.5.7.8 `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>• <code>kFLEXIO_SPI_TxEmptyFlag</code></li><li>• <code>kFLEXIO_SPI_RxEmptyFlag</code></li></ul> |

#### 14.5.7.9 `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> |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

#### 14.5.7.10 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> |

#### 14.5.7.11 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.           |

#### 14.5.7.12 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.

#### **14.5.7.13 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.

#### **14.5.7.14 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.  |

#### **14.5.7.15 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.                         |

#### **14.5.7.16 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.                                         |

**14.5.7.17 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.

**14.5.7.18 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. |

#### 14.5.7.19 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. |

#### 14.5.7.20 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. |

#### 14.5.7.21 void FLEXIO\_SPI\_FlushShifters ( [FLEXIO\\_SPI\\_Type](#) \* *base* )

Parameters

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

#### 14.5.7.22 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. |

#### 14.5.7.23 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 <code>flexio_spi_master_handle_t</code> structure to store the transfer state. |
| <i>xfer</i>   | FlexIO SPI transfer structure. See <a href="#">flexio_spi_transfer_t</a> .                    |

Return values

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

#### 14.5.7.24 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 <code>flexio_spi_master_handle_t</code> structure to store the transfer state. |

#### 14.5.7.25 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 <code>flexio_spi_master_handle_t</code> structure to store the transfer state. |
| <i>count</i>  | Number of bytes transferred so far by the non-blocking transaction.                           |

Return values

|                                |                                |
|--------------------------------|--------------------------------|
| <i>kStatus_InvalidArgument</i> | count is Invalid.              |
| <i>kStatus_Success</i>         | Successfully return the count. |

#### 14.5.7.26 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 <a href="#">flexio_spi_master_handle_t</a> structure to store the transfer state. |

**14.5.7.27 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 <a href="#">flexio_spi_slave_handle_t</a> structure to store the transfer state. |
| <i>callback</i> | The callback function.                                                                          |
| <i>userData</i> | The parameter of the callback function.                                                         |

Return values

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

**14.5.7.28 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 <a href="#">flexio_spi_slave_handle_t</a> structure to store the transfer state. |
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_SPI_Type</a> structure.                                       |
| <i>xfer</i>   | FlexIO SPI transfer structure. See <a href="#">flexio_spi_transfer_t</a> .                      |

Return values

|                                |                                                  |
|--------------------------------|--------------------------------------------------|
| <i>kStatus_Success</i>         | Successfully start a transfer.                   |
| <i>kStatus_InvalidArgument</i> | Input argument is invalid.                       |
| <i>kStatus_FLEXIO_SPI_Busy</i> | SPI is not idle; it is running another transfer. |

14.5.7.29 **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. |

**14.5.7.30 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> | count is Invalid.              |
| <i>kStatus_Success</i>         | Successfully return the count. |

**14.5.7.31 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 flexio_spi_slave_handle_t structure to store the transfer state. |

## 14.5.8 FlexIO eDMA SPI Driver

### 14.5.8.1 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.`

#### 14.5.8.2 Data Structure Documentation

##### 14.5.8.2.1 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`

### 14.5.8.3 Macro Definition Documentation

**14.5.8.3.1 #define FSL\_FLEXIO\_SPI\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 3, 0))**

### 14.5.8.4 Typedef Documentation

**14.5.8.4.1 typedef flexio\_spi\_master\_edma\_handle\_t flexio\_spi\_slave\_edma\_handle\_t**

### 14.5.8.5 Function Documentation

**14.5.8.5.1 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. |

**14.5.8.5.2 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 <code>flexio_spi_master_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. |

#### 14.5.8.5.3 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.                       |

#### 14.5.8.5.4 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. |

```
14.5.8.5.5 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.                                   |

**14.5.8.5.6 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. |

**14.5.8.5.7 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. |

**14.5.8.5.8 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. |

## 14.6 FlexIO UART Driver

### 14.6.1 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.

### 14.6.2 Typical use case

#### 14.6.2.1 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);
}

```

#### 14.6.2.2 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)
{
}

// ...
}

```

#### 14.6.2.3 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)
 {
 }
 }
}

// ...
}

```

#### 14.6.2.4 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.*

### 14.6.3 Data Structure Documentation

#### 14.6.3.1 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]`

#### 14.6.3.2 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**

**14.6.3.3 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**

**14.6.3.4 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`

### 14.6.4 Macro Definition Documentation

14.6.4.1 `#define FSL_FLEXIO_UART_DRIVER_VERSION (MAKE_VERSION(2, 4, 0))`

14.6.4.2 `#define UART_RETRY_TIMES 0U /* Defining to zero means to keep waiting for the flag until it is assert/deassert. */`

### 14.6.5 Typedef Documentation

14.6.5.1 `typedef void(* flexio_uart_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_handle_t *handle, status_t status, void *userData)`

### 14.6.6 Enumeration Type Documentation

#### 14.6.6.1 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.

#### 14.6.6.2 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

#### 14.6.6.3 enum \_flexio\_uart\_interrupt\_enable

Enumerator

*kFLEXIO\_UART\_TxDataRegEmptyInterruptEnable* Transmit buffer empty interrupt enable.  
*kFLEXIO\_UART\_RxDataRegFullInterruptEnable* Receive buffer full interrupt enable.

#### 14.6.6.4 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.

### 14.6.7 Function Documentation

#### 14.6.7.1 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. |

#### 14.6.7.2 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 |
|-------------|-------------------------------------------------------|

#### 14.6.7.3 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. |
|-------------------|----------------------------------------------------------------|

#### 14.6.7.4 `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.

#### 14.6.7.5 `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>• <code>kFLEXIO_UART_TxDataRegEmptyFlag</code></li> <li>• <code>kFLEXIO_UART_RxEmptyFlag</code></li> <li>• <code>kFLEXIO_UART_RxOverRunFlag</code></li> </ul> |

#### 14.6.7.6 `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.                                          |

#### 14.6.7.7 `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.                                          |

#### 14.6.7.8 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.

#### 14.6.7.9 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.

#### 14.6.7.10 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.                          |

#### 14.6.7.11 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.                          |

#### 14.6.7.12 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.   |

#### 14.6.7.13 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. |
|---------------|-------------------------|

#### 14.6.7.14 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.                    |

#### 14.6.7.15 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.            |

#### 14.6.7.16 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.         |

#### 14.6.7.17 `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 <code>flexio_uart_handle_t</code> 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. |

#### 14.6.7.18 `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.                                                                     |

### **14.6.7.19 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. |

### **14.6.7.20 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. |

#### 14.6.7.21 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. |

#### 14.6.7.22 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. |
|------------------------|--------------------------------|

**14.6.7.23 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.                |

**14.6.7.24 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 <code>flexio_uart_handle_t</code> structure to store the transfer state. |

#### 14.6.7.25 `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 <code>flexio_uart_handle_t</code> structure to store the transfer state. |
| <i>count</i>  | Number of bytes received so far by the non-blocking transaction.                        |

Return values

|                                     |                                                   |
|-------------------------------------|---------------------------------------------------|
| <i>kStatus_NoTransferInProgress</i> | transfer has finished or no transfer in progress. |
| <i>kStatus_Success</i>              | Successfully return the count.                    |

#### 14.6.7.26 `void FLEXIO_UART_TransferHandleIRQ ( void * uartType, void * uartHandle )`

This function processes the FlexIO UART transmit and receives the IRQ request.

Parameters

|                   |                                                                                         |
|-------------------|-----------------------------------------------------------------------------------------|
| <i>uartType</i>   | Pointer to the <a href="#">FLEXIO_UART_Type</a> structure.                              |
| <i>uartHandle</i> | Pointer to the <code>flexio_uart_handle_t</code> structure to store the transfer state. |

## 14.6.8 FlexIO eDMA UART Driver

### 14.6.8.1 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.*

#### 14.6.8.2 Data Structure Documentation

##### 14.6.8.2.1 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`

### 14.6.8.3 Macro Definition Documentation

14.6.8.3.1 `#define FSL_FLEXIO_UART_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 4, 1))`

### 14.6.8.4 Typedef Documentation

14.6.8.4.1 `typedef void(* flexio_uart_edma_transfer_callback_t)(FLEXIO_UART_Type *base, flexio_uart_edma_handle_t *handle, status_t status, void *userData)`

### 14.6.8.5 Function Documentation

14.6.8.5.1 `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 <a href="#">flexio_uart_edma_handle_t</a> structure. |
| <i>callback</i>     | The callback function.                                          |
| <i>userData</i>     | The parameter of the callback function.                         |
| <i>rxEdmaHandle</i> | User requested DMA handle for RX DMA transfer.                  |
| <i>txEdmaHandle</i> | User requested DMA handle for TX DMA transfer.                  |

Return values

|                           |                                                     |
|---------------------------|-----------------------------------------------------|
| <i>kStatus_Success</i>    | Successfully create the handle.                     |
| <i>kStatus_OutOfRange</i> | The FlexIO SPI eDMA type/handle table out of range. |

#### **14.6.8.5.2 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. |

#### **14.6.8.5.3 status\_t FLEXIO\_UART\_TransferReceiveEDMA ( [FLEXIO\\_UART\\_Type](#) \* *base*, [flexio\\_uart\\_edma\\_handle\\_t](#) \* *handle*, [flexio\\_uart\\_transfer\\_t](#) \* *xfer* )**

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

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>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. |

#### **14.6.8.5.4 void FLEXIO\_UART\_TransferAbortSendEDMA ( `FLEXIO_UART_Type * base,` `flexio_uart_edma_handle_t * handle` )**

This function aborts sent data which using eDMA.

Parameters

|               |                                                             |
|---------------|-------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                 |
| <i>handle</i> | Pointer to <code>flexio_uart_edma_handle_t</code> structure |

#### **14.6.8.5.5 void FLEXIO\_UART\_TransferAbortReceiveEDMA ( `FLEXIO_UART_Type * base,` `flexio_uart_edma_handle_t * handle` )**

This function aborts the receive data which using eDMA.

Parameters

|               |                                                             |
|---------------|-------------------------------------------------------------|
| <i>base</i>   | Pointer to <a href="#">FLEXIO_UART_Type</a>                 |
| <i>handle</i> | Pointer to <code>flexio_uart_edma_handle_t</code> structure |

#### **14.6.8.5.6 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.                    |

#### **14.6.8.5.7 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.                    |

## 14.7 FlexIO Camera Driver

### 14.7.1 Overview

The MCUXpresso SDK provides a driver for the camera function using Flexible I/O.

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

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

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

### 14.7.2 Typical use case

#### 14.7.2.1 FlexIO Camera Receive using eDMA method

```

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

flexio_camera_config_t cameraConfig;

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

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

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

```

```

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

```

## Modules

- FlexIO eDMA Camera Driver

## Data Structures

- struct **FLEXIO\_CAMERA\_Type**  
*Define structure of configuring the FlexIO Camera device.* [More...](#)
- struct **flexio\_camera\_config\_t**  
*Define FlexIO Camera user configuration structure.* [More...](#)
- struct **flexio\_camera\_transfer\_t**  
*Define FlexIO Camera transfer structure.* [More...](#)

## Macros

- #define **FLEXIO\_CAMERA\_PARALLEL\_DATA\_WIDTH** (8U)  
*Define the Camera CPI interface is constantly 8-bit width.*

## Enumerations

- enum {
 kStatus\_FLEXIO\_CAMERA\_RxBusy = MAKE\_STATUS(kStatusGroup\_FLEXIO\_CAMERA,

```

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

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

```

*Define FlexIO Camera status mask.*

## Driver version

- #define FSL\_FLEXIO\_CAMERA\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 3))
 *FlexIO Camera driver version 2.1.3.*

## Initialization and configuration

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

## Status

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

## Interrupts

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

## DMA support

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

*Enables/disables the FlexIO Camera receive DMA.*

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

### 14.7.3 Data Structure Documentation

#### 14.7.3.1 struct FLEXIO\_CAMERA\_Type

##### Data Fields

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

##### Field Documentation

(1) **FLEXIO\_Type\* FLEXIO\_CAMERA\_Type::flexioBase**

(2) **uint32\_t FLEXIO\_CAMERA\_Type::datPinStartIdx**

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

- (3) **uint32\_t FLEXIO\_CAMERA\_Type::pclkPinIdx**
- (4) **uint32\_t FLEXIO\_CAMERA\_Type::hrefPinIdx**
- (5) **uint32\_t FLEXIO\_CAMERA\_Type::shifterStartIdx**
- (6) **uint32\_t FLEXIO\_CAMERA\_Type::shifterCount**
- (7) **uint32\_t FLEXIO\_CAMERA\_Type::timerIdx**

#### 14.7.3.2 struct flexio\_camera\_config\_t

##### Data Fields

- bool **enablecamera**  
*Enable/disable FlexIO Camera 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.*

### Field Documentation

- (1) `bool flexio_camera_config_t::enablecamera`
- (2) `bool flexio_camera_config_t::enableFastAccess`

### 14.7.3.3 struct flexio\_camera\_transfer\_t

#### Data Fields

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

### 14.7.4 Macro Definition Documentation

#### 14.7.4.1 #define FSL\_FLEXIO\_CAMERA\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 3))

#### 14.7.4.2 #define FLEXIO\_CAMERA\_PARALLEL\_DATA\_WIDTH (8U)

### 14.7.5 Enumeration Type Documentation

#### 14.7.5.1 anonymous enum

Enumerator

- `kStatus_FLEXIO_CAMERA_RxBusy` Receiver is busy.  
`kStatus_FLEXIO_CAMERA_RxIdle` Camera receiver is idle.

#### 14.7.5.2 enum \_flexio\_camera\_status\_flags

Enumerator

- `kFLEXIO_CAMERA_RxDataRegFullFlag` Receive buffer full flag.  
`kFLEXIO_CAMERA_RxErrorFlag` Receive buffer error flag.

## 14.7.6 Function Documentation

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

Parameters

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

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

Note

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

Parameters

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

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

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

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

Parameters

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

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

Parameters

|               |                                                   |
|---------------|---------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> |
| <i>enable</i> | True to enable, false does not have any effect.   |

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

Parameters

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

Returns

FlexIO shifter status flags

- `FLEXIO_SHIFTSTAT_SSF_MASK`
- 0

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

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

Parameters

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

Returns

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

## 14.7.7 FlexIO eDMA Camera Driver

### 14.7.7.1 Overview

#### Data Structures

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

#### TypeDefs

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

#### Driver version

- #define `FSL_FLEXIO_CAMERA_EDMA_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 3)`)  
*FlexIO Camera EDMA driver version 2.1.3.*

#### eDMA transactional

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

### 14.7.7.2 Data Structure Documentation

#### 14.7.7.2.1 struct \_flexio\_camera\_edma\_handle

Forward declaration of the handle typedef.

#### Data Fields

- `flexio_camera_edma_transfer_callback_t` callback

- *Callback function.*
- `void *userData`  
*Camera callback function parameter.*
- `size_t rxSize`  
*Total bytes to be received.*
- `edma_handle_t *rxEdmaHandle`  
*The eDMA RX channel used.*
- `uint8_t nbytes`  
*eDMA minor byte transfer count initially configured.*
- `volatile uint8_t rxState`  
*RX transfer state.*

## Field Documentation

- (1) `flexio_camera_edma_transfer_callback_t flexio_camera_edma_handle_t::callback`
- (2) `void* flexio_camera_edma_handle_t::userData`
- (3) `size_t flexio_camera_edma_handle_t::rxSize`
- (4) `edma_handle_t* flexio_camera_edma_handle_t::rxEdmaHandle`
- (5) `uint8_t flexio_camera_edma_handle_t::nbytes`

### 14.7.7.3 Macro Definition Documentation

14.7.7.3.1 `#define FSL_FLEXIO_CAMERA_EDMA_DRIVER_VERSION(MAKE_VERSION(2, 1, 3))`

### 14.7.7.4 Typedef Documentation

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

### 14.7.7.5 Function Documentation

14.7.7.5.1 `status_t FLEXIO_CAMERA_TransferCreateHandleEDMA ( FLEXIO_CAMERA_Type  
* base, flexio_camera_edma_handle_t * handle, flexio_camera_edma_transfer-  
_callback_t callback, void * userData, edma_handle_t * rxEdmaHandle  
)`

#### Parameters

|                   |                                                     |
|-------------------|-----------------------------------------------------|
| <code>base</code> | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> . |
|-------------------|-----------------------------------------------------|

|                     |                                                   |
|---------------------|---------------------------------------------------|
| <i>handle</i>       | Pointer to flexio_camera_edma_handle_t 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.    |

Return values

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

#### 14.7.7.5.2 status\_t FLEXIO\_CAMERA\_TransferReceiveEDMA ( FLEXIO\_CAMERA\_Type \* *base*, flexio\_camera\_edma\_handle\_t \* *handle*, flexio\_camera\_transfer\_t \* *xfer* )

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

Parameters

|               |                                                                                |
|---------------|--------------------------------------------------------------------------------|
| <i>base</i>   | Pointer to the <a href="#">FLEXIO_CAMERA_Type</a> .                            |
| <i>handle</i> | Pointer to the flexio_camera_edma_handle_t structure.                          |
| <i>xfer</i>   | Camera eDMA transfer structure, see <a href="#">flexio_camera_transfer_t</a> . |

Return values

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

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

This function aborts the receive data which used the eDMA.

Parameters

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

|               |                                                       |
|---------------|-------------------------------------------------------|
| <i>handle</i> | Pointer to the flexio_camera_edma_handle_t structure. |
|---------------|-------------------------------------------------------|

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

This function gets the number of bytes still not received.

Parameters

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

Return values

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

# 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, 7, 2)`)  
*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, 7, 2))`

## 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> | GPIOpin output logic level. <ul style="list-style-type: none"><li>• 0: corresponding pin output low-logic level.</li><li>• 1: corresponding pin output high-logic level.</li></ul> |

### 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

## LLWU: Low-Leakage Wakeup Unit Driver

### 16.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Low-Leakage Wakeup Unit (LLWU) module of MCUXpresso SDK devices. The LLWU module allows the user to select external pin sources and internal modules as a wake-up source from low-leakage power modes.

### 16.2 External wakeup pins configurations

Configures the external wakeup pins' working modes, gets, and clears the wake pin flags. External wakeup pins are accessed by the `pinIndex`, which is started from 1. Numbers of the external pins depend on the SoC configuration.

### 16.3 Internal wakeup modules configurations

Enables/disables the internal wakeup modules and gets the module flags. Internal modules are accessed by `moduleIndex`, which is started from 1. Numbers of external pins depend the on SoC configuration.

### 16.4 Digital pin filter for external wakeup pin configurations

Configures the digital pin filter of the external wakeup pins' working modes, gets, and clears the pin filter flags. Digital pin filters are accessed by the `filterIndex`, which is started from 1. Numbers of external pins depend on the SoC configuration.

## Enumerations

- enum `llwu_external_pin_mode_t` {  
    `kLLWU_ExternalPinDisable` = 0U,  
    `kLLWU_ExternalPinRisingEdge` = 1U,  
    `kLLWU_ExternalPinFallingEdge` = 2U,  
    `kLLWU_ExternalPinAnyEdge` = 3U }  
    *External input pin control modes.*
- enum `llwu_pin_filter_mode_t` {  
    `kLLWU_PinFilterDisable` = 0U,  
    `kLLWU_PinFilterRisingEdge` = 1U,  
    `kLLWU_PinFilterFallingEdge` = 2U,  
    `kLLWU_PinFilterAnyEdge` = 3U }  
    *Digital filter control modes.*

## Driver version

- #define `FSL_LLWU_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 5)`)  
*LLWU driver version.*

## 16.5 Macro Definition Documentation

16.5.1 `#define FSL_LLWU_DRIVER_VERSION (MAKE_VERSION(2, 0, 5))`

## 16.6 Enumeration Type Documentation

16.6.1 `enum llwu_external_pin_mode_t`

Enumerator

*kLLWU\_ExternalPinDisable* Pin disabled as a wakeup input.

*kLLWU\_ExternalPinRisingEdge* Pin enabled with the rising edge detection.

*kLLWU\_ExternalPinFallingEdge* Pin enabled with the falling edge detection.

*kLLWU\_ExternalPinAnyEdge* Pin enabled with any change detection.

16.6.2 `enum llwu_pin_filter_mode_t`

Enumerator

*kLLWU\_PinFilterDisable* Filter disabled.

*kLLWU\_PinFilterRisingEdge* Filter positive edge detection.

*kLLWU\_PinFilterFallingEdge* Filter negative edge detection.

*kLLWU\_PinFilterAnyEdge* Filter any edge detection.

# Chapter 17

## LPADC: 12-bit SAR Analog-to-Digital Converter Driver

### 17.1 Overview

The MCUXpresso SDK provides a peripheral driver for the 12-bit SAR Analog-to-Digital Converter (LP-ADC) module of MCUXpresso SDK devices.

### 17.2 Typical use case

#### 17.2.1 Polling Configuration

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/lpadc

#### 17.2.2 Interrupt Configuration

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/lpadc

### Files

- file [fsl\\_lpadc.h](#)

### Data Structures

- struct [lpadc\\_config\\_t](#)  
*LPADC global configuration. [More...](#)*
- struct [lpadc\\_conv\\_command\\_config\\_t](#)  
*Define structure to keep the configuration for conversion command. [More...](#)*
- struct [lpadc\\_conv\\_trigger\\_config\\_t](#)  
*Define structure to keep the configuration for conversion trigger. [More...](#)*
- struct [lpadc\\_conv\\_result\\_t](#)  
*Define the structure to keep the conversion result. [More...](#)*

### Macros

- #define [LPADC\\_GET\\_ACTIVE\\_COMMAND\\_STATUS](#)(statusVal) ((statusVal & ADC\_STAT\_CMDACT\_MASK) >> ADC\_STAT\_CMDACT\_SHIFT)  
*Define the MACRO function to get command status from status value.*
- #define [LPADC\\_GET\\_ACTIVE\\_TRIGGER\\_STATUS](#)(statusVal) ((statusVal & ADC\_STAT\_TRGACT\_MASK) >> ADC\_STAT\_TRGACT\_SHIFT)  
*Define the MACRO function to get trigger status from status value.*

## Enumerations

- enum `_lpadc_status_flags` {
   
  `kLPADC_ResultFIFOOverflowFlag` = ADC\_STAT\_FOF\_MASK,
   
  `kLPADC_ResultFIFOReadyFlag` = ADC\_STAT\_RDY\_MASK }
   
    *Define hardware flags of the module.*
- enum `_lpadc_interrupt_enable` {
   
  `kLPADC_ResultFIFOOverflowInterruptEnable` = ADC\_IE\_FOFIE\_MASK,
   
  `kLPADC_FIFOWatermarkInterruptEnable` = ADC\_IE\_FWMIE\_MASK }
   
    *Define interrupt switchers of the module.*
- enum `lpadc_sample_scale_mode_t` {
   
  `kLPADC_SamplePartScale`,
   
  `kLPADC_SampleFullScale` = 1U }
   
    *Define enumeration of sample scale mode.*
- enum `lpadc_sample_channel_mode_t` {
   
  `kLPADC_SampleChannelSingleEndSideA` = 0U,
   
  `kLPADC_SampleChannelSingleEndSideB` = 1U }
   
    *Define enumeration of channel sample mode.*
- enum `lpadc_hardware_average_mode_t` {
   
  `kLPADC_HardwareAverageCount1` = 0U,
   
  `kLPADC_HardwareAverageCount2` = 1U,
   
  `kLPADC_HardwareAverageCount4` = 2U,
   
  `kLPADC_HardwareAverageCount8` = 3U,
   
  `kLPADC_HardwareAverageCount16` = 4U,
   
  `kLPADC_HardwareAverageCount32` = 5U,
   
  `kLPADC_HardwareAverageCount64` = 6U,
   
  `kLPADC_HardwareAverageCount128` = 7U }
   
    *Define enumeration of hardware average selection.*
- enum `lpadc_sample_time_mode_t` {
   
  `kLPADC_SampleTimeADCK3` = 0U,
   
  `kLPADC_SampleTimeADCK5` = 1U,
   
  `kLPADC_SampleTimeADCK7` = 2U,
   
  `kLPADC_SampleTimeADCK11` = 3U,
   
  `kLPADC_SampleTimeADCK19` = 4U,
   
  `kLPADC_SampleTimeADCK35` = 5U,
   
  `kLPADC_SampleTimeADCK67` = 6U,
   
  `kLPADC_SampleTimeADCK131` = 7U }
   
    *Define enumeration of sample time selection.*
- enum `lpadc_hardware_compare_mode_t` {
   
  `kLPADC_HardwareCompareDisabled` = 0U,
   
  `kLPADC_HardwareCompareStoreOnTrue` = 2U,
   
  `kLPADC_HardwareCompareRepeatUntilTrue` = 3U }
   
    *Define enumeration of hardware compare mode.*
- enum `lpadc_reference_voltage_source_t` {
   
  `kLPADC_ReferenceVoltageAlt1` = 0U,
   
  `kLPADC_ReferenceVoltageAlt2` = 1U,
   
  `kLPADC_ReferenceVoltageAlt3` = 2U }

- Define enumeration of reference voltage source.
- enum `lpadc_power_level_mode_t` {
   
    kLPADC\_PowerLevelAlt1 = 0U,  
 kLPADC\_PowerLevelAlt2 = 1U,  
 kLPADC\_PowerLevelAlt3 = 2U,  
 kLPADC\_PowerLevelAlt4 = 3U }
- Define enumeration of power configuration.
- enum `lpadc_trigger_priority_policy_t` {
   
    kLPADC\_TriggerPriorityPreemptImmediately = 0U,  
 kLPADC\_TriggerPriorityPreemptSoftly = 1U }
- Define enumeration of trigger priority policy.

## Driver version

- #define `FSL_LPADC_DRIVER_VERSION` (`MAKE_VERSION(2, 6, 2)`)  
*LPADC driver version 2.6.2.*

## Initialization & de-initialization.

- void `LPADC_Init` (ADC\_Type \*base, const `lpadc_config_t` \*config)  
*Initializes the LPADC module.*
- void `LPADC_GetDefaultConfig` (`lpadc_config_t` \*config)  
*Gets an available pre-defined settings for initial configuration.*
- void `LPADC_Deinit` (ADC\_Type \*base)  
*De-initializes the LPADC module.*
- static void `LPADC_Enable` (ADC\_Type \*base, bool enable)  
*Switch on/off the LPADC module.*
- static void `LPADC_DoResetFIFO` (ADC\_Type \*base)  
*Do reset the conversion FIFO.*
- static void `LPADC_DoResetConfig` (ADC\_Type \*base)  
*Do reset the module's configuration.*

## Status

- static uint32\_t `LPADC_GetStatusFlags` (ADC\_Type \*base)  
*Get status flags.*
- static void `LPADC_ClearStatusFlags` (ADC\_Type \*base, uint32\_t mask)  
*Clear status flags.*

## Interrupts

- static void `LPADC_EnableInterrupts` (ADC\_Type \*base, uint32\_t mask)  
*Enable interrupts.*
- static void `LPADC_DisableInterrupts` (ADC\_Type \*base, uint32\_t mask)  
*Disable interrupts.*

## DMA Control

- static void `LPADC_EnableFIFOWatermarkDMA` (ADC\_Type \*base, bool enable)  
*Switch on/off the DMA trigger for FIFO watermark event.*

## Trigger and conversion with FIFO.

- static uint32\_t [LPADC\\_GetConvResultCount](#) (ADC\_Type \*base)  
*Get the count of result kept in conversion FIFO.*
- bool [LPADC\\_GetConvResult](#) (ADC\_Type \*base, [lpadc\\_conv\\_result\\_t](#) \*result)  
*Get the result in conversion FIFO.*
- void [LPADC\\_SetConvTriggerConfig](#) (ADC\_Type \*base, uint32\_t triggerId, const [lpadc\\_conv\\_trigger\\_config\\_t](#) \*config)  
*Configure the conversion trigger source.*
- void [LPADC\\_GetDefaultConvTriggerConfig](#) ([lpadc\\_conv\\_trigger\\_config\\_t](#) \*config)  
*Gets an available pre-defined settings for trigger's configuration.*
- static void [LPADC\\_DoSoftwareTrigger](#) (ADC\_Type \*base, uint32\_t triggerIdMask)  
*Do software trigger to conversion command.*
- void [LPADC\\_SetConvCommandConfig](#) (ADC\_Type \*base, uint32\_t commandId, const [lpadc\\_conv\\_command\\_config\\_t](#) \*config)  
*Configure conversion command.*
- void [LPADC\\_GetDefaultConvCommandConfig](#) ([lpadc\\_conv\\_command\\_config\\_t](#) \*config)  
*Gets an available pre-defined settings for conversion command's configuration.*

## 17.3 Data Structure Documentation

### 17.3.1 struct [lpadc\\_config\\_t](#)

This structure would used to keep the settings for initialization.

#### Data Fields

- bool [enableInDozeMode](#)  
*Control system transition to Stop and Wait power modes while ADC is converting.*
- bool [enableAnalogPreliminary](#)  
*ADC analog circuits are pre-enabled and ready to execute conversions without startup delays(at the cost of higher DC current consumption).*
- uint32\_t [powerUpDelay](#)  
*When the analog circuits are not pre-enabled, the ADC analog circuits are only powered while the ADC is active and there is a counted delay defined by this field after an initial trigger transitions the ADC from its Idle state to allow time for the analog circuits to stabilize.*
- [lpadc\\_reference\\_voltage\\_source\\_t](#) [referenceVoltageSource](#)  
*Selects the voltage reference high used for conversions.*
- [lpadc\\_power\\_level\\_mode\\_t](#) [powerLevelMode](#)  
*Power Configuration Selection.*
- [lpadc\\_trigger\\_priority\\_policy\\_t](#) [triggerPriorityPolicy](#)  
*Control how higher priority triggers are handled, see to [lpadc\\_trigger\\_priority\\_policy\\_t](#).*
- bool [enableConvPause](#)  
*Enables the ADC pausing function.*
- uint32\_t [convPauseDelay](#)  
*Controls the duration of pausing during command execution sequencing.*
- uint32\_t [FIFOWatermark](#)  
*FIFOWatermark is a programmable threshold setting.*

**Field Documentation****(1) bool lpadc\_config\_t::enableInDozeMode**

When enabled in Doze mode, immediate entries to Wait or Stop are allowed. When disabled, the ADC will wait for the current averaging iteration/FIFO storage to complete before acknowledging stop or wait mode entry.

**(2) bool lpadc\_config\_t::enableAnalogPreliminary****(3) uint32\_t lpadc\_config\_t::powerUpDelay**

The startup delay count of (powerUpDelay \* 4) ADCK cycles must result in a longer delay than the analog startup time.

**(4) lpadc\_reference\_voltage\_source\_t lpadc\_config\_t::referenceVoltageSource****(5) lpadc\_power\_level\_mode\_t lpadc\_config\_t::powerLevelMode****(6) lpadc\_trigger\_priority\_policy\_t lpadc\_config\_t::triggerPriorityPolicy****(7) bool lpadc\_config\_t::enableConvPause**

When enabled, a programmable delay is inserted during command execution sequencing between LOOP iterations, between commands in a sequence, and between conversions when command is executing in "Compare Until True" configuration.

**(8) uint32\_t lpadc\_config\_t::convPauseDelay**

The pause delay is a count of (convPauseDelay\*4) ADCK cycles. Only available when ADC pausing function is enabled. The available value range is in 9-bit.

**(9) uint32\_t lpadc\_config\_t::FIFOWatermark**

When the number of datawords stored in the ADC Result FIFO is greater than the value in this field, the ready flag would be asserted to indicate stored data has reached the programmable threshold.

**17.3.2 struct lpadc\_conv\_command\_config\_t****Data Fields**

- [lpadc\\_sample\\_channel\\_mode\\_t sampleChannelMode](#)  
*Channel sample mode.*
- [uint32\\_t channelNumber](#)  
*Channel number, select the channel or channel pair.*
- [uint32\\_t chainedNextCommandNumber](#)  
*Selects the next command to be executed after this command completes.*
- [bool enableAutoChannelIncrement](#)

*Loop with increment: when disabled, the "loopCount" field selects the number of times the selected channel is converted consecutively; when enabled, the "loopCount" field defines how many consecutive channels are converted as part of the command execution.*

- `uint32_t loopCount`  
*Selects how many times this command executes before finish and transition to the next command or Idle state.*
- `lpadc_hardware_average_mode_t hardwareAverageMode`  
*Hardware average selection.*
- `lpadc_sample_time_mode_t sampleTimeMode`  
*Sample time selection.*
- `lpadc_hardware_compare_mode_t hardwareCompareMode`  
*Hardware compare selection.*
- `uint32_t hardwareCompareValueHigh`  
*Compare Value High.*
- `uint32_t hardwareCompareValueLow`  
*Compare Value Low.*

## Field Documentation

(1) `lpadc_sample_channel_mode_t lpadc_conv_command_config_t::sampleChannelMode`

(2) `uint32_t lpadc_conv_command_config_t::channelNumber`

(3) `uint32_t lpadc_conv_command_config_t::chainedNextCommandNumber`

1-15 is available, 0 is to terminate the chain after this command.

(4) `bool lpadc_conv_command_config_t::enableAutoChannelIncrement`

(5) `uint32_t lpadc_conv_command_config_t::loopCount`

Command executes LOOP+1 times. 0-15 is available.

(6) `lpadc_hardware_average_mode_t lpadc_conv_command_config_t::hardwareAverageMode`

(7) `lpadc_sample_time_mode_t lpadc_conv_command_config_t::sampleTimeMode`

(8) `lpadc_hardware_compare_mode_t lpadc_conv_command_config_t::hardwareCompareMode`

(9) `uint32_t lpadc_conv_command_config_t::hardwareCompareValueHigh`

The available value range is in 16-bit.

(10) `uint32_t lpadc_conv_command_config_t::hardwareCompareValueLow`

The available value range is in 16-bit.

### 17.3.3 struct Ipadc\_conv\_trigger\_config\_t

#### Data Fields

- `uint32_t targetCommandId`  
*Select the command from command buffer to execute upon detect of the associated trigger event.*
- `uint32_t delayPower`  
*Select the trigger delay duration to wait at the start of servicing a trigger event.*
- `uint32_t priority`  
*Sets the priority of the associated trigger source.*
- `bool enableHardwareTrigger`  
*Enable hardware trigger source to initiate conversion on the rising edge of the input trigger source or not.*

#### Field Documentation

(1) `uint32_t Ipadc_conv_trigger_config_t::targetCommandId`

(2) `uint32_t Ipadc_conv_trigger_config_t::delayPower`

When this field is clear, then no delay is incurred. When this field is set to a non-zero value, the duration for the delay is  $2^{\text{delayPower}}$  ADCK cycles. The available value range is 4-bit.

(3) `uint32_t Ipadc_conv_trigger_config_t::priority`

If two or more triggers have the same priority level setting, the lower order trigger event has the higher priority. The lower value for this field is for the higher priority, the available value range is 1-bit.

(4) `bool Ipadc_conv_trigger_config_t::enableHardwareTrigger`

The software trigger is always available.

### 17.3.4 struct Ipadc\_conv\_result\_t

#### Data Fields

- `uint32_t commandIdSource`  
*Indicate the command buffer being executed that generated this result.*
- `uint32_t loopCountIndex`  
*Indicate the loop count value during command execution that generated this result.*
- `uint32_t triggerIdSource`  
*Indicate the trigger source that initiated a conversion and generated this result.*
- `uint16_t convValue`  
*Data result.*

**Field Documentation**

- (1) `uint32_t lpadc_conv_result_t::commandIdSource`
- (2) `uint32_t lpadc_conv_result_t::loopCountIndex`
- (3) `uint32_t lpadc_conv_result_t::triggerIdSource`
- (4) `uint16_t lpadc_conv_result_t::convValue`

**17.4 Macro Definition Documentation**

**17.4.1 #define FSL\_LPADC\_DRIVER\_VERSION (MAKE\_VERSION(2, 6, 2))**

**17.4.2 #define LPADC\_GET\_ACTIVE\_COMMAND\_STATUS( *statusVal* ) ((*statusVal* & ADC\_STAT\_CMDACT\_MASK) >> ADC\_STAT\_CMDACT\_SHIFT)**

The *statusVal* is the return value from [LPADC\\_GetStatusFlags\(\)](#).

**17.4.3 #define LPADC\_GET\_ACTIVE\_TRIGGER\_STATUS( *statusVal* ) ((*statusVal* & ADC\_STAT\_TRGACT\_MASK) >> ADC\_STAT\_TRGACT\_SHIFT)**

The *statusVal* is the return value from [LPADC\\_GetStatusFlags\(\)](#).

**17.5 Enumeration Type Documentation****17.5.1 enum \_lpadc\_status\_flags**

Enumerator

***kLPADC\_ResultFIFOOverflowFlag*** Indicates that more data has been written to the Result FIFO than it can hold.

***kLPADC\_ResultFIFOReadyFlag*** Indicates when the number of valid datawords in the result FIFO is greater than the setting watermark level.

**17.5.2 enum \_lpadc\_interrupt\_enable**

Enumerator

***kLPADC\_ResultFIFOOverflowInterruptEnable*** Configures ADC to generate overflow interrupt requests when FOF flag is asserted.

***kLPADC\_FIFOWatermarkInterruptEnable*** Configures ADC to generate watermark interrupt requests when RDY flag is asserted.

### 17.5.3 enum lpadc\_sample\_scale\_mode\_t

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results in a voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

Enumerator

- kLPADC\_SamplePartScale*** Use divided input voltage signal. (For scale select, please refer to the reference manual).
- kLPADC\_SampleFullScale*** Full scale (Factor of 1).

### 17.5.4 enum lpadc\_sample\_channel\_mode\_t

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

Enumerator

- kLPADC\_SampleChannelSingleEndSideA*** Single end mode, using side A.
- kLPADC\_SampleChannelSingleEndSideB*** Single end mode, using side B.

### 17.5.5 enum lpadc\_hardware\_average\_mode\_t

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Enumerator

- kLPADC\_HardwareAverageCount1*** Single conversion.
- kLPADC\_HardwareAverageCount2*** 2 conversions averaged.
- kLPADC\_HardwareAverageCount4*** 4 conversions averaged.
- kLPADC\_HardwareAverageCount8*** 8 conversions averaged.
- kLPADC\_HardwareAverageCount16*** 16 conversions averaged.
- kLPADC\_HardwareAverageCount32*** 32 conversions averaged.
- kLPADC\_HardwareAverageCount64*** 64 conversions averaged.
- kLPADC\_HardwareAverageCount128*** 128 conversions averaged.

### 17.5.6 enum lpadc\_sample\_time\_mode\_t

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower

overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

Enumerator

- kLPADC\_SampleTimeADCK3* 3 ADCK cycles total sample time.
- kLPADC\_SampleTimeADCK5* 5 ADCK cycles total sample time.
- kLPADC\_SampleTimeADCK7* 7 ADCK cycles total sample time.
- kLPADC\_SampleTimeADCK11* 11 ADCK cycles total sample time.
- kLPADC\_SampleTimeADCK19* 19 ADCK cycles total sample time.
- kLPADC\_SampleTimeADCK35* 35 ADCK cycles total sample time.
- kLPADC\_SampleTimeADCK67* 69 ADCK cycles total sample time.
- kLPADC\_SampleTimeADCK131* 131 ADCK cycles total sample time.

### 17.5.7 enum lpadc\_hardware\_compare\_mode\_t

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

Enumerator

- kLPADC\_HardwareCompareDisabled* Compare disabled.
- kLPADC\_HardwareCompareStoreOnTrue* Compare enabled. Store on true.
- kLPADC\_HardwareCompareRepeatUntilTrue* Compare enabled. Repeat channel acquisition until true.

### 17.5.8 enum lpadc\_reference\_voltage\_source\_t

For detail information, need to check the SoC's specification.

Enumerator

- kLPADC\_ReferenceVoltageAlt1* Option 1 setting.
- kLPADC\_ReferenceVoltageAlt2* Option 2 setting.
- kLPADC\_ReferenceVoltageAlt3* Option 3 setting.

### 17.5.9 enum lpadc\_power\_level\_mode\_t

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

Enumerator

- kLPADC\_PowerLevelAlt1*** Lowest power setting.
- kLPADC\_PowerLevelAlt2*** Next lowest power setting.
- kLPADC\_PowerLevelAlt3*** ...
- kLPADC\_PowerLevelAlt4*** Highest power setting.

### 17.5.10 enum lpadc\_trigger\_priority\_policy\_t

This selection controls how higher priority triggers are handled.

Enumerator

- kLPADC\_TriggerPriorityPreemptImmediately*** If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started.
- kLPADC\_TriggerPriorityPreemptSoftly*** If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated.

## 17.6 Function Documentation

### 17.6.1 void LPADC\_Init ( ADC\_Type \* *base*, const lpadc\_config\_t \* *config* )

Parameters

|               |                                                           |
|---------------|-----------------------------------------------------------|
| <i>base</i>   | LPADC peripheral base address.                            |
| <i>config</i> | Pointer to configuration structure. See "lpadc_config_t". |

### 17.6.2 void LPADC\_GetDefaultConfig ( lpadc\_config\_t \* *config* )

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

```
* config->enableInDozeMode = true;
* config->enableAnalogPreliminary = false;
* config->powerUpDelay = 0x80;
* config->referenceVoltageSource = kLPADC_ReferenceVoltageAlt1;
* config->powerLevelMode = kLPADC_PowerLevelAlt1;
* config->triggerPriorityPolicy = kLPADC_TriggerPriorityPreemptImmediately
;
* config->enableConvPause = false;
* config->convPauseDelay = 0U;
* config->FIFOWatermark = 0U;
*
```

Parameters

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

### 17.6.3 void LPADC\_Deinit ( ADC\_Type \* *base* )

Parameters

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

### 17.6.4 static void LPADC\_Enable ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

|               |                                |
|---------------|--------------------------------|
| <i>base</i>   | LPADC peripheral base address. |
| <i>enable</i> | switcher to the module.        |

### 17.6.5 static void LPADC\_DoResetFIFO ( ADC\_Type \* *base* ) [inline], [static]

Parameters

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

### 17.6.6 static void LPADC\_DoResetConfig ( ADC\_Type \* *base* ) [inline], [static]

Reset all ADC internal logic and registers, except the Control Register (ADCx\_CTRL).

Parameters

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

### 17.6.7 static uint32\_t LPADC\_GetStatusFlags ( ADC\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

status flags' mask. See to [\\_lpadc\\_status\\_flags](#).

### 17.6.8 static void LPADC\_ClearStatusFlags ( ADC\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Only the flags can be cleared by writing ADCx\_STATUS register would be cleared by this API.

Parameters

|             |                                                                                  |
|-------------|----------------------------------------------------------------------------------|
| <i>base</i> | LPADC peripheral base address.                                                   |
| <i>mask</i> | Mask value for flags to be cleared. See to <a href="#">_lpadc_status_flags</a> . |

### 17.6.9 static void LPADC\_EnableInterrupts ( ADC\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                   |
|-------------|-----------------------------------------------------------------------------------|
| <i>base</i> | LPADC peripheral base address.                                                    |
| <i>mask</i> | Mask value for interrupt events. See to <a href="#">_lpadc_interrupt_enable</a> . |

### 17.6.10 static void LPADC\_DisableInterrupts ( ADC\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                   |
|-------------|-----------------------------------------------------------------------------------|
| <i>base</i> | LPADC peripheral base address.                                                    |
| <i>mask</i> | Mask value for interrupt events. See to <a href="#">_lpadc_interrupt_enable</a> . |

### 17.6.11 static void LPADC\_EnableFIFOWatermarkDMA ( ADC\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

|               |                                |
|---------------|--------------------------------|
| <i>base</i>   | LPADC peripheral base address. |
| <i>enable</i> | Switcher to the event.         |

### 17.6.12 static uint32\_t LPADC\_GetConvResultCount ( ADC\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

The count of result kept in conversion FIFO.

### 17.6.13 bool LPADC\_GetConvResult ( ADC\_Type \* *base*, lpadc\_conv\_result\_t \* *result* )

Parameters

|               |                                                                                    |
|---------------|------------------------------------------------------------------------------------|
| <i>base</i>   | LPADC peripheral base address.                                                     |
| <i>result</i> | Pointer to structure variable that keeps the conversion result in conversion FIFO. |

Returns

Status whether FIFO entry is valid.

**17.6.14 void LPADC\_SetConvTriggerConfig ( ADC\_Type \* *base*, uint32\_t *triggerId*, const lpadc\_conv\_trigger\_config\_t \* *config* )**

Each programmable trigger can launch the conversion command in command buffer.

## Parameters

|                  |                                                                                          |
|------------------|------------------------------------------------------------------------------------------|
| <i>base</i>      | LPADC peripheral base address.                                                           |
| <i>triggerId</i> | ID for each trigger. Typically, the available value range is from 0.                     |
| <i>config</i>    | Pointer to configuration structure. See to <a href="#">lpadc_conv_trigger_config_t</a> . |

**17.6.15 void LPADC\_GetDefaultConvTriggerConfig ( *lpadc\_conv\_trigger\_config\_t \* config* )**

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

```
* config->commandIdSource = 0U;
* config->loopCountIndex = 0U;
* config->triggerIdSource = 0U;
* config->enableHardwareTrigger = false;
*
```

## Parameters

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

**17.6.16 static void LPADC\_DoSoftwareTrigger ( *ADC\_Type \* base, uint32\_t triggerIdMask* ) [inline], [static]**

## Parameters

|                      |                                                                 |
|----------------------|-----------------------------------------------------------------|
| <i>base</i>          | LPADC peripheral base address.                                  |
| <i>triggerIdMask</i> | Mask value for software trigger indexes, which count from zero. |

**17.6.17 void LPADC\_SetConvCommandConfig ( *ADC\_Type \* base, uint32\_t commandId, const lpadc\_conv\_command\_config\_t \* config* )**

## Parameters

|                  |                                                                                          |
|------------------|------------------------------------------------------------------------------------------|
| <i>base</i>      | LPADC peripheral base address.                                                           |
| <i>commandId</i> | ID for command in command buffer. Typically, the available value range is 1 - 15.        |
| <i>config</i>    | Pointer to configuration structure. See to <a href="#">lpadc_conv_command_config_t</a> . |

### 17.6.18 void LPADC\_GetDefaultConvCommandConfig ( [lpadc\\_conv\\_command\\_config\\_t](#) \* *config* )

This function initializes the conversion command's configuration structure with an available settings. The default values are:

```
* config->sampleScaleMode = kLPADC_SampleFullScale;
* config->channelBScaleMode = kLPADC_SampleFullScale;
* config->channelSampleMode = kLPADC_SampleChannelSingleEndSideA
 ;
* config->channelNumber = OU;
* config->chainedNextCmdNumber = OU;
* config->enableAutoChannelIncrement = false;
* config->loopCount = OU;
* config->hardwareAverageMode = kLPADC_HardwareAverageCount1;
* config->sampleTimeMode = kLPADC_SampleTimeADCK3;
* config->hardwareCompareMode = kLPADC_HardwareCompareDisabled;
* config->hardwareCompareValueHigh = OU;
* config->hardwareCompareValueLow = OU;
* config->conversionResolutionMode = kLPADC_ConversionResolutionStandard;
* config->enableWaitTrigger = false;
* config->enableChannelB = false;
```

#### Parameters

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

# Chapter 18

## LPI2C: Low Power Inter-Integrated Circuit Driver

### 18.1 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, 5, 0))`  
*LPI2C driver version.*

## 18.2 Macro Definition Documentation

18.2.1 `#define FSL_LPI2C_DRIVER_VERSION (MAKE_VERSION(2, 5, 0))`

18.2.2 `#define I2C_RETRY_TIMES 0U /* Define to zero means keep waiting until the flag is assert/deassert. */`

## 18.3 Enumeration Type Documentation

### 18.3.1 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.

## 18.4 LPI2C Master Driver

### 18.4.1 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.

### 18.4.2 Data Structure Documentation

#### 18.4.2.1 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`

#### 18.4.2.2 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`

#### 18.4.2.3 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**

### 18.4.2.4 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`

## 18.4.3 Typedef Documentation

### 18.4.3.1 `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. |
|-----------------|------------------------------------------------------------|

## 18.4.4 Enumeration Type Documentation

### 18.4.4.1 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.

### 18.4.4.2 enum lpi2c\_direction\_t

Enumerator

- kLPI2C\_Write* Master transmit.
- kLPI2C\_Read* Master receive.

#### 18.4.4.3 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)

#### 18.4.4.4 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.

#### 18.4.4.5 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.

#### 18.4.4.6 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.

#### 18.4.4.7 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.

#### 18.4.5 Function Documentation

##### 18.4.5.1 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> . |
|-----|---------------------|------------------------------------------------------------------------------------------------------------|

#### 18.4.5.2 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.            |

#### 18.4.5.3 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. |
|-------------|------------------------------------|

#### 18.4.5.4 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. |

#### 18.4.5.5 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. |
|-------------|------------------------------------|

#### 18.4.5.6 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. |

#### 18.4.5.7 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](#)

#### 18.4.5.8 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](#).

#### 18.4.5.9 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. |

#### 18.4.5.10 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. |

#### 18.4.5.11 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.

#### 18.4.5.12 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.  |

#### 18.4.5.13 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.

#### 18.4.5.14 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.

#### 18.4.5.15 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.         |

#### 18.4.5.16 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.  |

#### 18.4.5.17 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.          |

#### 18.4.5.18 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. |

#### 18.4.5.19 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.                            |

#### 18.4.5.20 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.                                     |

#### 18.4.5.21 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.  |

#### 18.4.5.22 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.  |

#### 18.4.5.23 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.                                |

#### 18.4.5.24 **status\_t LPI2C\_MasterTransferBlocking ( LPI2C\_Type \* *base*, lpi2c\_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.  |

**18.4.5.25 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.      |

**18.4.5.26 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. |

**18.4.5.27 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. |

#### 18.4.5.28 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. |

#### 18.4.5.29 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. |

## 18.5 LPI2C Slave Driver

### 18.5.1 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 void [LPI2C\\_SlaveEnableAckStall](#) (LPI2C\_Type \*base, bool enable)  
*Enables or disables ACKSTALL.*
- 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.*

### 18.5.2 Data Structure Documentation

#### 18.5.2.1 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`

### 18.5.2.2 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`

### 18.5.2.3 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`

## 18.5.3 Typedef Documentation

### 18.5.3.1 `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.                           |

**18.5.4 Enumeration Type Documentation****18.5.4.1 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.

#### 18.5.4.2 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.

#### 18.5.4.3 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.

#### 18.5.5 Function Documentation

##### 18.5.5.1 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> . |
|------------|--------------------|----------------------------------------------------------------------------------------------------------------------|

#### 18.5.5.2 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.             |

#### 18.5.5.3 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. |
|-------------|------------------------------------|

#### 18.5.5.4 static void LPI2C\_SlaveReset ( **LPI2C\_Type** \* *base* ) [inline], [**static**]

Parameters

|             |                                    |
|-------------|------------------------------------|
| <i>base</i> | The LPI2C peripheral base address. |
|-------------|------------------------------------|

#### 18.5.5.5 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. |

#### 18.5.5.6 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](#)

#### 18.5.5.7 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](#).

#### 18.5.5.8 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. |

#### 18.5.5.9 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. |

#### 18.5.5.10 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.

#### 18.5.5.11 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.                                                                            |

#### 18.5.5.12 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. |

#### 18.5.5.13 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. |

#### 18.5.5.14 static void LPI2C\_SlaveEnableAckStall ( **LPI2C\_Type** \* *base*, **bool enable** ) [**inline**], [**static**]

When enables ACKSTALL, software can transmit either an ACK or NAK on the I2C bus in response to a byte from the master.

Parameters

|               |                                                         |
|---------------|---------------------------------------------------------|
| <i>base</i>   | The LPI2C peripheral base address.                      |
| <i>enable</i> | True will enable ACKSTALL, false will disable ACKSTALL. |

#### 18.5.5.15 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.

#### 18.5.5.16 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.

#### 18.5.5.17 **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.

#### 18.5.5.18 **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.      |

**18.5.5.19 status\_t LPI2C\_SlaveTransferNonBlocking ( LPI2C\_Type \* *base*, lpi2c\_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 [lpi2c\\_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 <a href="#">lpi2c_slave_handle_t</a> structure which stores the transfer state.                                                                                                                                                                                                             |
| <i>eventMask</i> | Bit mask formed by OR'ing together <a href="#">lpi2c_slave_transfer_event_t</a> enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and <a href="#">kLPI2C_SlaveAllEvents</a> to enable all events. |

## Return values

|                                    |                                                           |
|------------------------------------|-----------------------------------------------------------|
| <a href="#">kStatus_Success</a>    | Slave transfers were successfully started.                |
| <a href="#">kStatus_LPI2C_Busy</a> | Slave transfers have already been started on this handle. |

**18.5.5.20 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 i2c_slave_handle_t structure.                                                              |
| out | <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

|                                      |  |
|--------------------------------------|--|
| <i>kStatus_Success</i>               |  |
| <i>kStatus_NoTransferIn-Progress</i> |  |

**18.5.5.21 void LPI2C\_SlaveTransferAbort ( LPI2C\_Type \* *base*, Ipi2c\_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> |  |

**18.5.5.22 void LPI2C\_SlaveTransferHandleIRQ ( LPI2C\_Type \* *base*, Ipi2c\_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. |

## 18.6 LPI2C Master DMA Driver

### 18.6.1 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.*

### 18.6.2 Data Structure Documentation

#### 18.6.2.1 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.

### 18.6.3 Typedef Documentation

18.6.3.1 **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.                     |

## 18.6.4 Function Documentation

**18.6.4.1 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.                                   |

**18.6.4.2 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. |

#### 18.6.4.3 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. |

#### 18.6.4.4 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. |

## 18.7 LPI2C FreeRTOS Driver

### 18.7.1 Overview

#### Driver version

- `#define FSL_LPI2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 3, 2))`  
*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.*

### 18.7.2 Macro Definition Documentation

#### 18.7.2.1 `#define FSL_LPI2C_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 3, 2))`

### 18.7.3 Function Documentation

#### 18.7.3.1 `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.

### 18.7.3.2 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. |
|---------------|------------------------|

#### 18.7.3.3 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.

## 18.8 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.

### 18.8.1 LPI2C CMSIS Driver

#### 18.8.1.1 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;
```

#### 18.8.1.2 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;

```

### 18.8.1.3 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 19

## LPIT: Low-Power Interrupt Timer

### 19.1 Overview

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

### 19.2 Function groups

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

#### 19.2.1 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.

#### 19.2.2 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

#### 19.2.3 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.

## 19.2.4 Status

Provides functions to get and clear the LPIT status.

## 19.2.5 Interrupt

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

## 19.3 Typical use case

### 19.3.1 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, 1, 0)`)

*Version 2.1.0.*

## 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 void [LPIT\\_SetTimerValue](#) (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.*

## 19.4 Data Structure Documentation

### 19.4.1 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`

**19.4.2 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*

**19.5 Enumeration Type Documentation****19.5.1 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.

**19.5.2 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

### 19.5.3 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

***kLPIT\_Trigger\_TimerChn0*** Channel 0 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn1*** Channel 1 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn2*** Channel 2 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn3*** Channel 3 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn4*** Channel 4 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn5*** Channel 5 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn6*** Channel 6 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn7*** Channel 7 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn8*** Channel 8 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn9*** Channel 9 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn10*** Channel 10 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn11*** Channel 11 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn12*** Channel 12 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn13*** Channel 13 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn14*** Channel 14 is selected as a trigger source.

***kLPIT\_Trigger\_TimerChn15*** Channel 15 is selected as a trigger source.

### 19.5.4 enum lpit\_trigger\_source\_t

Enumerator

***kLPIT\_TriggerSource\_External*** Use external trigger input.

***kLPIT\_TriggerSource\_Internal*** Use internal trigger.

### 19.5.5 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. |

**19.5.6 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. |

**19.6 Function Documentation****19.6.1 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. |

**19.6.2 void LPIT\_Deinit ( LPIT\_Type \* *base* )**

Parameters

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

### 19.6.3 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. |
|---------------|----------------------------------------------|

### 19.6.4 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.         |

### 19.6.5 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> |

### 19.6.6 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> |

### 19.6.7 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](#)

### 19.6.8 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](#)

### 19.6.9 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> |

**19.6.10 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. |

**19.6.11 static void LPIT\_SetTimerValue ( LPIT\_Type \* *base*, lpit\_chnl\_t *channel*, uint32\_t *ticks* ) [inline], [static]**

In the Dual 16-bit Periodic Counter mode, the counter will load and then the lower 16-bits will decrement down to zero, which will assert the output pre-trigger. The upper 16-bits will then decrement down to zero, which will negate the output pre-trigger and set the timer interrupt flag.

## Note

Set TVAL register to 0 or 1 is invalid in compare mode.

## Parameters

|                |                                 |
|----------------|---------------------------------|
| <i>base</i>    | LPIT peripheral base address.   |
| <i>channel</i> | Timer channel number.           |
| <i>ticks</i>   | Timer period in units of ticks. |

### 19.6.12 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.

### 19.6.13 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.         |

### 19.6.14 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.         |

### 19.6.15 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 20

## LPSPI: Low Power Serial Peripheral Interface

### 20.1 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](#)

## 20.2 LPSPI Peripheral driver

### 20.2.1 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.

### 20.2.2 Function groups

#### 20.2.2.1 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.

#### 20.2.2.2 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.

#### 20.2.2.3 LPSPI Transfer Operation

This function group controls the transfer, master send/receive data, and slave send/receive data.

#### 20.2.2.4 LPSPI Status Operation

This function group gets/clears the LPSPI status.

#### 20.2.2.5 LPSPI Block Transfer Operation

This function group transfers a block of data, gets the transfer status, and aborts the transfer.

### 20.2.3 Typical use case

#### 20.2.3.1 Master Operation

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/lpspi

### 20.2.3.2 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, 4, 5)`)  
*LPSPI driver version.*

## Initialization and deinitialization

- void `LPSPI_MasterInit` (`LPSPI_Type` \*base, const `lpspi_master_config_t` \*masterConfig, `uint32_t` srcClock\_Hz)

*Initializes the LPSPI master.*

- 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.*

## 20.2.4 Data Structure Documentation

### 20.2.4.1 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).*
- `bool enableInputDelay`  
*Enable master to sample the input data on a delayed SCK.*

### 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`
- (12) `bool lpspi_master_config_t::enableInputDelay`

This can help improve slave setup time. Refer to device data sheet for specific time length.

### 20.2.4.2 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`

### 20.2.4.3 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.

### 20.2.4.4 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`

### 20.2.4.5 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

### 20.2.5 Macro Definition Documentation

20.2.5.1 #define FSL\_LPSPI\_DRIVER\_VERSION (MAKE\_VERSION(2, 4, 5))

20.2.5.2 #define LPSPI\_DUMMY\_DATA (0x00U)

Dummy data used for tx if there is not txData.

- 20.2.5.3 `#define SPI_RETRY_TIMES 0U /* Define to zero means keep waiting until the flag is assert/deassert. */`
- 20.2.5.4 `#define LPSPI_MASTER_PCS_SHIFT (4U)`
- 20.2.5.5 `#define LPSPI_MASTER_PCS_MASK (0xF0U)`
- 20.2.5.6 `#define LPSPI_SLAVE_PCS_SHIFT (4U)`
- 20.2.5.7 `#define LPSPI_SLAVE_PCS_MASK (0xF0U)`

## 20.2.6 Typedef Documentation

- 20.2.6.1 `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.      |

### 20.2.6.2 `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.      |

## 20.2.7 Enumeration Type Documentation

### 20.2.7.1 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.

### 20.2.7.2 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.

### 20.2.7.3 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.

### 20.2.7.4 enum \_lpspi\_dma\_enable

Enumerator

*kLPSPI\_TxDmaEnable* Transmit data DMA enable.  
*kLPSPI\_RxDmaEnable* Receive data DMA enable.

### 20.2.7.5 enum lpspi\_master\_slave\_mode\_t

Enumerator

*kLPSPI\_Master* LPSPI peripheral operates in master mode.  
*kLPSPI\_Slave* LPSPI peripheral operates in slave mode.

### 20.2.7.6 enum lpspi\_which\_pcs\_t

Enumerator

*kLPSPI\_Pcs0* PCS[0].  
*kLPSPI\_Pcs1* PCS[1].  
*kLPSPI\_Pcs2* PCS[2].  
*kLPSPI\_Pcs3* PCS[3].

### 20.2.7.7 enum lpspi\_pcs\_polarity\_config\_t

Enumerator

*kLPSPI\_PcsActiveHigh* PCS Active High (idles low)

*kLPSPI\_PcsActiveLow* PCS Active Low (idles high)

### 20.2.7.8 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).

### 20.2.7.9 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)

### 20.2.7.10 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.

### 20.2.7.11 enum lpspi\_shift\_direction\_t

Enumerator

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

*kLPSPI\_LsbFirst* Data transfers start with least significant bit.

### 20.2.7.12 enum lpspi\_host\_request\_select\_t

Enumerator

*kLPSPI\_HostReqExtPin* Host Request is an ext pin.

*kLPSPI\_HostReqInternalTrigger* Host Request is an internal trigger.

### 20.2.7.13 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.

### 20.2.7.14 enum lpspi\_pin\_config\_t

Enumerator

*kLPSPI\_SdiInSdoOut* LPSPI SDI input, SDO output.

*kLPSPI\_SdiInSdiOut* LPSPI SDI input, SDI output.

*kLPSPI\_SdoInSdoOut* LPSPI SDO input, SDO output.

*kLPSPI\_SdoInSdiOut* LPSPI SDO input, SDI output.

### 20.2.7.15 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.

### 20.2.7.16 enum lpspi\_transfer\_width\_t

Enumerator

*kLPSPI\_SingleBitXfer* 1-bit shift at a time, data out on SDO, in on SDI (normal mode)

*kLPSPI\_TwoBitXfer* 2-bits shift out on SDO/SDI and in on SDO/SDI

*kLPSPI\_FourBitXfer* 4-bits shift out on SDO/SDI/PCS[3:2] and in on SDO/SDI/PCS[3:2]

### 20.2.7.17 enum lpspi\_delay\_type\_t

Enumerator

*kLPSPI\_PcsToSck* PCS-to-SCK delay.

*kLPSPI\_LastSckToPcs* Last SCK edge to PCS delay.

*kLPSPI\_BetweenTransfer* Delay between transfers.

### 20.2.7.18 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.

### 20.2.7.19 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.

### 20.2.7.20 enum \_lpspi\_transfer\_state

Enumerator

**kLPSPI\_Idle** Nothing in the transmitter/receiver.

**kLPSPI\_Busy** Transfer queue is not finished.

**kLPSPI\_Error** Transfer error.

## 20.2.8 Function Documentation

### 20.2.8.1 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 <a href="#">lpspi_master_config_t</a> . |
| <i>srcClock_Hz</i>  | Module source input clock in Hertz                           |

### 20.2.8.2 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 <a href="#">lpspi_master_config_t</a> structure |
|---------------------|------------------------------------------------------------|

### 20.2.8.3 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 <a href="#">lpspi_slave_config_t</a> . |

#### 20.2.8.4 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 <a href="#">lpspi_slave_config_t</a> structure. |
|--------------------|------------------------------------------------------------|

#### 20.2.8.5 void LPSPI\_Deinit ( [LPSPI\\_Type](#) \* *base* )

Call this API to disable the LPSPI clock.

Parameters

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

#### 20.2.8.6 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. |
|-------------|---------------------------|

#### 20.2.8.7 [uint32\\_t](#) LPSPI\_GetInstance ( [LPSPI\\_Type](#) \* *base* )

Parameters

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

Returns

LPSPI instance.

#### 20.2.8.8 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. |

#### 20.2.8.9 static uint32\_t LPSPI\_GetStatusFlags ( LPSPI\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

The LPSPI status(in SR register).

#### 20.2.8.10 static uint8\_t LPSPI\_GetTxFifoSize ( LPSPI\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

The LPSPI Tx FIFO size.

#### 20.2.8.11 static uint8\_t LPSPI\_GetRxFifoSize ( LPSPI\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

The LPSPI Rx FIFO size.

#### 20.2.8.12 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.

#### 20.2.8.13 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.

#### 20.2.8.14 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).

#### 20.2.8.15 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. |

#### 20.2.8.16 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. |

#### 20.2.8.17 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_Rx_dmaEnable);
*
```

Parameters

|             |                                                     |
|-------------|-----------------------------------------------------|
| <i>base</i> | LPSPI peripheral address.                           |
| <i>mask</i> | The interrupt mask; Use the enum _lpspi_dma_enable. |

#### 20.2.8.18 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_Rx_dmaEnable);
*
```

Parameters

|             |                                                     |
|-------------|-----------------------------------------------------|
| <i>base</i> | LPSPI peripheral address.                           |
| <i>mask</i> | The interrupt mask; Use the enum _lpspi_dma_enable. |

#### 20.2.8.19 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.

#### 20.2.8.20 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.

#### 20.2.8.21 **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.

#### 20.2.8.22 **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. |

#### 20.2.8.23 **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. |

#### 20.2.8.24 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>IsContinuous</i> | True to set the transfer PCS to continuous mode, false to set to uncontinuous mode. |

#### 20.2.8.25 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.

#### 20.2.8.26 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. |

#### 20.2.8.27 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. |

#### 20.2.8.28 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. |

#### 20.2.8.29 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. |

#### 20.2.8.30 `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.

### 20.2.8.31 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 <i>lpspi_delay_type_t</i> . |

### 20.2.8.32 *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 lpspi_delay_type_t. |
| <i>srcClock_Hz</i>         | Module source input clock in Hertz.                                           |

Returns

actual Calculated delay value in nano-seconds.

#### 20.2.8.33 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. |

#### 20.2.8.34 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.

#### 20.2.8.35 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. |

**20.2.8.36 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 <b>lpspi_master_handle_t</b> . |
| <i>callback</i> | DSPI callback.                                         |
| <i>userData</i> | callback function parameter.                           |

**20.2.8.37 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 <b>lpspi_transfer_t</b> structure. |

Returns

status of **status\_t**.

### **20.2.8.38 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 <a href="#">lpspi_transfer_t</a> structure.                      |

Returns

status of status\_t.

### **20.2.8.39 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 lpspi_master_handle_t 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.

### **20.2.8.40 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 lpspi_master_handle_t structure which stores the transfer state. |

#### 20.2.8.41 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 lpspi_master_handle_t structure which stores the transfer state. |

#### 20.2.8.42 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 lpspi_slave_handle_t. |
| <i>callback</i> | DSPI callback.                                |
| <i>userData</i> | callback function parameter.                  |

#### 20.2.8.43 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`.

#### 20.2.8.44 `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`.

#### 20.2.8.45 `void LPSPI_SlaveTransferAbort ( LPSPI_Type * base, lpspi_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 <code>lpspi_slave_handle_t</code> structure which stores the transfer state. |

#### 20.2.8.46 `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. |

## 20.2.9 Variable Documentation

### 20.2.9.1 volatile uint8\_t g\_lpspiDummyData[]

## 20.3 LPSPI eDMA Driver

### 20.3.1 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.*
- status\_t `LPSPI_MasterTransferPrepareEDMALite` (LPSPI\_Type \*base, lpspi\_master\_edma\_handle\_t \*handle, uint32\_t configFlags)  
*LPSPI master config transfer parameter while using eDMA.*
- status\_t `LPSPI_MasterTransferEDMALite` (LPSPI\_Type \*base, lpspi\_master\_edma\_handle\_t \*handle, `lpspi_transfer_t` \*transfer)  
*LPSPI master transfer data using eDMA without configs.*
- 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, 4, 0))`  
*LPSPI EDMA driver version.*

### 20.3.2 Data Structure Documentation

#### 20.3.2.1 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`

- volatile uint32\_t **readRegRemainingTimes**  
*Write TDR register remaining times.*
- uint32\_t **totalByteCount**  
*Read RDR register remaining times.*
- uint32\_t **txBuffIfNull**  
*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`

### 20.3.2.2 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

### 20.3.3 Macro Definition Documentation

20.3.3.1 #define FSL\_LPSPI\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 4, 0))

### 20.3.4 Typedef Documentation

20.3.4.1 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.   |

#### 20.3.4.2 **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.   |

### 20.3.5 Function Documentation

#### 20.3.5.1 **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 <code>lpspi_master_edma_handle_t</code> . |
| <i>callback</i>                  | LPSPI callback.                                                   |
| <i>userData</i>                  | callback function parameter.                                      |
| <i>edmaRxRegTo-RxDataHandle</i>  | edmaRxRegToRxDataHandle pointer to <code>edma_handle_t</code> .   |
| <i>edmaTxData-ToTxReg-Handle</i> | edmaTxDataToTxRegHandle pointer to <code>edma_handle_t</code> .   |

#### 20.3.5.2 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`.

#### 20.3.5.3 status\_t LPSPI\_MasterTransferPrepareEDMALite ( `LPSPI_Type * base,` `lpspi_master_edma_handle_t * handle, uint32_t configFlags` )

This function is preparing to transfer data using eDMA, work with `LPSPI_MasterTransferEDMALite`.

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>configFlags</i> | transfer configuration flags. <code>_lpspi_transfer_config_flag_for_master</code> .           |

Returns

Indicates whether LPSPI master transfer was successful or not.

Return values

|                           |                           |
|---------------------------|---------------------------|
| <i>kStatus_Success</i>    | Execution successfully.   |
| <i>kStatus_LPSPI_Busy</i> | The LPSPI device is busy. |

#### 20.3.5.4 `status_t LPSPI_MasterTransferEDMALite ( 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: This API is only for transfer through DMA without configuration. Before calling this API, you must call `LPSPI_MasterTransferPrepareEDMALite` to configure it once. 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, config field is not used.                 |

Returns

Indicates whether LPSPI master transfer was successful or not.

Return values

|                                |                                    |
|--------------------------------|------------------------------------|
| <i>kStatus_Success</i>         | Execution successfully.            |
| <i>kStatus_LPSPI_Busy</i>      | The LPSPI device is busy.          |
| <i>kStatus_InvalidArgument</i> | The transfer structure is invalid. |

#### 20.3.5.5 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 <b>lpspi_master_edma_handle_t</b> structure which stores the transfer state. |

#### 20.3.5.6 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 <b>lpspi_master_edma_handle_t</b> structure which stores the transfer state. |
| <i>count</i>  | Number of bytes transferred so far by the EDMA transaction.                             |

Returns

status of **status\_t**.

#### 20.3.5.7 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 <code>lpspi_slave_edma_handle_t</code> . |
| <i>callback</i>                  | LPSPI callback.                                                  |
| <i>userData</i>                  | callback function parameter.                                     |
| <i>edmaRxRegTo-RxDataHandle</i>  | edmaRxRegToRxDataHandle pointer to <code>edma_handle_t</code> .  |
| <i>edmaTxData-ToTxReg-Handle</i> | edmaTxDataToTxRegHandle pointer to <code>edma_handle_t</code> .  |

#### 20.3.5.8 `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 <code>lpspi_slave_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`.

#### 20.3.5.9 `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. |

#### 20.3.5.10 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.

## 20.4 LPSPI FreeRTOS Driver

### 20.4.1 Overview

#### Driver version

- #define `FSL_LPSPI_FREERTOS_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))`  
*LPSPI FreeRTOS driver version 2.3.1.*

#### 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.*

### 20.4.2 Macro Definition Documentation

#### 20.4.2.1 #define FSL\_LPSPI\_FREERTOS\_DRIVER\_VERSION (MAKE\_VERSION(2, 3, 1))

### 20.4.3 Function Documentation

#### 20.4.3.1 `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.

#### 20.4.3.2 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. |
|---------------|------------------------|

#### 20.4.3.3 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.

## 20.5 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>.

### 20.5.1 Function groups

#### 20.5.1.1 LPSPI CMSIS GetVersion Operation

This function group will return the DSPI CMSIS Driver version to user.

#### 20.5.1.2 LPSPI CMSIS GetCapabilities Operation

This function group will return the capabilities of this driver.

#### 20.5.1.3 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.

#### 20.5.1.4 LPSPI Transfer Operation

This function group controls the transfer, master send/receive data, and slave send/receive data.

#### 20.5.1.5 LPSPI Status Operation

This function group gets the LPSPI transfer status.

#### 20.5.1.6 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.

## 20.5.2 Typical use case

### 20.5.2.1 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();
```

### 20.5.2.2 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 21

## LPTMR: Low-Power Timer

### 21.1 Overview

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

### 21.2 Function groups

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

#### 21.2.1 Initialization and deinitialization

The function [LPTMR\\_Init\(\)](#) initializes the LPTMR with specified configurations. The function [LPTMR\\_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the LPTMR for a timer or a pulse counter mode mode. It also sets up the LPTMR's free running mode operation and a clock source.

The function [LPTMR\\_DeInit\(\)](#) disables the LPTMR module and gates the module clock.

#### 21.2.2 Timer period Operations

The function [LPTMR\\_SetTimerPeriod\(\)](#) sets the timer period in units of count. Timers counts from 0 to the count value set here.

The function [LPTMR\\_GetCurrentTimerCount\(\)](#) reads the current timer counting value. This function returns the real-time timer counting value ranging from 0 to a timer period.

The timer period operation function takes the count value in ticks. Call the utility macros provided in the `fsl_common.h` file to convert to microseconds or milliseconds.

#### 21.2.3 Start and Stop timer operations

The function [LPTMR\\_StartTimer\(\)](#) starts the timer counting. After calling this function, the timer counts up to the counter value set earlier by using the [LPTMR\\_SetPeriod\(\)](#) function. Each time the timer reaches the count value and increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

The function [LPTMR\\_StopTimer\(\)](#) stops the timer counting and resets the timer's counter register.

## 21.2.4 Status

Provides functions to get and clear the LPTMR status.

## 21.2.5 Interrupt

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

## 21.3 Typical use case

### 21.3.1 LPTMR tick example

Updates the LPTMR period and toggles an LED periodically. 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.*

## 21.4 Data Structure Documentation

### 21.4.1 struct lptmr\_config\_t

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the [LPTMR\\_GetDefaultConfig\(\)](#) function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

## Data Fields

- [lptmr\\_timer\\_mode\\_t timerMode](#)  
*Time counter mode or pulse counter mode.*
- [lptmr\\_pin\\_select\\_t pinSelect](#)  
*LPTMR pulse input pin select; used only in pulse counter mode.*
- [lptmr\\_pin\\_polarity\\_t pinPolarity](#)  
*LPTMR pulse input pin polarity; used only in pulse counter mode.*
- bool [enableFreeRunning](#)

*True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set.*

- bool [bypassPrescaler](#)  
*True: bypass prescaler; false: use clock from prescaler.*
- [lptmr\\_prescaler\\_clock\\_select\\_t prescalerClockSource](#)  
*LPTMR clock source.*
- [lptmr\\_prescaler\\_glitch\\_value\\_t value](#)  
*Prescaler or glitch filter value.*

## 21.5 Enumeration Type Documentation

### 21.5.1 enum lptmr\_pin\_select\_t

Enumerator

***kLPTMR\_PinSelectInput\_0*** Pulse counter input 0 is selected.

***kLPTMR\_PinSelectInput\_1*** Pulse counter input 1 is selected.

***kLPTMR\_PinSelectInput\_2*** Pulse counter input 2 is selected.

***kLPTMR\_PinSelectInput\_3*** Pulse counter input 3 is selected.

### 21.5.2 enum lptmr\_pin\_polarity\_t

Enumerator

***kLPTMR\_PinPolarityActiveHigh*** Pulse Counter input source is active-high.

***kLPTMR\_PinPolarityActiveLow*** Pulse Counter input source is active-low.

### 21.5.3 enum lptmr\_timer\_mode\_t

Enumerator

***kLPTMR\_TimerModeTimeCounter*** Time Counter mode.

***kLPTMR\_TimerModePulseCounter*** Pulse Counter mode.

### 21.5.4 enum lptmr\_prescaler\_glitch\_value\_t

Enumerator

***kLPTMR\_Prescale\_Glitch\_0*** Prescaler divide 2, glitch filter does not support this setting.

***kLPTMR\_Prescale\_Glitch\_1*** Prescaler divide 4, glitch filter 2.

***kLPTMR\_Prescale\_Glitch\_2*** Prescaler divide 8, glitch filter 4.

***kLPTMR\_Prescale\_Glitch\_3*** Prescaler divide 16, glitch filter 8.

***kLPTMR\_Prescale\_Glitch\_4*** Prescaler divide 32, glitch filter 16.

- kLPTMR\_Prescale\_Glitch\_5* Prescaler divide 64, glitch filter 32.
- kLPTMR\_Prescale\_Glitch\_6* Prescaler divide 128, glitch filter 64.
- kLPTMR\_Prescale\_Glitch\_7* Prescaler divide 256, glitch filter 128.
- kLPTMR\_Prescale\_Glitch\_8* Prescaler divide 512, glitch filter 256.
- kLPTMR\_Prescale\_Glitch\_9* Prescaler divide 1024, glitch filter 512.
- kLPTMR\_Prescale\_Glitch\_10* Prescaler divide 2048 glitch filter 1024.
- kLPTMR\_Prescale\_Glitch\_11* Prescaler divide 4096, glitch filter 2048.
- kLPTMR\_Prescale\_Glitch\_12* Prescaler divide 8192, glitch filter 4096.
- kLPTMR\_Prescale\_Glitch\_13* Prescaler divide 16384, glitch filter 8192.
- kLPTMR\_Prescale\_Glitch\_14* Prescaler divide 32768, glitch filter 16384.
- kLPTMR\_Prescale\_Glitch\_15* Prescaler divide 65536, glitch filter 32768.

### 21.5.5 enum lptmr\_prescaler\_clock\_select\_t

Note

Clock connections are SoC-specific

Enumerator

- kLPTMR\_PrescalerClock\_0* Prescaler/glitch filter clock 0 selected.
- kLPTMR\_PrescalerClock\_1* Prescaler/glitch filter clock 1 selected.
- kLPTMR\_PrescalerClock\_2* Prescaler/glitch filter clock 2 selected.
- kLPTMR\_PrescalerClock\_3* Prescaler/glitch filter clock 3 selected.

### 21.5.6 enum lptmr\_interrupt\_enable\_t

Enumerator

*kLPTMR\_TimerInterruptEnable* Timer interrupt enable.

### 21.5.7 enum lptmr\_status\_flags\_t

Enumerator

*kLPTMR\_TimerCompareFlag* Timer compare flag.

## 21.6 Function Documentation

### 21.6.1 void LPTMR\_Init ( LPTMR\_Type \* *base*, const lptmr\_config\_t \* *config* )

## Note

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

## Parameters

|               |                                                 |
|---------------|-------------------------------------------------|
| <i>base</i>   | LPTMR peripheral base address                   |
| <i>config</i> | A pointer to the LPTMR configuration structure. |

**21.6.2 void LPTMR\_Deinit ( LPTMR\_Type \* *base* )**

## Parameters

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

**21.6.3 void LPTMR\_GetDefaultConfig ( lptmr\_config\_t \* *config* )**

The default values are as follows.

```
* config->timerMode = kLPTMR_TimerModeTimeCounter;
* config->pinSelect = kLPTMR_PinSelectInput_0;
* config->polarity = 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. |
|---------------|-------------------------------------------------|

**21.6.4 static void LPTMR\_EnableInterrupts ( LPTMR\_Type \* *base*, uint32\_t *mask* )  
[inline], [static]**

## Parameters

|             |                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                         |
| <i>mask</i> | The interrupts to enable. This is a logical OR of members of the enumeration <a href="#">lptmr_interrupt_enable_t</a> |

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

Parameters

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | LPTMR peripheral base address                                                                                            |
| <i>mask</i> | The interrupts to disable. This is a logical OR of members of the enumeration <a href="#">lptmr_interrupt_enable_t</a> . |

### 21.6.6 static uint32\_t LPTMR\_GetEnabledInterrupts ( LPTMR\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [lptmr\\_interrupt\\_enable\\_t](#)

### 21.6.7 static 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. |

### 21.6.8 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](#)

### 21.6.9 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 <a href="#">lptmr_status_flags_t</a> . |

### 21.6.10 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. |

### 21.6.11 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

**21.6.12 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 |
|-------------|-------------------------------|

**21.6.13 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 22

# LPUART: Low Power Universal Asynchronous Receiver-/Transmitter Driver

### 22.1 Overview

#### Modules

- [LPUART CMSIS Driver](#)
- [LPUART Driver](#)
- [LPUART FreeRTOS Driver](#)
- [LPUART eDMA Driver](#)

## 22.2 LPUART Driver

### 22.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Low Power UART (LPUART) module of MCUXpresso SDK devices.

### 22.2.2 Typical use case

#### 22.2.2.1 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, 7, 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.*

## 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.*

## 22.2.3 Data Structure Documentation

### 22.2.3.1 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](#)  
*RX FIFO watermark.*
- 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`

### 22.2.3.2 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`

### 22.2.3.3 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`

### 22.2.4 Macro Definition Documentation

22.2.4.1 `#define FSL_LPUART_DRIVER_VERSION (MAKE_VERSION(2, 7, 3))`

22.2.4.2 `#define UART_RETRY_TIMES 0U /* Defining to zero means to keep waiting for the flag until it is assert/deassert. */`

### 22.2.5 Typedef Documentation

22.2.5.1 `typedef void(* lpuart_transfer_callback_t)(LPUART_Type *base, lpuart_handle_t *handle, status_t status, void *userData)`

### 22.2.6 Enumeration Type Documentation

#### 22.2.6.1 anonymous enum

Enumerator

*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.

### 22.2.6.2 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.

### 22.2.6.3 enum lpuart\_data\_bits\_t

Enumerator

*kLPUART\_EightDataBits* Eight data bit.  
*kLPUART\_SevenDataBits* Seven data bit.

### 22.2.6.4 enum lpuart\_stop\_bit\_count\_t

Enumerator

*kLPUART\_OneStopBit* One stop bit.  
*kLPUART\_TwoStopBit* Two stop bits.

### 22.2.6.5 enum lpuart\_transmit\_cts\_source\_t

Enumerator

*kLPUART\_CtsSourcePin* CTS resource is the LPUART\_CTS pin.

***kLPUART\_CtsSourceMatchResult*** CTS resource is the match result.

### 22.2.6.6 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.

### 22.2.6.7 enum lpuart\_idle\_type\_select\_t

Enumerator

***kLPUART\_IdleTypeStartBit*** Start counting after a valid start bit.

***kLPUART\_IdleTypeStopBit*** Start counting after a stop bit.

### 22.2.6.8 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.

### 22.2.6.9 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

#### 22.2.6.10 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

## 22.2.7 Function Documentation

### 22.2.7.1 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. |
|-------------|---------------------------------|

### 22.2.7.2 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                        |

### 22.2.7.3 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. |
|-------------|---------------------------------|

#### 22.2.7.4 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. |
|---------------|---------------------------------------|

#### 22.2.7.5 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.                                |

#### 22.2.7.6 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. |

#### 22.2.7.7 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.          |

#### 22.2.7.8 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. |

#### 22.2.7.9 static void LPUART\_SetRx\_fifoWatermark ( LPUART\_Type \* *base*, uint8\_t *water* ) [inline], [static]

Parameters

|              |                                 |
|--------------|---------------------------------|
| <i>base</i>  | LPUART peripheral base address. |
| <i>water</i> | Rx FIFO watermark.              |

### 22.2.7.10 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.              |

### 22.2.7.11 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`.

### 22.2.7.12 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.                                                         |

**22.2.7.13 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 <a href="#">_lpuart_interrupt_enable</a> . |

**22.2.7.14 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 <a href="#">_lpuart_interrupt_enable</a> . |

### 22.2.7.15 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](#).

### 22.2.7.16 uint32\_t LPUARTGetInstance ( LPUART\_Type \* *base* )

Parameters

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

Returns

LPUART instance.

### 22.2.7.17 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. |

#### **22.2.7.18 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. |

#### **22.2.7.19 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.  |

#### **22.2.7.20 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.

22.2.7.21 **static uint8\_t LPUART\_GetRxFifoCount( LPUART\_Type \* *base* ) [inline],  
[static]**

Parameters

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

Returns

rx FIFO data count.

### 22.2.7.22 static uint8\_t LPUART\_GetTxFifoCount ( LPUART\_Type \* *base* ) [inline], [static]

Parameters

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

Returns

tx FIFO data count.

### 22.2.7.23 void LPUART\_SendAddress ( LPUART\_Type \* *base*, uint8\_t *address* )

Parameters

|                |                                 |
|----------------|---------------------------------|
| <i>base</i>    | LPUART peripheral base address. |
| <i>address</i> | LPUART slave address.           |

### 22.2.7.24 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_-<br/>Timeout</i> | Transmission timed out and was aborted. |
| <i>kStatus_Success</i>              | Successfully wrote all data.            |

### 22.2.7.25 **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-<br/>HardwareOverrun</i> | Receiver overrun happened while receiving data. |
| <i>kStatus_LPUART_Noise-<br/>Error</i>        | Noise error happened while receiving data.      |
| <i>kStatus_LPUART_-<br/>FramingError</i>      | Framing error happened while receiving data.    |
| <i>kStatus_LPUART_Parity-<br/>Error</i>       | Parity error happened while receiving data.     |
| <i>kStatus_LPUART_-<br/>Timeout</i>           | Transmission timed out and was aborted.         |
| <i>kStatus_Success</i>                        | Successfully received all data.                 |

### 22.2.7.26 void LPUART\_TransferCreateHandle ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle*, Ipuart\_transfer\_callback\_t *callback*, void \* *userData* )

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

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

Parameters

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

### 22.2.7.27 status\_t LPUART\_TransferSendNonBlocking ( LPUART\_Type \* *base*, Ipuart\_handle\_t \* *handle*, Ipuart\_transfer\_t \* *xfer* )

This function send data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data written to the transmitter register. When all data is written to the TX register in the ISR, the LPUART driver calls the callback function and passes the [kStatus\\_LPUART\\_TxIdle](#) as status parameter.

Note

The [kStatus\\_LPUART\\_TxIdle](#) is passed to the upper layer when all data are written to the TX register. However, there is no check to ensure that all the data sent out. Before disabling the T-X, check the [kLPUART\\_TransmissionCompleteFlag](#) to ensure that the transmit is finished.

Parameters

|               |                                                                    |
|---------------|--------------------------------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.                                    |
| <i>handle</i> | LPUART handle pointer.                                             |
| <i>xfer</i>   | LPUART transfer structure, see <a href="#">Ipuart_transfer_t</a> . |

Return values

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

### 22.2.7.28 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.                                                                     |

### 22.2.7.29 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.          |

22.2.7.30 `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.

### 22.2.7.31 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.          |

### 22.2.7.32 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_NoTransferIn-Progress</i> | No send in progress.                                  |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

### 22.2.7.33 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 <i>uart_transfer_t</i> . |
| <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.                                        |

### 22.2.7.34 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.          |

### 22.2.7.35 status\_t LPUART\_TransferGetReceiveCount ( LPUART\_Type \* *base*,                   lpuart\_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 <i>count</i> ; |

### 22.2.7.36 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.          |

### 22.2.7.37 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.          |

## 22.3 LPUART eDMA Driver

### 22.3.1 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, 6, 0))  
*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.*

## 22.3.2 Data Structure Documentation

### 22.3.2.1 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`

### 22.3.3 Macro Definition Documentation

**22.3.3.1 #define FSL\_LPUART\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 6, 0))**

### 22.3.4 Typedef Documentation

**22.3.4.1 typedef void(\* lpuart\_edma\_transfer\_callback\_t)(LPUART\_Type \*base, lpuart\_edma\_handle\_t \*handle, status\_t status, void \*userData)**

### 22.3.5 Function Documentation

**22.3.5.1 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 lpuart_edma_handle_t structure.     |
| <i>callback</i>     | Callback function.                             |
| <i>userData</i>     | User data.                                     |
| <i>txEdmaHandle</i> | User requested DMA handle for TX DMA transfer. |
| <i>rxEdmaHandle</i> | User requested DMA handle for RX DMA transfer. |

### 22.3.5.2 status\_t LPUART\_SendEDMA ( LPUART\_Type \* *base*, Ipuart\_edma\_handle\_t \* *handle*, Ipuart\_transfer\_t \* *xfer* )

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

Parameters

|               |                                                                         |
|---------------|-------------------------------------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.                                         |
| <i>handle</i> | LPUART handle pointer.                                                  |
| <i>xfer</i>   | LPUART eDMA transfer structure. See <a href="#">Ipuart_transfer_t</a> . |

Return values

|                                |                             |
|--------------------------------|-----------------------------|
| <i>kStatus_Success</i>         | if succeed, others failed.  |
| <i>kStatus_LPUART_TxBusy</i>   | Previous transfer on going. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.           |

### 22.3.5.3 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 lpuart_edma_handle_t structure. |

|             |                                                                         |
|-------------|-------------------------------------------------------------------------|
| <i>xfer</i> | LPUART eDMA transfer structure, see <a href="#">lpuart_transfer_t</a> . |
|-------------|-------------------------------------------------------------------------|

Return values

|                                |                            |
|--------------------------------|----------------------------|
| <i>kStatus_Success</i>         | if succeed, others fail.   |
| <i>kStatus_LPUART_Rx-Busy</i>  | Previous transfer ongoing. |
| <i>kStatus_InvalidArgument</i> | Invalid argument.          |

#### 22.3.5.4 void LPUART\_TransferAbortSendEDMA ( LPUART\_Type \* *base*, lpuart\_edma\_handle\_t \* *handle* )

This function aborts the sent data using eDMA.

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.            |
| <i>handle</i> | Pointer to lpuart_edma_handle_t structure. |

#### 22.3.5.5 void LPUART\_TransferAbortReceiveEDMA ( LPUART\_Type \* *base*, lpuart\_edma\_handle\_t \* *handle* )

This function aborts the received data using eDMA.

Parameters

|               |                                            |
|---------------|--------------------------------------------|
| <i>base</i>   | LPUART peripheral base address.            |
| <i>handle</i> | Pointer to lpuart_edma_handle_t structure. |

#### 22.3.5.6 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_NoTransferIn-Progress</i> | No send in progress.                                  |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

### 22.3.5.7 **status\_t LPUART\_TransferGetReceiveCountEDMA ( LPUART\_Type \* *base*, Ipuart\_edma\_handle\_t \* *handle*, uint32\_t \* *count* )**

This function gets the number of received bytes.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | LPUART peripheral base address. |
| <i>handle</i> | LPUART handle pointer.          |
| <i>count</i>  | Receive bytes count.            |

Return values

|                                      |                                                       |
|--------------------------------------|-------------------------------------------------------|
| <i>kStatus_NoTransferIn-Progress</i> | No receive in progress.                               |
| <i>kStatus_InvalidArgument</i>       | Parameter is invalid.                                 |
| <i>kStatus_Success</i>               | Get successfully through the parameter <i>count</i> ; |

### 22.3.5.8 **void LPUART\_TransferEdmaHandleIRQ ( LPUART\_Type \* *base*, void \* *IpuartEdmaHandle* )**

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>lpuartEdma-Handle</i> | LPUART handle pointer.          |

## 22.4 LPUART FreeRTOS Driver

### 22.4.1 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.*

### 22.4.2 Data Structure Documentation

#### 22.4.2.1 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`

## 22.4.3 Macro Definition Documentation

**22.4.3.1 #define FSL\_LPUART\_FREERTOS\_DRIVER\_VERSION (MAKE\_VERSION(2, 6, 0))**

## 22.4.4 Function Documentation

**22.4.4.1 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

#### 22.4.4.2 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. |
|---------------|-------------------------|

#### 22.4.4.3 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.   |

#### 22.4.4.4 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. |

#### **22.4.4.5 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.

#### **22.4.4.6 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.

## 22.5 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.

### 22.5.1 Function groups

#### 22.5.1.1 LPUART CMSIS GetVersion Operation

This function group will return the LPUART CMSIS Driver version to user.

#### 22.5.1.2 LPUART CMSIS GetCapabilities Operation

This function group will return the capabilities of this driver.

#### 22.5.1.3 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.

#### 22.5.1.4 LPUART CMSIS Transfer Operation

This function group controls the transfer, send/receive data.

#### 22.5.1.5 LPUART CMSIS Status Operation

This function group gets the LPUART transfer status.

### **22.5.1.6 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 23

## LTC: LP Trusted Cryptography

### 23.1 Overview

The MCUXpresso SDK provides a peripheral driver for the LP Trusted Cryptography (LTC) module of MCUXpresso SDK devices. LP Trusted Cryptography is a set of cryptographpic hardware accelerator engines that share common registers. LTC architecture can support AES, DES, 3DES, MDHA (SHA), RSA, and ECC. The actual list of implemented cryptographpic hardware accelerator engines depends on the specific microcontroller.

The driver comprises two sets of API functions.

In the first set, blocking synchronous APIs are provided, for all operations supported by LTC hardware. The LTC operations are complete (and results are made availabe for further usage) when a function returns. When called, these functions do not return until an LTC operation is complete. These functions use main CPU for simple polling loops to determine operation complete or error status and also for plaintext or ciphertext data movements. The driver functions are not re-entrant. These functions provide typical interface to upper layer or application software.

In the second set, DMA support for symmetric LTC processing is provided, for AES and DES engines. APIs in the second set use DMA for data movement to and from the LTC input and output FIFOs. By using these functions, main CPU is not used for plaintext or ciphertext data movements (DMA is used instead). Thus, CPU processing power can be used for other application tasks, at cost of decreased maximum data throughput (because of DMA module and transactions management overhead). These functions provide less typical interface, for applications that must offload main CPU while ciphertext or plaintext is being processed, at cost of longer cryptographpic processing time.

### 23.2 LTC Driver Initialization and Configuration

LTC Driver is initialized by calling the `LTC_Init()` function, it enables the LTC module clock in the SIM module. If AES or DES engine is used and the LTC module implementation features the LTC DPA Mask Seed register, seed the DPA mask generator by using the seed from a random number generator. The `LTC_SetDpaMaskSeed()` function is provided to set the DPA mask seed.

### 23.3 Comments about API usage in RTOS

LTC operations provided by this driver are not re-entrant. Thus, application software shall ensure the LTC module operation is not requested from different tasks or interrupt service routines while an operation is in progress.

### 23.4 Comments about API usage in interrupt handler

All APIs can be used from interrupt handler although execution time shall be considered (interrupt latency of equal and lower priority interrupts increases).

## 23.5 LTC Driver Examples

### 23.5.1 Simple examples

Initialize LTC after Power On Reset or reset cycle Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ltcEncrypt plaintext by DES engine Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ltc Encrypt plaintext by AES engine Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ltc Compute keyed hash by AES engine (CMAC) Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ltc Compute hash by MDHA engine (SHA-256) Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ltc Compute modular integer exponentiation Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ltc Compute elliptic curve point multiplication Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/ltc

## Modules

- [LTC Blocking APIs](#)
- [LTC Non-blocking eDMA APIs](#)

## Functions

- void [LTC\\_Init](#) (LTC\_Type \*base)  
*Initializes the LTC driver.*
- void [LTC\\_Deinit](#) (LTC\_Type \*base)  
*Deinitializes the LTC driver.*

## Driver version

- #define [FSL\\_LTC\\_DRIVER\\_VERSION](#) (MAKE\_VERSION(2, 0, 16))  
*LTC driver version.*

## 23.6 Macro Definition Documentation

### 23.6.1 #define FSL\_LTC\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 16))

Version 2.0.16.

Current version: 2.0.16

Change log:

- Version 2.0.1
  - fixed warning during g++ compilation
- Version 2.0.2
  - fixed [KPSDK-10932][LTC][SHA] [LTC\\_HASH\(\)](#) blocks indefinitely when message size exceeds 4080 bytes
- Version 2.0.3
  - fixed LTC\_PKHA\_CompareBigNum() in case an integer argument is an array of all zeroes

- Version 2.0.4
  - constant LTC\_PKHA\_CompareBigNum() processing time
- Version 2.0.5
  - Fix MISRA issues
- Version 2.0.6
  - fixed [KPSDK-23603][LTC] AES Decrypt in ECB and CBC modes fail when ciphertext size > 0xff0 bytes
- Version 2.0.7
  - Fix MISRA-2012 issues
- Version 2.0.8
  - Fix Coverity issues
- Version 2.0.9
  - Fix sign-compare warning in ltc\_set\_context and in ltc\_get\_context
- Version 2.0.10
  - Fix MISRA-2012 issues
- Version 2.0.11
  - Fix MISRA-2012 issues
- Version 2.0.12
  - Fix AES Decrypt in CBC modes fail when used kLTC\_DecryptKey.
- Version 2.0.13
  - Add feature macro FSL\_FEATURE\_LTC\_HAS\_NO\_CLOCK\_CONTROL\_BIT into LTC\_Init function.
- Version 2.0.14
  - Add feature macro FSL\_FEATURE\_LTC\_HAS\_NO\_CLOCK\_CONTROL\_BIT into LTC\_Deinit function.
- Version 2.0.15
  - Fix MISRA-2012 issues
- Version 2.0.16
  - Fix uninitialized GCC warning in [LTC\\_AES\\_GenerateDecryptKey\(\)](#)

## 23.7 Function Documentation

### 23.7.1 void LTC\_Init ( LTC\_Type \* *base* )

This function initializes the LTC driver.

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | LTC peripheral base address |
|-------------|-----------------------------|

### 23.7.2 void LTC\_Deinit ( LTC\_Type \* *base* )

This function deinitializes the LTC driver.

### Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | LTC peripheral base address |
|-------------|-----------------------------|

## 23.8 LTC Blocking APIs

### 23.8.1 Overview

This section describes the programming interface of the LTC Synchronous Blocking functions

### Modules

- [LTC AES driver](#)
- [LTC DES driver](#)
- [LTC HASH driver](#)
- [LTC PKHA driver](#)

## 23.8.2 LTC DES driver

### 23.8.2.1 Overview

This section describes the programming interface of the LTC DES driver.

#### Macros

- `#define LTC_DES_KEY_SIZE 8`  
*LTC DES key size - 64 bits.*
- `#define LTC_DES_IV_SIZE 8`  
*LTC DES IV size - 8 bytes.*

### 23.8.2.2 Macro Definition Documentation

#### 23.8.2.2.1 `#define LTC_DES_KEY_SIZE 8`

## 23.8.3 LTC AES driver

### 23.8.3.1 Overview

This section describes the programming interface of the LTC AES driver.

#### Macros

- `#define LTC_AES_BLOCK_SIZE 16U`  
*AES block size in bytes.*
- `#define LTC_AES_IV_SIZE 16`  
*AES Input Vector size in bytes.*
- `#define LTC_AES_DecryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft) LTC_AES_CryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)`  
*AES CTR decrypt is mapped to the AES CTR generic operation.*
- `#define LTC_AES_EncryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft) LTC_AES_CryptCtr(base, input, output, size, counter, key, keySize, counterlast, szLeft)`  
*AES CTR encrypt is mapped to the AES CTR generic operation.*

#### Enumerations

- `enum ltc_aes_key_t {  
 kLTC_EncryptKey = 0U,  
 kLTC_DecryptKey = 1U }`  
*Type of AES key for ECB and CBC decrypt operations.*

#### Functions

- `status_t LTC_AES_GenerateDecryptKey (LTC_Type *base, const uint8_t *encryptKey, uint8_t *decryptKey, uint32_t keySize)`  
*Transforms an AES encrypt key (forward AES) into the decrypt key (inverse AES).*
- `status_t LTC_AES_EncryptEcb (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t *key, uint32_t keySize)`  
*Encrypts AES using the ECB block mode.*
- `status_t LTC_AES_DecryptEcb (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t *key, uint32_t keySize, ltc_aes_key_t keyType)`  
*Decrypts AES using ECB block mode.*
- `status_t LTC_AES_EncryptCbc (LTC_Type *base, const uint8_t *plaintext, uint8_t *ciphertext, uint32_t size, const uint8_t iv[LTC_AES_IV_SIZE], const uint8_t *key, uint32_t keySize)`  
*Encrypts AES using CBC block mode.*
- `status_t LTC_AES_DecryptCbc (LTC_Type *base, const uint8_t *ciphertext, uint8_t *plaintext, uint32_t size, const uint8_t iv[LTC_AES_IV_SIZE], const uint8_t *key, uint32_t keySize, ltc_aes_key_t keyType)`  
*Decrypts AES using CBC block mode.*
- `status_t LTC_AES_CryptCtr (LTC_Type *base, const uint8_t *input, uint8_t *output, uint32_t size, uint8_t counter[LTC_AES_BLOCK_SIZE], const uint8_t *key, uint32_t keySize, uint8_t`

- counterlast[LTC\_AES\_BLOCK\_SIZE], uint32\_t \*szLeft)  
*Encrypts or decrypts AES using CTR block mode.*
- **status\_t LTC\_AES\_EncryptTagCcm** (LTC\_Type \*base, const uint8\_t \*plaintext, uint8\_t \*ciphertext, uint32\_t size, const uint8\_t \*iv, uint32\_t ivSize, const uint8\_t \*aad, uint32\_t aadSize, const uint8\_t \*key, uint32\_t keySize, uint8\_t \*tag, uint32\_t tagSize)  
*Encrypts AES and tags using CCM block mode.*
  - **status\_t LTC\_AES\_DecryptTagCcm** (LTC\_Type \*base, const uint8\_t \*ciphertext, uint8\_t \*plaintext, uint32\_t size, const uint8\_t \*iv, uint32\_t ivSize, const uint8\_t \*aad, uint32\_t aadSize, const uint8\_t \*key, uint32\_t keySize, const uint8\_t \*tag, uint32\_t tagSize)  
*Decrypts AES and authenticates using CCM block mode.*

### 23.8.3.2 Enumeration Type Documentation

#### 23.8.3.2.1 enum ltc\_aes\_key\_t

Enumerator

**kLTC\_EncryptKey** Input key is an encrypt key.

**kLTC\_DecryptKey** Input key is a decrypt key.

### 23.8.3.3 Function Documentation

#### 23.8.3.3.1 status\_t LTC\_AES\_GenerateDecryptKey ( LTC\_Type \* *base*, const uint8\_t \* *encryptKey*, uint8\_t \* *decryptKey*, uint32\_t *keySize* )

Transforms the AES encrypt key (forward AES) into the decrypt key (inverse AES). The key derived by this function can be used as a direct load decrypt key for AES ECB and CBC decryption operations (keyType argument).

Parameters

|            |                   |                                                                       |
|------------|-------------------|-----------------------------------------------------------------------|
|            | <i>base</i>       | LTC peripheral base address                                           |
|            | <i>encryptKey</i> | Input key for decrypt key transformation                              |
| <i>out</i> | <i>decryptKey</i> | Output key, the decrypt form of the AES key.                          |
|            | <i>keySize</i>    | Size of the input key and output key in bytes. Must be 16, 24, or 32. |

Returns

Status from key generation operation

#### 23.8.3.3.2 status\_t LTC\_AES\_EncryptEcb ( LTC\_Type \* *base*, const uint8\_t \* *plaintext*, uint8\_t \* *ciphertext*, uint32\_t *size*, const uint8\_t \* *key*, uint32\_t *keySize* )

Encrypts AES using the ECB block mode.

Parameters

|     |                   |                                                                       |
|-----|-------------------|-----------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                           |
|     | <i>plaintext</i>  | Input plain text to encrypt                                           |
| out | <i>ciphertext</i> | Output cipher text                                                    |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes. |
|     | <i>key</i>        | Input key to use for encryption                                       |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.               |

Returns

Status from encrypt operation

**23.8.3.3.3 status\_t LTC\_AES\_DecryptEcb ( LTC\_Type \* *base*, const uint8\_t \* *ciphertext*, uint8\_t \* *plaintext*, uint32\_t *size*, const uint8\_t \* *key*, uint32\_t *keySize*, ltc\_aes\_key\_t *keyType* )**

Decrypts AES using ECB block mode.

Parameters

|     |                   |                                                                                            |
|-----|-------------------|--------------------------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                                                |
|     | <i>ciphertext</i> | Input cipher text to decrypt                                                               |
| out | <i>plaintext</i>  | Output plain text                                                                          |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes.                      |
|     | <i>key</i>        | Input key.                                                                                 |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.                                    |
|     | <i>keyType</i>    | Input type of the key (allows to directly load decrypt key for AES ECB decrypt operation.) |

Returns

Status from decrypt operation

**23.8.3.3.4 status\_t LTC\_AES\_EncryptCbc ( LTC\_Type \* *base*, const uint8\_t \* *plaintext*, uint8\_t \* *ciphertext*, uint32\_t *size*, const uint8\_t iv[LTC\_AES\_IV\_SIZE], const uint8\_t \* *key*, uint32\_t *keySize* )**

Parameters

|     |                   |                                                                       |
|-----|-------------------|-----------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                           |
|     | <i>plaintext</i>  | Input plain text to encrypt                                           |
| out | <i>ciphertext</i> | Output cipher text                                                    |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes. |
|     | <i>iv</i>         | Input initial vector to combine with the first input block.           |
|     | <i>key</i>        | Input key to use for encryption                                       |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.               |

Returns

Status from encrypt operation

**23.8.3.3.5 status\_t LTC\_AES\_DecryptCbc ( LTC\_Type \* *base*, const uint8\_t \* *ciphertext*, uint8\_t \* *plaintext*, uint32\_t *size*, const uint8\_t *iv*[LTC\_AES\_IV\_SIZE], const uint8\_t \* *key*, uint32\_t *keySize*, ltc\_aes\_key\_t *keyType* )**

Parameters

|     |                   |                                                                                            |
|-----|-------------------|--------------------------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                                                |
|     | <i>ciphertext</i> | Input cipher text to decrypt                                                               |
| out | <i>plaintext</i>  | Output plain text                                                                          |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes.                      |
|     | <i>iv</i>         | Input initial vector to combine with the first input block.                                |
|     | <i>key</i>        | Input key to use for decryption                                                            |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.                                    |
|     | <i>keyType</i>    | Input type of the key (allows to directly load decrypt key for AES CBC decrypt operation.) |

Returns

Status from decrypt operation

**23.8.3.3.6 status\_t LTC\_AES\_CryptCtr ( LTC\_Type \* *base*, const uint8\_t \* *input*, uint8\_t \* *output*, uint32\_t *size*, uint8\_t *counter*[LTC\_AES\_BLOCK\_SIZE], const uint8\_t \* *key*, uint32\_t *keySize*, uint8\_t *counterlast*[LTC\_AES\_BLOCK\_SIZE], uint32\_t \* *szLeft* )**

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is

that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

#### Parameters

|         |                    |                                                                                                               |
|---------|--------------------|---------------------------------------------------------------------------------------------------------------|
|         | <i>base</i>        | LTC peripheral base address                                                                                   |
|         | <i>input</i>       | Input data for CTR block mode                                                                                 |
| out     | <i>output</i>      | Output data for CTR block mode                                                                                |
|         | <i>size</i>        | Size of input and output data in bytes                                                                        |
| in, out | <i>counter</i>     | Input counter (updates on return)                                                                             |
|         | <i>key</i>         | Input key to use for forward AES cipher                                                                       |
|         | <i>keySize</i>     | Size of the input key, in bytes. Must be 16, 24, or 32.                                                       |
| out     | <i>counterlast</i> | Output cipher of last counter, for chained CTR calls. NULL can be passed if chained calls are not used.       |
| out     | <i>szLeft</i>      | Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used. |

#### Returns

Status from encrypt operation

**23.8.3.3.7 status\_t LTC\_AES\_EncryptTagCcm ( LTC\_Type \* *base*, const uint8\_t \* *plaintext*, uint8\_t \* *ciphertext*, uint32\_t *size*, const uint8\_t \* *iv*, uint32\_t *ivSize*, const uint8\_t \* *aad*, uint32\_t *aadSize*, const uint8\_t \* *key*, uint32\_t *keySize*, uint8\_t \* *tag*, uint32\_t *tagSize* )**

Encrypts AES and optionally tags using CCM block mode.

#### Parameters

|     |                   |                                                                         |
|-----|-------------------|-------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                             |
|     | <i>plaintext</i>  | Input plain text to encrypt                                             |
| out | <i>ciphertext</i> | Output cipher text.                                                     |
|     | <i>size</i>       | Size of input and output data in bytes. Zero means authentication only. |
|     | <i>iv</i>         | Nonce                                                                   |

|     |                |                                                                                  |
|-----|----------------|----------------------------------------------------------------------------------|
|     | <i>ivSize</i>  | Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.                |
|     | <i>aad</i>     | Input additional authentication data. Can be NULL if aadSize is zero.            |
|     | <i>aadSize</i> | Input size in bytes of AAD. Zero means data mode only (authentication skipped).  |
|     | <i>key</i>     | Input key to use for encryption                                                  |
|     | <i>keySize</i> | Size of the input key, in bytes. Must be 16, 24, or 32.                          |
| out | <i>tag</i>     | Generated output tag. Set to NULL to skip tag processing.                        |
|     | <i>tagSize</i> | Input size of the tag to generate, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16. |

Returns

Status from encrypt operation

**23.8.3.3.8 status\_t LTC\_AES\_DecryptTagCcm ( LTC\_Type \* *base*, const uint8\_t \* *ciphertext*, uint8\_t \* *plaintext*, uint32\_t *size*, const uint8\_t \* *iv*, uint32\_t *ivSize*, const uint8\_t \* *aad*, uint32\_t *aadSize*, const uint8\_t \* *key*, uint32\_t *keySize*, const uint8\_t \* *tag*, uint32\_t *tagSize* )**

Decrypts AES and optionally authenticates using CCM block mode.

Parameters

|     |                   |                                                                                 |
|-----|-------------------|---------------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                                     |
|     | <i>ciphertext</i> | Input cipher text to decrypt                                                    |
| out | <i>plaintext</i>  | Output plain text.                                                              |
|     | <i>size</i>       | Size of input and output data in bytes. Zero means authentication only.         |
|     | <i>iv</i>         | Nonce                                                                           |
|     | <i>ivSize</i>     | Length of the Nonce in bytes. Must be 7, 8, 9, 10, 11, 12, or 13.               |
|     | <i>aad</i>        | Input additional authentication data. Can be NULL if aadSize is zero.           |
|     | <i>aadSize</i>    | Input size in bytes of AAD. Zero means data mode only (authentication skipped). |

|  |                |                                                                                                                |
|--|----------------|----------------------------------------------------------------------------------------------------------------|
|  | <i>key</i>     | Input key to use for decryption                                                                                |
|  | <i>keySize</i> | Size of the input key, in bytes. Must be 16, 24, or 32.                                                        |
|  | <i>tag</i>     | Received tag. Set to NULL to skip tag processing.                                                              |
|  | <i>tagSize</i> | Input size of the received tag to compare with the computed tag, in bytes. Must be 4, 6, 8, 10, 12, 14, or 16. |

Returns

Status from decrypt operation

## 23.8.4 LTC HASH driver

### 23.8.4.1 Overview

This section describes the programming interface of the LTC HASH driver.

#### Data Structures

- struct `ltc_hash_ctx_t`  
*Storage type used to save hash context. [More...](#)*

#### Macros

- #define `LTC_HASH_CTX_SIZE` 29  
*LTC HASH Context size.*

#### Enumerations

- enum `ltc_hash_algo_t` {
   
`kLTC_XcbcMac` = 0,  
`kLTC_Cmac` }
- Supported cryptographic block cipher functions for HASH creation.*

#### Functions

- `status_t LTC_HASH_Init (LTC_Type *base, ltc_hash_ctx_t *ctx, ltc_hash_algo_t algo, const uint8_t *key, uint32_t keySize)`  
*Initialize HASH context.*
- `status_t LTC_HASH_Update (ltc_hash_ctx_t *ctx, const uint8_t *input, uint32_t inputSize)`  
*Add data to current HASH.*
- `status_t LTC_HASH_Finish (ltc_hash_ctx_t *ctx, uint8_t *output, uint32_t *outputSize)`  
*Finalize hashing.*
- `status_t LTC_HASH (LTC_Type *base, ltc_hash_algo_t algo, const uint8_t *input, uint32_t inputSize, const uint8_t *key, uint32_t keySize, uint8_t *output, uint32_t *outputSize)`  
*Create HASH on given data.*

### 23.8.4.2 Data Structure Documentation

#### 23.8.4.2.1 struct ltc\_hash\_ctx\_t

### 23.8.4.3 Macro Definition Documentation

#### 23.8.4.3.1 #define LTC\_HASH\_CTX\_SIZE 29

### 23.8.4.4 Enumeration Type Documentation

#### 23.8.4.4.1 enum ltc\_hash\_algo\_t

Enumerator

*kLTC\_XcbcMac* XCBC-MAC (AES engine)  
*kLTC\_Cmac* CMAC (AES engine)

### 23.8.4.5 Function Documentation

#### 23.8.4.5.1 status\_t LTC\_HASH\_Init ( LTC\_Type \* *base*, ltc\_hash\_ctx\_t \* *ctx*, ltc\_hash\_algo\_t *algo*, const uint8\_t \* *key*, uint32\_t *keySize* )

This function initialize the HASH. Key shall be supplied if the underlaying algorithm is AES XCBC-MAC or CMAC. Key shall be NULL if the underlaying algorithm is SHA.

For XCBC-MAC, the key length must be 16. For CMAC, the key length can be the AES key lengths supported by AES engine. For MDHA the key length argument is ignored.

Parameters

|     |                |                                                    |
|-----|----------------|----------------------------------------------------|
|     | <i>base</i>    | LTC peripheral base address                        |
| out | <i>ctx</i>     | Output hash context                                |
|     | <i>algo</i>    | Underlaying algorithm to use for hash computation. |
|     | <i>key</i>     | Input key (NULL if underlaying algorithm is SHA)   |
|     | <i>keySize</i> | Size of input key in bytes                         |

Returns

Status of initialization

#### 23.8.4.5.2 status\_t LTC\_HASH\_Update ( ltc\_hash\_ctx\_t \* *ctx*, const uint8\_t \* *input*, uint32\_t *inputSize* )

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data to be hashed.

Parameters

|                |                  |                             |
|----------------|------------------|-----------------------------|
| <i>in, out</i> | <i>ctx</i>       | HASH context                |
|                | <i>input</i>     | Input data                  |
|                | <i>inputSize</i> | Size of input data in bytes |

Returns

Status of the hash update operation

#### 23.8.4.5.3 status\_t LTC\_HASH\_Finish ( *Ltc\_hash\_ctx\_t \* ctx, uint8\_t \* output, uint32\_t \* outputSize* )

Outputs the final hash and erases the context.

Parameters

|                |                   |                                                               |
|----------------|-------------------|---------------------------------------------------------------|
| <i>in, out</i> | <i>ctx</i>        | Input hash context                                            |
| <i>out</i>     | <i>output</i>     | Output hash data                                              |
| <i>out</i>     | <i>outputSize</i> | Output parameter storing the size of the output hash in bytes |

Returns

Status of the hash finish operation

#### 23.8.4.5.4 status\_t LTC\_HASH ( *LTC\_Type \* base, Ltc\_hash\_algo\_t algo, const uint8\_t \* input, uint32\_t inputSize, const uint8\_t \* key, uint32\_t keySize, uint8\_t \* output, uint32\_t \* outputSize* )

Perform the full keyed HASH in one function call.

Parameters

|  |             |                                                 |
|--|-------------|-------------------------------------------------|
|  | <i>base</i> | LTC peripheral base address                     |
|  | <i>algo</i> | Block cipher algorithm to use for CMAC creation |

|     |                   |                                                               |
|-----|-------------------|---------------------------------------------------------------|
|     | <i>input</i>      | Input data                                                    |
|     | <i>inputSize</i>  | Size of input data in bytes                                   |
|     | <i>key</i>        | Input key                                                     |
|     | <i>keySize</i>    | Size of input key in bytes                                    |
| out | <i>output</i>     | Output hash data                                              |
| out | <i>outputSize</i> | Output parameter storing the size of the output hash in bytes |

## Returns

Status of the one call hash operation.

## 23.8.5 LTC PKHA driver

### 23.8.5.1 Overview

This section describes the programming interface of the LTC PKHA driver.

### Data Structures

- struct `ltc_pkha_ecc_point_t`  
*PKHA ECC point structure.* [More...](#)

### Enumerations

- enum `ltc_pkha_timing_t` {
   
`kLTC_PKHA_NoTimingEqualized` = 0U,
   
`kLTC_PKHA_TimingEqualized` = 1U }
   
*Use of timing equalized version of a PKHA function.*
- enum `ltc_pkha_f2m_t` {
   
`kLTC_PKHA_IntegerArith` = 0U,
   
`kLTC_PKHA_F2mArith` = 1U }
   
*Integer vs binary polynomial arithmetic selection.*
- enum `ltc_pkha_montgomery_form_t` {
   
`kLTC_PKHA_NormalValue` = 0U,
   
`kLTC_PKHA_MontgomeryFormat` = 1U }
   
*Montgomery or normal PKHA input format.*

### 23.8.5.2 Data Structure Documentation

#### 23.8.5.2.1 struct `ltc_pkha_ecc_point_t`

##### Data Fields

- `uint8_t * X`  
*X coordinate (affine)*
- `uint8_t * Y`  
*Y coordinate (affine)*

### 23.8.5.3 Enumeration Type Documentation

#### 23.8.5.3.1 enum `ltc_pkha_timing_t`

Enumerator

- `kLTC_PKHA_NoTimingEqualized`** Normal version of a PKHA operation.  
**`kLTC_PKHA_TimingEqualized`** Timing-equalized version of a PKHA operation.

### 23.8.5.3.2 enum ltc\_pkha\_f2m\_t

Enumerator

*kLTC\_PKHA\_IntegerArith* Use integer arithmetic.

*kLTC\_PKHA\_F2mArith* Use binary polynomial arithmetic.

### 23.8.5.3.3 enum ltc\_pkha\_montgomery\_form\_t

Enumerator

*kLTC\_PKHA\_NormalValue* PKHA number is normal integer.

*kLTC\_PKHA\_MontgomeryFormat* PKHA number is in montgomery format.

## 23.9 LTC Non-blocking eDMA APIs

### 23.9.1 Overview

This section describes the programming interface of the LTC eDMA Non Blocking functions

### Modules

- [LTC eDMA AES driver](#)
- [LTC eDMA DES driver](#)

### Data Structures

- struct [ltc\\_edma\\_handle\\_t](#)  
*LTC eDMA handle.* [More...](#)

### Typedefs

- [typedef void\(\\* ltc\\_edma\\_callback\\_t \)\(LTC\\_Type \\*base, ltc\\_edma\\_handle\\_t \\*handle, status\\_t status, void \\*userData\)](#)  
*LTC eDMA callback function.*
- [typedef status\\_t\(\\* ltc\\_edma\\_state\\_machine\\_t \)\(LTC\\_Type \\*base, ltc\\_edma\\_handle\\_t \\*handle\)](#)  
*LTC eDMA state machine function.*

### Functions

- [void LTC\\_CreateHandleEDMA \(LTC\\_Type \\*base, ltc\\_edma\\_handle\\_t \\*handle, ltc\\_edma\\_callback\\_t callback, void \\*userData, edma\\_handle\\_t \\*inputFifoEdmaHandle, edma\\_handle\\_t \\*outputFifoEdmaHandle\)](#)  
*Init the LTC eDMA handle which is used in transactional functions.*

### Driver version

- [#define FSL\\_LTC\\_EDMA\\_DRIVER\\_VERSION \(MAKE\\_VERSION\(2, 0, 15\)\)](#)  
*LTC EDMA driver version.*

### 23.9.2 Data Structure Documentation

#### 23.9.2.1 struct \_ltc\_edma\_handle

It is defined only for private usage inside LTC eDMA driver.

## Data Fields

- **ltc\_edma\_callback\_t callback**  
*Callback function.*
- **void \* userData**  
*LTC callback function parameter.*
- **edma\_handle\_t \* inputFifoEdmaHandle**  
*The eDMA TX channel used.*
- **edma\_handle\_t \* outputFifoEdmaHandle**  
*The eDMA RX channel used.*
- **ltc\_edma\_state\_machine\_t state\_machine**  
*State machine.*
- **uint32\_t state**  
*Internal state.*
- **const uint8\_t \* inData**  
*Input data.*
- **uint8\_t \* outData**  
*Output data.*
- **uint32\_t size**  
*Size of input and output data in bytes.*
- **uint32\_t modeReg**  
*LTC mode register.*
- **uint8\_t \* counter**  
*Input counter (updates on return)*
- **const uint8\_t \* key**  
*Input key to use for forward AES cipher.*
- **uint32\_t keySize**  
*Size of the input key, in bytes.*
- **uint8\_t \* counterlast**  
*Output cipher of last counter, for chained CTR calls.*
- **uint32\_t \* szLeft**  
*Output number of bytes in left unused in counterlast block.*
- **uint32\_t lastSize**  
*Last size.*

**Field Documentation**

- (1) `ltc_edma_callback_t ltc_edma_handle_t::callback`
- (2) `void* ltc_edma_handle_t::userData`
- (3) `edma_handle_t* ltc_edma_handle_t::inputFifoEdmaHandle`
- (4) `edma_handle_t* ltc_edma_handle_t::outputFifoEdmaHandle`
- (5) `ltc_edma_state_machine_t ltc_edma_handle_t::state_machine`
- (6) `uint32_t ltc_edma_handle_t::state`
- (7) `const uint8_t* ltc_edma_handle_t::inData`
- (8) `uint8_t* ltc_edma_handle_t::outData`
- (9) `uint32_t ltc_edma_handle_t::size`
- (10) `uint32_t ltc_edma_handle_t::modeReg`
- (11) `uint32_t ltc_edma_handle_t::keySize`

Must be 16, 24, or 32.

- (12) `uint8_t* ltc_edma_handle_t::counterlast`

NULL can be passed if chained calls are not used.

- (13) `uint32_t* ltc_edma_handle_t::szLeft`

NULL can be passed if chained calls are not used.

- (14) `uint32_t ltc_edma_handle_t::lastSize`

### **23.9.3 Macro Definition Documentation**

#### **23.9.3.1 #define FSL\_LTC\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 0, 15))**

Version 2.0.15.

### 23.9.4 Typedef Documentation

**23.9.4.1 `typedef void(* ltc_edma_callback_t)(LTC_Type *base, ltc_edma_handle_t *handle, status_t status, void *userData)`**

**23.9.4.2 `typedef status_t(* ltc_edma_state_machine_t)(LTC_Type *base, ltc_edma_handle_t *handle)`**

It is defined only for private usage inside LTC eDMA driver.

### 23.9.5 Function Documentation

**23.9.5.1 `void LTC_CreateHandleEDMA ( LTC_Type * base, ltc_edma_handle_t * handle, ltc_edma_callback_t callback, void * userData, edma_handle_t * inputFifoEdmaHandle, edma_handle_t * outputFifoEdmaHandle )`**

Parameters

|                                   |                                                     |
|-----------------------------------|-----------------------------------------------------|
| <i>base</i>                       | LTC module base address                             |
| <i>handle</i>                     | Pointer to <code>ltc_edma_handle_t</code> structure |
| <i>callback</i>                   | Callback function, NULL means no callback.          |
| <i>userData</i>                   | Callback function parameter.                        |
| <i>inputFifo-<br/>EdmaHandle</i>  | User requested eDMA handle for Input FIFO eDMA.     |
| <i>outputFifo-<br/>EdmaHandle</i> | User requested eDMA handle for Output FIFO eDMA.    |

### 23.9.6 LTC eDMA DES driver

This section describes the programming interface of the LTC eDMA DES driver.

## 23.9.7 LTC eDMA AES driver

### 23.9.7.1 Overview

This section describes the programming interface of the LTC eDMA AES driver.

#### Macros

- #define `LTC_AES_DecryptCtrEDMA`(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft) `LTC_AES_CryptCtrEDMA`(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft)  
*AES CTR decrypt is mapped to the AES CTR generic operation.*
- #define `LTC_AES_EncryptCtrEDMA`(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft) `LTC_AES_CryptCtrEDMA`(base, handle, input, output, size, counter, key, keySize, counterlast, szLeft)  
*AES CTR encrypt is mapped to the AES CTR generic operation.*

#### Functions

- `status_t LTC_AES_EncryptEcbEDMA` (LTC\_Type \*base, ltc\_edma\_handle\_t \*handle, const uint8\_t \*plaintext, uint8\_t \*ciphertext, uint32\_t size, const uint8\_t \*key, uint32\_t keySize)  
*Encrypts AES using the ECB block mode.*
- `status_t LTC_AES_DecryptEcbEDMA` (LTC\_Type \*base, ltc\_edma\_handle\_t \*handle, const uint8\_t \*ciphertext, uint8\_t \*plaintext, uint32\_t size, const uint8\_t \*key, uint32\_t keySize, `ltc_aes_key_t` keyType)  
*Decrypts AES using ECB block mode.*
- `status_t LTC_AES_EncryptCbcEDMA` (LTC\_Type \*base, ltc\_edma\_handle\_t \*handle, const uint8\_t \*plaintext, uint8\_t \*ciphertext, uint32\_t size, const uint8\_t iv[`LTC_AES_IV_SIZE`], const uint8\_t \*key, uint32\_t keySize)  
*Encrypts AES using CBC block mode.*
- `status_t LTC_AES_DecryptCbcEDMA` (LTC\_Type \*base, ltc\_edma\_handle\_t \*handle, const uint8\_t \*ciphertext, uint8\_t \*plaintext, uint32\_t size, const uint8\_t iv[`LTC_AES_IV_SIZE`], const uint8\_t \*key, uint32\_t keySize, `ltc_aes_key_t` keyType)  
*Decrypts AES using CBC block mode.*
- `status_t LTC_AES_CryptCtrEDMA` (LTC\_Type \*base, ltc\_edma\_handle\_t \*handle, const uint8\_t \*input, uint8\_t \*output, uint32\_t size, uint8\_t counter[`LTC_AES_BLOCK_SIZE`], const uint8\_t \*key, uint32\_t keySize, uint8\_t counterlast[`LTC_AES_BLOCK_SIZE`], uint32\_t \*szLeft)  
*Encrypts or decrypts AES using CTR block mode.*

### 23.9.7.2 Function Documentation

23.9.7.2.1 `status_t LTC_AES_EncryptEcbEDMA ( LTC_Type * base, ltc_edma_handle_t * handle, const uint8_t * plaintext, uint8_t * ciphertext, uint32_t size, const uint8_t * key, uint32_t keySize )`

Encrypts AES using the ECB block mode.

## Parameters

|     |                   |                                                                            |
|-----|-------------------|----------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                                |
|     | <i>handle</i>     | pointer to ltc_edma_handle_t structure which stores the transaction state. |
|     | <i>plaintext</i>  | Input plain text to encrypt                                                |
| out | <i>ciphertext</i> | Output cipher text                                                         |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes.      |
|     | <i>key</i>        | Input key to use for encryption                                            |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.                    |

## Returns

Status from encrypt operation

**23.9.7.2.2 status\_t LTC\_AES\_DecryptEcbEDMA ( LTC\_Type \* *base*, ltc\_edma\_handle\_t \* *handle*, const uint8\_t \* *ciphertext*, uint8\_t \* *plaintext*, uint32\_t *size*, const uint8\_t \* *key*, uint32\_t *keySize*, ltc\_aes\_key\_t *keyType* )**

Decrypts AES using ECB block mode.

## Parameters

|     |                   |                                                                                            |
|-----|-------------------|--------------------------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                                                |
|     | <i>handle</i>     | pointer to ltc_edma_handle_t structure which stores the transaction state.                 |
|     | <i>ciphertext</i> | Input cipher text to decrypt                                                               |
| out | <i>plaintext</i>  | Output plain text                                                                          |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes.                      |
|     | <i>key</i>        | Input key.                                                                                 |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.                                    |
|     | <i>keyType</i>    | Input type of the key (allows to directly load decrypt key for AES ECB decrypt operation.) |

## Returns

Status from decrypt operation

23.9.7.2.3 status\_t LTC\_AES\_EncryptCbcEDMA ( LTC\_Type \* *base*, Itc\_edma\_handle\_t \* *handle*, const uint8\_t \* *plaintext*, uint8\_t \* *ciphertext*, uint32\_t *size*, const uint8\_t *iv*[*LTC\_AES\_IV\_SIZE*], const uint8\_t \* *key*, uint32\_t *keySize* )

## Parameters

|     |                   |                                                                            |
|-----|-------------------|----------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                                |
|     | <i>handle</i>     | pointer to ltc_edma_handle_t structure which stores the transaction state. |
|     | <i>plaintext</i>  | Input plain text to encrypt                                                |
| out | <i>ciphertext</i> | Output cipher text                                                         |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes.      |
|     | <i>iv</i>         | Input initial vector to combine with the first input block.                |
|     | <i>key</i>        | Input key to use for encryption                                            |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.                    |

## Returns

Status from encrypt operation

```
23.9.7.2.4 status_t LTC_AES_DecryptCbcEDMA (LTC_Type * base, ltc_edma_handle_t * handle, const uint8_t * ciphertext, uint8_t * plaintext, uint32_t size, const uint8_t iv[LTC_AES_IV_SIZE], const uint8_t * key, uint32_t keySize, ltc_aes_key_t keyType)
```

## Parameters

|     |                   |                                                                                            |
|-----|-------------------|--------------------------------------------------------------------------------------------|
|     | <i>base</i>       | LTC peripheral base address                                                                |
|     | <i>handle</i>     | pointer to ltc_edma_handle_t structure which stores the transaction state.                 |
|     | <i>ciphertext</i> | Input cipher text to decrypt                                                               |
| out | <i>plaintext</i>  | Output plain text                                                                          |
|     | <i>size</i>       | Size of input and output data in bytes. Must be multiple of 16 bytes.                      |
|     | <i>iv</i>         | Input initial vector to combine with the first input block.                                |
|     | <i>key</i>        | Input key to use for decryption                                                            |
|     | <i>keySize</i>    | Size of the input key, in bytes. Must be 16, 24, or 32.                                    |
|     | <i>keyType</i>    | Input type of the key (allows to directly load decrypt key for AES CBC decrypt operation.) |

## Returns

Status from decrypt operation

```
23.9.7.2.5 status_t LTC_AES_CryptCtrEDMA (LTC_Type * base, ltc_edma_handle_t
* handle, const uint8_t * input, uint8_t * output, uint32_t size, uint8_t
counter[LTC_AES_BLOCK_SIZE], const uint8_t * key, uint32_t keySize, uint8_t
counterlast[LTC_AES_BLOCK_SIZE], uint32_t * szLeft)
```

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

#### Parameters

|        |                    |                                                                                                               |
|--------|--------------------|---------------------------------------------------------------------------------------------------------------|
|        | <i>base</i>        | LTC peripheral base address                                                                                   |
|        | <i>handle</i>      | pointer to ltc_edma_handle_t structure which stores the transaction state.                                    |
|        | <i>input</i>       | Input data for CTR block mode                                                                                 |
| out    | <i>output</i>      | Output data for CTR block mode                                                                                |
|        | <i>size</i>        | Size of input and output data in bytes                                                                        |
| in,out | <i>counter</i>     | Input counter (updates on return)                                                                             |
|        | <i>key</i>         | Input key to use for forward AES cipher                                                                       |
|        | <i>keySize</i>     | Size of the input key, in bytes. Must be 16, 24, or 32.                                                       |
| out    | <i>counterlast</i> | Output cipher of last counter, for chained CTR calls. NULL can be passed if chained calls are not used.       |
| out    | <i>szLeft</i>      | Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used. |

#### Returns

Status from encrypt operation

# Chapter 24

## MSMC: Multicore System Mode Controller

### 24.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Multicore System Mode Controller (MSMC) module of MCUXpresso SDK devices.

### 24.2 Typical use case

#### 24.2.1 Set Core 0 from RUN to VLPR mode

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/msmc

#### 24.2.2 Set Core 0 from VLPR/HSRUN to RUN mode

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/msmc

### 24.3 Typical use case

#### 24.3.1 Set Core 0 from RUN to HSRUN mode

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/msmc

#### 24.3.2 Enter wait or stop modes

SMC driver provides APIs to set MCU to different wait modes and stop modes. At the same time, there are pre-function and post-function for the modes setting. The pre-function and post-function are used for:

1. Disable/enable the interrupt through PRIMASK. In practise, there is such scenario: the application sets the wakeup interrupt and calls SMC function [SMC\\_SetPowerModeStop](#) to set MCU to STO-P mode, but the wakeup interrupt happens so quickly that the ISR completed before the function [SMC\\_SetPowerModeStop](#), as a result, the MCU enters STOP mode and never be wakeup by the interrupt. In this case, application could first disable interrupt through PRIMASK, then set the wakeup interrupt and enter STOP mode. After wakeup, the first thing is enable the interrupt through PRIMASK. The MCU could still be wakeup when disable interrupt through PRIMASK. The pre- and post- functions handle the PRIMASK inside.

```

SMC_PreEnterStopModes();

/* Enable the wakeup interrupt here. */

SMC_SetPowerModeStop(SMC0, kSMC_PartialStop);

SMC_PostExitStopModes();

```

For other use cases, please refer to the comments of the MSMC driver header file. Some example codes are also provided.

## Files

- file `fsl_msmc.h`

## Data Structures

- struct `smc_reset_pin_filter_config_t`  
*Reset pin filter configuration.* [More...](#)

## Enumerations

- enum `smc_power_mode_protection_t` {
 `kSMC_AllowPowerModeVlls` = SMC\_PMPROT\_AVLLS\_MASK,
 `kSMC_AllowPowerModeLls` = SMC\_PMPROT\_ALLS\_MASK,
 `kSMC_AllowPowerModeVlp` = SMC\_PMPROT\_AVLP\_MASK,
 `kSMC_AllowPowerModeHsrun` = SMC\_PMPROT\_AHSRUN\_MASK,
 `kSMC_AllowPowerModeAll` }
   
*Power Modes Protection.*
- enum `smc_power_state_t` {
 `kSMC_PowerStateRun` = 1U,
 `kSMC_PowerStateStop` = 1U << 1U,
 `kSMC_PowerStateVlpr` = 1U << 2U,
 `kSMC_PowerStateHsrun` = 1U << 7U }
   
*Power Modes in PMSTAT.*
- enum `smc_power_stop_entry_status_t` {
 `kSMC_PowerStopEntryAlt0` = 1U,
 `kSMC_PowerStopEntryAlt1` = 1U << 1,
 `kSMC_PowerStopEntryAlt2` = 1U << 2,
 `kSMC_PowerStopEntryAlt3` = 1U << 3,
 `kSMC_PowerStopEntryAlt4` = 1U << 4,
 `kSMC_PowerStopEntryAlt5` = 1U << 5 }
   
*Power Stop Entry Status in PMSTAT.*
- enum `smc_run_mode_t` {
 `kSMC_RunNormal` = 0U,
 `kSMC_RunVlpr` = 2U,
 `kSMC_Hsrun` = 3U }
   
*Run mode definition.*

- enum `smc_stop_mode_t` {
   
  `kSMC_StopNormal` = 0U,
   
  `kSMC_StopVlps` = 2U,
   
  `kSMC_StopLls` = 3U,
   
  `kSMC_StopVlls` = 4U }
   
    *Stop mode definition.*
- enum `smc_partial_stop_option_t` {
   
  `kSMC_PartialStop` = 0U,
   
  `kSMC_PartialStop1` = 1U,
   
  `kSMC_PartialStop2` = 2U,
   
  `kSMC_PartialStop3` = 3U }
   
    *Partial STOP option.*
- enum { `kStatus_SMC_StopAbort` = MAKE\_STATUS(kStatusGroup\_POWER, 0) }
   
    *SMC configuration status.*
- enum `smc_reset_source_t` {
   
  `kSMC_SourceWakeup` = SMC\_SRS\_WAKEUP\_MASK,
   
  `kSMC_SourcePor` = SMC\_SRS\_POR\_MASK,
   
  `kSMC_SourceLvd` = SMC\_SRS\_LVD\_MASK,
   
  `kSMC_SourceHvd` = SMC\_SRS\_HVD\_MASK,
   
  `kSMC_SourceWarm` = SMC\_SRS\_WARM\_MASK,
   
  `kSMC_SourceFatal` = SMC\_SRS\_FATAL\_MASK,
   
  `kSMC_SourceCore`,
   
  `kSMC_SourcePin` = SMC\_SRS\_PIN\_MASK,
   
  `kSMC_SourceMdm` = SMC\_SRS\_MDM\_MASK,
   
  `kSMC_SourceRstAck` = SMC\_SRS\_RSTACK\_MASK,
   
  `kSMC_SourceStopAck` = SMC\_SRS\_STOPACK\_MASK,
   
  `kSMC_SourceScg` = SMC\_SRS\_SCG\_MASK,
   
  `kSMC_SourceWdog` = SMC\_SRS\_WDOG\_MASK,
   
  `kSMC_SourceSoftware` = SMC\_SRS\_SW\_MASK,
   
  `kSMC_SourceLockup` = SMC\_SRS\_LOCKUP\_MASK,
   
  `kSMC_SourceJtag` = SMC\_SRS\_JTAG\_MASK }
   
    *System Reset Source Name definitions.*
- enum `smc_interrupt_enable_t` {
   
  `kSMC_IntNone` = 0U,
   
  `kSMC_IntPin` = SMC\_SRIE\_PIN\_MASK,
   
  `kSMC_IntMdm` = SMC\_SRIE\_MDM\_MASK,
   
  `kSMC_IntStopAck` = SMC\_SRIE\_STOPACK\_MASK,
   
  `kSMC_IntWdog` = SMC\_SRIE\_WDOG\_MASK,
   
  `kSMC_IntSoftware` = SMC\_SRIE\_SW\_MASK,
   
  `kSMC_IntLockup` = SMC\_SRIE\_LOCKUP\_MASK,
   
  `kSMC_IntAll` }
   
    *System reset interrupt enable bit definitions.*

## Driver version

- #define `FSL_MSMC_DRIVER_VERSION` (MAKE\_VERSION(2, 1, 2))

*MSMC driver version.*

## System mode controller APIs

- static void [SMC\\_SetPowerModeProtection](#) (SMC\_Type \*base, uint8\_t allowedModes)  
*Configures all power mode protection settings.*
- static [smc\\_power\\_state\\_t SMC\\_GetPowerModeState](#) (SMC\_Type \*base)  
*Gets the current power mode status.*
- static void [SMC\\_PreEnterStopModes](#) (void)  
*Prepare to enter stop modes.*
- static void [SMC\\_PostExitStopModes](#) (void)  
*Recovering after wake up from stop modes.*
- static void [SMC\\_PreEnterWaitModes](#) (void)  
*Prepare to enter wait modes.*
- static void [SMC\\_PostExitWaitModes](#) (void)  
*Recovering after wake up from stop modes.*
- [status\\_t SMC\\_SetPowerModeRun](#) (SMC\_Type \*base)  
*Configure the system to RUN power mode.*
- [status\\_t SMC\\_SetPowerModeHsrn](#) (SMC\_Type \*base)  
*Configure the system to HSRUN power mode.*
- [status\\_t SMC\\_SetPowerModeWait](#) (SMC\_Type \*base)  
*Configure the system to WAIT power mode.*
- [status\\_t SMC\\_SetPowerModeStop](#) (SMC\_Type \*base, [smc\\_partial\\_stop\\_option\\_t](#) option)  
*Configure the system to Stop power mode.*
- [status\\_t SMC\\_SetPowerModeVlpr](#) (SMC\_Type \*base)  
*Configure the system to VLPR power mode.*
- [status\\_t SMC\\_SetPowerModeVlpw](#) (SMC\_Type \*base)  
*Configure the system to VLPW power mode.*
- [status\\_t SMC\\_SetPowerModeVlps](#) (SMC\_Type \*base)  
*Configure the system to VLPS power mode.*
- [status\\_t SMC\\_SetPowerModeLls](#) (SMC\_Type \*base)  
*Configure the system to LLS power mode.*
- [status\\_t SMC\\_SetPowerModeVlls](#) (SMC\_Type \*base)  
*Configure the system to VLLS power mode.*
- static uint32\_t [SMC\\_GetPreviousResetSources](#) (SMC\_Type \*base)  
*Gets the reset source status which caused a previous reset.*
- static uint32\_t [SMC\\_GetStickyResetSources](#) (SMC\_Type \*base)  
*Gets the sticky reset source status.*
- static void [SMC\\_ClearStickyResetSources](#) (SMC\_Type \*base, uint32\_t sourceMasks)  
*Clears the sticky reset source status.*
- void [SMC\\_ConfigureResetPinFilter](#) (SMC\_Type \*base, const [smc\\_reset\\_pin\\_filter\\_config\\_t](#) \*config)  
*Configures the reset pin filter.*
- static void [SMC\\_SetSystemResetInterruptConfig](#) (SMC\_Type \*base, uint32\_t intMask)  
*Sets the system reset interrupt configuration.*
- static uint32\_t [SMC\\_GetResetInterruptSourcesStatus](#) (SMC\_Type \*base)  
*Gets the source status of the system reset interrupt.*
- static void [SMC\\_ClearResetInterruptSourcesStatus](#) (SMC\_Type \*base, uint32\_t intMask)  
*Clears the source status of the system reset interrupt.*
- static uint32\_t [SMC\\_GetBootOptionConfig](#) (SMC\_Type \*base)  
*Gets the boot option configuration.*

## 24.4 Data Structure Documentation

### 24.4.1 struct smc\_reset\_pin\_filter\_config\_t

#### Data Fields

- `uint8_t slowClockFilterCount`  
*Reset pin bus clock filter width from 1 to 32 slow clock cycles.*
- `bool enableFilter`  
*Reset pin filter enable/disable.*

#### Field Documentation

(1) `uint8_t smc_reset_pin_filter_config_t::slowClockFilterCount`

(2) `bool smc_reset_pin_filter_config_t::enableFilter`

## 24.5 Macro Definition Documentation

24.5.1 `#define FSL_MSMC_DRIVER_VERSION (MAKE_VERSION(2, 1, 2))`

## 24.6 Enumeration Type Documentation

### 24.6.1 enum smc\_power\_mode\_protection\_t

Enumerator

`kSMC_AllowPowerModeVlls` Allow Very-Low-Leakage Stop Mode.

`kSMC_AllowPowerModeLls` Allow Low-Leakage Stop Mode.

`kSMC_AllowPowerModeVlp` Allow Very-Low-Power Mode.

`kSMC_AllowPowerModeHsrun` Allow High Speed Run mode.

`kSMC_AllowPowerModeAll` Allow all power mode.

### 24.6.2 enum smc\_power\_state\_t

Enumerator

`kSMC_PowerStateRun` 0000\_0001 - Current power mode is RUN

`kSMC_PowerStateStop` 0000\_0010 - Current power mode is any STOP mode

`kSMC_PowerStateVlpr` 0000\_0100 - Current power mode is VLPR

`kSMC_PowerStateHsrun` 1000\_0000 - Current power mode is HSRUN

### 24.6.3 enum smc\_power\_stop\_entry\_status\_t

Enumerator

- kSMC\_PowerStopEntryAlt0* Indicates a Stop mode entry since this field was last cleared.
- kSMC\_PowerStopEntryAlt1* Indicates the system bus masters acknowledged the Stop mode entry.
- kSMC\_PowerStopEntryAlt2* Indicates the system clock peripherals acknowledged the Stop mode entry.
- kSMC\_PowerStopEntryAlt3* Indicates the bus clock peripherals acknowledged the Stop mode entry.
- kSMC\_PowerStopEntryAlt4* Indicates the slow clock peripherals acknowledged the Stop mode entry.
- kSMC\_PowerStopEntryAlt5* Indicates Stop mode entry completed.

### 24.6.4 enum smc\_run\_mode\_t

Enumerator

- kSMC\_RunNormal* normal RUN mode.
- kSMC\_RunVlpr* Very-Low-Power RUN mode.
- kSMC\_Hsrun* High Speed Run mode (HSRUN).

### 24.6.5 enum smc\_stop\_mode\_t

Enumerator

- kSMC\_StopNormal* Normal STOP mode.
- kSMC\_StopVlps* Very-Low-Power STOP mode.
- kSMC\_StopLls* Low-Leakage Stop mode.
- kSMC\_StopVlls* Very-Low-Leakage Stop mode.

### 24.6.6 enum smc\_partial\_stop\_option\_t

Enumerator

- kSMC\_PartialStop* STOP - Normal Stop mode.
- kSMC\_PartialStop1* Partial Stop with both system and bus clocks disabled.
- kSMC\_PartialStop2* Partial Stop with system clock disabled and bus clock enabled.
- kSMC\_PartialStop3* Partial Stop with system clock enabled and bus clock disabled.

## 24.6.7 anonymous enum

Enumerator

*kStatus\_SMC\_StopAbort* Entering Stop mode is abort.

## 24.6.8 enum smc\_reset\_source\_t

Enumerator

*kSMC\_SourceWakeup* Very low-leakage wakeup reset.

*kSMC\_SourcePor* Power on reset.

*kSMC\_SourceLvd* Low-voltage detect reset.

*kSMC\_SourceHvd* High-voltage detect reset.

*kSMC\_SourceWarm* Warm reset. Warm Reset flag will assert if any of the system reset sources in this register assert (SRS[31:8])

*kSMC\_SourceFatal* Fatal reset.

*kSMC\_SourceCore* Software reset that only reset the core, NOT a sticky system reset source.

*kSMC\_SourcePin* RESET\_B pin reset.

*kSMC\_SourceMdm* MDM reset.

*kSMC\_SourceRstAck* Reset Controller timeout reset.

*kSMC\_SourceStopAck* Stop timeout reset.

*kSMC\_SourceScg* SCG loss of lock or loss of clock.

*kSMC\_SourceWdog* Watchdog reset.

*kSMC\_SourceSoftware* Software reset.

*kSMC\_SourceLockup* Lockup reset. Core lockup or exception.

*kSMC\_SourceJtag* JTAG system reset.

## 24.6.9 enum smc\_interrupt\_enable\_t

Enumerator

*kSMC\_IntNone* No interrupt enabled.

*kSMC\_IntPin* Pin reset interrupt.

*kSMC\_IntMdm* MDM reset interrupt.

*kSMC\_IntStopAck* Stop timeout reset interrupt.

*kSMC\_IntWdog* Watchdog interrupt.

*kSMC\_IntSoftware* Software reset interrupts.

*kSMC\_IntLockup* Lock up interrupt.

*kSMC\_IntAll* All system reset interrupts.

## 24.7 Function Documentation

### 24.7.1 static void SMC\_SetPowerModeProtection ( **SMC\_Type** \* *base*, **uint8\_t** *allowedModes* ) [inline], [static]

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the smc\_power\_mode\_protection\_t. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map, for example, to allow LLS and VLLS, use SMC\_SetPowerModeProtection(kSMC\_AllowPowerModeLls | kSMC\_AllowPowerModeVlls). 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. |

### 24.7.2 static smc\_power\_state\_t SMC\_GetPowerModeState ( **SMC\_Type** \* *base* ) [inline], [static]

This function returns the current power mode stat. Once application switches the power mode, it should always check the stat to check whether it runs into the specified mode or not. An 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 stat.

Parameters

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

Returns

Current power mode status.

### 24.7.3 static void SMC\_PreEnterStopModes ( void ) [inline], [static]

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

#### 24.7.4 static void SMC\_PostExitStopModes ( void ) [inline], [static]

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

#### 24.7.5 static void SMC\_PreEnterWaitModes ( void ) [inline], [static]

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

#### 24.7.6 static void SMC\_PostExitWaitModes ( void ) [inline], [static]

This function should be called after wake up from WAIT/VLPW modes. It is used together with [SMC\\_PreEnterWaitModes](#).

#### 24.7.7 status\_t SMC\_SetPowerModeRun ( SMC\_Type \* *base* )

Parameters

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

Returns

SMC configuration error code.

#### 24.7.8 status\_t SMC\_SetPowerModeHsrun ( SMC\_Type \* *base* )

Parameters

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

Returns

SMC configuration error code.

#### 24.7.9 status\_t SMC\_SetPowerModeWait ( SMC\_Type \* *base* )

Parameters

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

Returns

SMC configuration error code.

#### 24.7.10 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.

#### 24.7.11 status\_t SMC\_SetPowerModeVlpr ( SMC\_Type \* *base* )

Parameters

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

Returns

SMC configuration error code.

#### 24.7.12 status\_t SMC\_SetPowerModeVlpw ( SMC\_Type \* *base* )

Parameters

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

Returns

SMC configuration error code.

#### **24.7.13 status\_t SMC\_SetPowerModeVips ( SMC\_Type \* *base* )**

Parameters

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

Returns

SMC configuration error code.

#### **24.7.14 status\_t SMC\_SetPowerModeLIs ( SMC\_Type \* *base* )**

Parameters

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

Returns

SMC configuration error code.

#### **24.7.15 status\_t SMC\_SetPowerModeVlls ( SMC\_Type \* *base* )**

Parameters

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

Returns

SMC configuration error code.

#### 24.7.16 static uint32\_t SMC\_GetPreviousResetSources ( SMC\_Type \* *base* ) [inline], [static]

This function gets the current reset source status. Use source masks defined in the smc\_reset\_source\_t to get the desired source status.

Example: To get all reset source statuses.

```
resetStatus = SMC_GetPreviousResetSources(SMC0) & kSMC_SourceAll;
```

Example: To test whether the MCU is reset using Watchdog.

```
uint32_t resetStatus;

resetStatus = SMC_GetPreviousResetSources(SMC0) &
 kSMC_SourceWdog;
```

Example: To test multiple reset sources.

```
uint32_t resetStatus;

resetStatus = SMC_GetPreviousResetSources(SMC0) & (
 kSMC_SourceWdog | kSMC_SourcePin);
```

Parameters

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

Returns

All reset source status bit map.

#### 24.7.17 static uint32\_t SMC\_GetStickyResetSources ( SMC\_Type \* *base* ) [inline], [static]

This function gets the current reset source status that has not been cleared by software for some specific source.

Example: To get all reset source statuses.

```
uint32_t resetStatus;

resetStatus = SMC_GetStickyResetSources(SMC0) & kSMC_SourceAll;
```

Example, To test whether the MCU is reset using Watchdog.

```
uint32_t resetStatus;

resetStatus = SMC_GetStickyResetSources(SMC0) &
 kSMC_SourceWdog;
```

Example To test multiple reset sources.

```
uint32_t resetStatus;

resetStatus = SMC_GetStickyResetSources(SMC0) & (
 kSMC_SourceWdog | kSMC_SourcePin);
```

Parameters

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

Returns

All reset source status bit map.

#### 24.7.18 static void SMC\_ClearStickyResetSources ( SMC\_Type \* *base*, uint32\_t *sourceMasks* ) [inline], [static]

This function clears the sticky system reset flags indicated by source masks.

Example: Clears multiple reset sources.

```
SMC_ClearStickyResetSources(SMC0, (kSMC_SourceWdog |
 kSMC_SourcePin));
```

Parameters

|                    |                              |
|--------------------|------------------------------|
| <i>base</i>        | SMC peripheral base address. |
| <i>sourceMasks</i> | reset source status bit map  |

#### 24.7.19 void SMC\_ConfigureResetPinFilter ( SMC\_Type \* *base*, const smc\_reset\_pin\_filter\_config\_t \* *config* )

This function sets the reset pin filter including the enablement/disablement and filter width.

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | SMC peripheral base address.            |
| <i>config</i> | Pointer to the configuration structure. |

#### 24.7.20 static void SMC\_SetSystemResetInterruptConfig ( SMC\_Type \* *base*, uint32\_t *intMask* ) [inline], [static]

For a graceful shut down, the MSMC 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. The interrupts are passed in as bit mask. See smc\_interrupt\_enable\_t for details. For example, to delay a reset after the WDOG timeout or PIN reset occurs, configure as follows: SMC\_SetSystemResetInterruptConfig(SMC0, (kSMC\_IntWdog | kSMC\_IntPin));

Parameters

|                |                                                                                            |
|----------------|--------------------------------------------------------------------------------------------|
| <i>base</i>    | SMC peripheral base address.                                                               |
| <i>intMask</i> | Bit mask of the system reset interrupts to enable. See smc_interrupt_enable_t for details. |

#### 24.7.21 static uint32\_t SMC\_GetResetInterruptSourcesStatus ( SMC\_Type \* *base* ) [inline], [static]

This function gets the source status of the reset interrupt. Use source masks defined in the smc\_interrupt\_enable\_t to get the desired source status.

Example: To get all reset interrupt source statuses.

```
uint32_t interruptStatus;

interruptStatus = SMC_GetResetInterruptSourcesStatus(SMC0) &
 kSMC_IntAll;
```

Example: To test whether the reset interrupt of Watchdog is pending.

```
uint32_t interruptStatus;

interruptStatus = SMC_GetResetInterruptSourcesStatus(SMC0) &
 kSMC_IntWdog;
```

Example: To test multiple reset interrupt sources.

```
uint32_t interruptStatus;

interruptStatus = SMC_GetResetInterruptSourcesStatus(SMC0) & (
 kSMC_IntWdog | kSMC_IntPin);
```

Parameters

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

Returns

All reset interrupt source status bit map.

#### 24.7.22 static void SMC\_ClearResetInterruptSourcesStatus ( SMC\_Type \* *base*, uint32\_t *intMask* ) [inline], [static]

This function clears the source status of the reset interrupt. Use source masks defined in the smc\_interrupt\_enable\_t to get the desired source status.

Example: To clear all reset interrupt source statuses.

```
uint32_t interruptStatus;
MMC_ClearResetInterruptSourcesStatus(SMC0, kSMC_IntAll);
```

Example, To clear the reset interrupt of Watchdog.

```
uint32_t interruptStatus;
SMC_ClearResetInterruptSourcesStatus(SMC0,
 kSMC_IntWdog);
```

Example, To clear multiple reset interrupt sources status.

```
uint32_t interruptStatus;
SMC_ClearResetInterruptSourcesStatus(SMC0, (
 kSMC_IntWdog | kSMC_IntPin));
```

Parameters

|                |                                                     |
|----------------|-----------------------------------------------------|
| <i>base</i>    | SMC peripheral base address.                        |
| <i>intMask</i> | All reset interrupt source status bit map to clear. |

#### 24.7.23 static uint32\_t SMC\_GetBootOptionConfig ( SMC\_Type \* *base* ) [inline], [static]

This function gets the boot option configuration of MSMC.

### Parameters

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

### Returns

The boot option configuration. 1 means boot option enabled. 0 means not.

# Chapter 25

## MU: Messaging Unit

### 25.1 Overview

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

### 25.2 Function description

The MU driver provides these functions:

- Functions to initialize the MU module.
- Functions to send and receive messages.
- Functions for MU flags for both MU sides.
- Functions for status flags and interrupts.
- Other miscellaneous functions.

#### 25.2.1 MU initialization

The function [MU\\_Init\(\)](#) initializes the MU module and enables the MU clock. It should be called before any other MU functions.

The function [MU\\_Deinit\(\)](#) deinitializes the MU module and disables the MU clock. No MU functions can be called after this function.

#### 25.2.2 MU message

The MU message must be sent when the transmit register is empty. The MU driver provides blocking API and non-blocking API to send message.

The [MU\\_SendMsgNonBlocking\(\)](#) function writes a message to the MU transmit register without checking the transmit register status. The upper layer should check that the transmit register is empty before calling this function. This function can be used in the ISR for better performance.

The [MU\\_SendMsg\(\)](#) function is a blocking function. It waits until the transmit register is empty and sends the message.

Correspondingly, there are blocking and non-blocking APIs for receiving a message. The [MU\\_ReadMsgNonBlocking\(\)](#) function is a non-blocking API. The [MU\\_ReadMsg\(\)](#) function is the blocking API.

### 25.2.3 MU flags

The MU driver provides 3-bit general purpose flags. When the flags are set on one side, they are reflected on the other side.

The MU flags must be set when the previous flags have been updated to the other side. The MU driver provides a non-blocking function and a blocking function. The blocking function [MU\\_SetFlags\(\)](#) waits until previous flags have been updated to the other side and then sets flags. The non-blocking function sets the flags directly. Ensure that the kMU\_FlagsUpdatingFlag is not pending before calling this function.

The function [MU\\_GetFlags\(\)](#) gets the MU flags on the current side.

### 25.2.4 Status and interrupt

The function [MU\\_GetStatusFlags\(\)](#) returns all MU status flags. Use the `_mu_status_flags` to check for specific flags, for example, to check RX0 and RX1 register full, use the following code:

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/mu. The receive full flags are cleared automatically after messages are read out. The transmit empty flags are cleared automatically after new messages are written to the transmit register. The general purpose interrupt flags must be cleared manually using the function [MU\\_ClearStatusFlags\(\)](#).

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/mu. To enable or disable a specific interrupt, use [MU\\_EnableInterrupts\(\)](#) and [MU\\_DisableInterrupts\(\)](#) functions. The interrupts to enable or disable should be passed in as a bit mask of the `_mu_interrupt_enable`.

The [MU\\_TriggerInterrupts\(\)](#) function triggers general purpose interrupts and NMI to the other core. The interrupts to trigger are passed in as a bit mask of the `_mu_interrupt_trigger`. If previously triggered interrupts have not been processed by the other side, this function returns an error.

### 25.2.5 MU misc functions

The [MU\\_BootCoreB\(\)](#) and [MU\\_HoldCoreBReset\(\)](#) functions should only be used from A side. They are used to boot the core B or to hold core B in reset.

The [MU\\_ResetBothSides\(\)](#) function resets MU at both A and B sides. However, only the A side can call this function.

If a core enters stop mode, the platform clock of this core is disabled by default. The function [MU\\_SetClockOnOtherCoreEnable\(\)](#) forces the other core's platform clock to remain enabled even after that core has entered a stop mode. In this case, the other core's platform clock keeps running until the current core enters stop mode too.

Function [MU\\_GetOtherCorePowerMode\(\)](#) gets the power mode of the other core.

## Enumerations

- enum `_mu_status_flags` {
   
kMU\_Tx0EmptyFlag = (1U << (MU\_SR\_TEEn\_SHIFT + 3U)),
   
kMU\_Tx1EmptyFlag = (1U << (MU\_SR\_TEEn\_SHIFT + 2U)),
   
kMU\_Tx2EmptyFlag = (1U << (MU\_SR\_TEEn\_SHIFT + 1U)),
   
kMU\_Tx3EmptyFlag = (1U << (MU\_SR\_TEEn\_SHIFT + 0U)),
   
kMU\_Rx0FullFlag = (1U << (MU\_SR\_RFn\_SHIFT + 3U)),
   
kMU\_Rx1FullFlag = (1U << (MU\_SR\_RFn\_SHIFT + 2U)),
   
kMU\_Rx2FullFlag = (1U << (MU\_SR\_RFn\_SHIFT + 1U)),
   
kMU\_Rx3FullFlag = (1U << (MU\_SR\_RFn\_SHIFT + 0U)),
   
kMU\_GenInt0Flag = (1U << (MU\_SR\_GIPn\_SHIFT + 3U)),
   
kMU\_GenInt1Flag = (1U << (MU\_SR\_GIPn\_SHIFT + 2U)),
   
kMU\_GenInt2Flag = (1U << (MU\_SR\_GIPn\_SHIFT + 1U)),
   
kMU\_GenInt3Flag = (1U << (MU\_SR\_GIPn\_SHIFT + 0U)),
   
kMU\_EventPendingFlag = MU\_SR\_EP\_MASK,
   
kMU\_FlagsUpdatingFlag = MU\_SR\_FUP\_MASK }

*MU status flags.*

- enum `_mu_interrupt_enable` {
   
kMU\_Tx0EmptyInterruptEnable = (1U << (MU\_CR\_TIEEn\_SHIFT + 3U)),
   
kMU\_Tx1EmptyInterruptEnable = (1U << (MU\_CR\_TIEEn\_SHIFT + 2U)),
   
kMU\_Tx2EmptyInterruptEnable = (1U << (MU\_CR\_TIEEn\_SHIFT + 1U)),
   
kMU\_Tx3EmptyInterruptEnable = (1U << (MU\_CR\_TIEEn\_SHIFT + 0U)),
   
kMU\_Rx0FullInterruptEnable = (1U << (MU\_CR\_RIEn\_SHIFT + 3U)),
   
kMU\_Rx1FullInterruptEnable = (1U << (MU\_CR\_RIEn\_SHIFT + 2U)),
   
kMU\_Rx2FullInterruptEnable = (1U << (MU\_CR\_RIEn\_SHIFT + 1U)),
   
kMU\_Rx3FullInterruptEnable = (1U << (MU\_CR\_RIEn\_SHIFT + 0U)),
   
kMU\_GenInt0InterruptEnable = (int)(1U << (MU\_CR\_GIEn\_SHIFT + 3U)),
   
kMU\_GenInt1InterruptEnable = (1U << (MU\_CR\_GIEn\_SHIFT + 2U)),
   
kMU\_GenInt2InterruptEnable = (1U << (MU\_CR\_GIEn\_SHIFT + 1U)),
   
kMU\_GenInt3InterruptEnable = (1U << (MU\_CR\_GIEn\_SHIFT + 0U)) }

*MU interrupt source to enable.*

- enum `_mu_interrupt_trigger` {
   
kMU\_NmiInterruptTrigger = MU\_CR\_NMI\_MASK,
   
kMU\_GenInt0InterruptTrigger = (1U << (MU\_CR\_GIRn\_SHIFT + 3U)),
   
kMU\_GenInt1InterruptTrigger = (1U << (MU\_CR\_GIRn\_SHIFT + 2U)),
   
kMU\_GenInt2InterruptTrigger = (1U << (MU\_CR\_GIRn\_SHIFT + 1U)),
   
kMU\_GenInt3InterruptTrigger = (1U << (MU\_CR\_GIRn\_SHIFT + 0U)) }

*MU interrupt that could be triggered to the other core.*

- enum `mu_msg_reg_index_t`
  
*MU message register.*

## Driver version

- #define `FSL_MU_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 2)`)
   
*MU driver version.*

## MU initialization.

- void **MU\_Init** (MU\_Type \*base)  
*Initializes the MU module.*
- void **MU\_Deinit** (MU\_Type \*base)  
*De-initializes the MU module.*

## MU Message

- static void **MU\_SendMsgNonBlocking** (MU\_Type \*base, uint32\_t regIndex, uint32\_t msg)  
*Writes a message to the TX register.*
- void **MU\_SendMsg** (MU\_Type \*base, uint32\_t regIndex, uint32\_t msg)  
*Blocks to send a message.*
- static uint32\_t **MU\_ReceiveMsgNonBlocking** (MU\_Type \*base, uint32\_t regIndex)  
*Reads a message from the RX register.*
- uint32\_t **MU\_ReceiveMsg** (MU\_Type \*base, uint32\_t regIndex)  
*Blocks to receive a message.*

## MU Flags

- static void **MU\_SetFlagsNonBlocking** (MU\_Type \*base, uint32\_t flags)  
*Sets the 3-bit MU flags reflect on the other MU side.*
- void **MU\_SetFlags** (MU\_Type \*base, uint32\_t flags)  
*Blocks setting the 3-bit MU flags reflect on the other MU side.*
- static uint32\_t **MU\_GetFlags** (MU\_Type \*base)  
*Gets the current value of the 3-bit MU flags set by the other side.*

## Status and Interrupt.

- static uint32\_t **MU\_GetStatusFlags** (MU\_Type \*base)  
*Gets the MU status flags.*
- static uint32\_t **MU\_GetInterruptsPending** (MU\_Type \*base)  
*Gets the MU IRQ pending status.*
- static void **MU\_ClearStatusFlags** (MU\_Type \*base, uint32\_t mask)  
*Clears the specific MU status flags.*
- static void **MU\_EnableInterrupts** (MU\_Type \*base, uint32\_t mask)  
*Enables the specific MU interrupts.*
- static void **MU\_DisableInterrupts** (MU\_Type \*base, uint32\_t mask)  
*Disables the specific MU interrupts.*
- status\_t **MU\_TriggerInterrupts** (MU\_Type \*base, uint32\_t mask)  
*Triggers interrupts to the other core.*
- static void **MU\_ClearNmi** (MU\_Type \*base)  
*Clear non-maskable interrupt (NMI) sent by the other core.*

## MU misc functions

- void **MU\_BootCoreB** (MU\_Type \*base, mu\_core\_boot\_mode\_t mode)  
*Boots the core at B side.*
- static void **MU\_HoldCoreBReset** (MU\_Type \*base)  
*Holds the core reset of B side.*
- void **MU\_BootOtherCore** (MU\_Type \*base, mu\_core\_boot\_mode\_t mode)

- static void [MU\\_HoldOtherCoreReset](#) (MU\_Type \*base)  
*Holds the other core reset.*
- static void [MU\\_ResetBothSides](#) (MU\_Type \*base)  
*Resets the MU for both A side and B side.*
- void [MU\\_HardwareResetOtherCore](#) (MU\_Type \*base, bool waitReset, bool holdReset, mu\_core\_boot\_mode\_t bootMode)  
*Hardware reset the other core.*
- static void [MU\\_SetClockOnOtherCoreEnable](#) (MU\_Type \*base, bool enable)  
*Enables or disables the clock on the other core.*
- static mu\_power\_mode\_t [MU\\_GetOtherCorePowerMode](#) (MU\_Type \*base)  
*Gets the power mode of the other core.*

## 25.3 Macro Definition Documentation

### 25.3.1 #define FSL\_MU\_DRIVER\_VERSION (MAKE\_VERSION(2, 1, 2))

## 25.4 Enumeration Type Documentation

### 25.4.1 enum \_mu\_status\_flags

Enumerator

- kMU\_Tx0EmptyFlag* TX0 empty.
- kMU\_Tx1EmptyFlag* TX1 empty.
- kMU\_Tx2EmptyFlag* TX2 empty.
- kMU\_Tx3EmptyFlag* TX3 empty.
- kMU\_Rx0FullFlag* RX0 full.
- kMU\_Rx1FullFlag* RX1 full.
- kMU\_Rx2FullFlag* RX2 full.
- kMU\_Rx3FullFlag* RX3 full.
- kMU\_GenInt0Flag* General purpose interrupt 0 pending.
- kMU\_GenInt1Flag* General purpose interrupt 1 pending.
- kMU\_GenInt2Flag* General purpose interrupt 2 pending.
- kMU\_GenInt3Flag* General purpose interrupt 3 pending.
- kMU\_EventPendingFlag* MU event pending.
- kMU\_FlagsUpdatingFlag* MU flags update is on-going.

### 25.4.2 enum \_mu\_interrupt\_enable

Enumerator

- kMU\_Tx0EmptyInterruptEnable* TX0 empty.
- kMU\_Tx1EmptyInterruptEnable* TX1 empty.
- kMU\_Tx2EmptyInterruptEnable* TX2 empty.
- kMU\_Tx3EmptyInterruptEnable* TX3 empty.

*kMU\_Rx0FullInterruptEnable* RX0 full.  
*kMU\_Rx1FullInterruptEnable* RX1 full.  
*kMU\_Rx2FullInterruptEnable* RX2 full.  
*kMU\_Rx3FullInterruptEnable* RX3 full.  
*kMU\_GenInt0InterruptEnable* General purpose interrupt 0.  
*kMU\_GenInt1InterruptEnable* General purpose interrupt 1.  
*kMU\_GenInt2InterruptEnable* General purpose interrupt 2.  
*kMU\_GenInt3InterruptEnable* General purpose interrupt 3.

### 25.4.3 enum \_mu\_interrupt\_trigger

Enumerator

*kMU\_NmiInterruptTrigger* NMI interrupt.  
*kMU\_GenInt0InterruptTrigger* General purpose interrupt 0.  
*kMU\_GenInt1InterruptTrigger* General purpose interrupt 1.  
*kMU\_GenInt2InterruptTrigger* General purpose interrupt 2.  
*kMU\_GenInt3InterruptTrigger* General purpose interrupt 3.

## 25.5 Function Documentation

### 25.5.1 void MU\_Init ( MU\_Type \* *base* )

This function enables the MU clock only.

Parameters

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

### 25.5.2 void MU\_Deinit ( MU\_Type \* *base* )

This function disables the MU clock only.

Parameters

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

### 25.5.3 static void MU\_SendMsgNonBlocking ( MU\_Type \* *base*, uint32\_t *regIndex*, uint32\_t *msg* ) [inline], [static]

This function writes a message to the specific TX register. It does not check whether the TX register is empty or not. The upper layer should make sure the TX register is empty before calling this function. This

function can be used in ISR for better performance.

```
* while (!(kMU_Tx0EmptyFlag & MU_GetStatusFlags(base))) { } Wait for TX0
 register empty.
* MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG_VAL); Write message to the TX0
 register.
*
```

Parameters

|                 |                                                             |
|-----------------|-------------------------------------------------------------|
| <i>base</i>     | MU peripheral base address.                                 |
| <i>regIndex</i> | TX register index, see <a href="#">mu_msg_reg_index_t</a> . |
| <i>msg</i>      | Message to send.                                            |

#### 25.5.4 void MU\_SendMsg ( MU\_Type \* *base*, uint32\_t *regIndex*, uint32\_t *msg* )

This function waits until the TX register is empty and sends the message.

Parameters

|                 |                                                             |
|-----------------|-------------------------------------------------------------|
| <i>base</i>     | MU peripheral base address.                                 |
| <i>regIndex</i> | MU message register, see <a href="#">mu_msg_reg_index_t</a> |
| <i>msg</i>      | Message to send.                                            |

#### 25.5.5 static uint32\_t MU\_ReceiveMsgNonBlocking ( MU\_Type \* *base*, uint32\_t *regIndex* ) [inline], [static]

This function reads a message from the specific RX register. It does not check whether the RX register is full or not. The upper layer should make sure the RX register is full before calling this function. This function can be used in ISR for better performance.

```
* uint32_t msg;
* while (!(kMU_Rx0FullFlag & MU_GetStatusFlags(base)))
* {
* } Wait for the RX0 register full.
*
* msg = MU_ReceiveMsgNonBlocking(base, kMU_MsgReg0); Read message from RX0
 register.
*
```

## Parameters

|             |                                                          |
|-------------|----------------------------------------------------------|
| <i>base</i> | MU peripheral base address.                              |
| <i>RX</i>   | register index, see <a href="#">mu_msg_reg_index_t</a> . |

## Returns

The received message.

**25.5.6 `uint32_t MU_ReceiveMsg ( MU_Type * base, uint32_t regIndex )`**

This function waits until the RX register is full and receives the message.

## Parameters

|                 |                                                             |
|-----------------|-------------------------------------------------------------|
| <i>base</i>     | MU peripheral base address.                                 |
| <i>regIndex</i> | MU message register, see <a href="#">mu_msg_reg_index_t</a> |

## Returns

The received message.

**25.5.7 `static void MU_SetFlagsNonBlocking ( MU_Type * base, uint32_t flags ) [inline], [static]`**

This function sets the 3-bit MU flags directly. Every time the 3-bit MU flags are changed, the status flag kMU\_FlagsUpdatingFlag asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag kMU\_FlagsUpdatingFlag is cleared by hardware. During the flags updating period, the flags cannot be changed. The upper layer should make sure the status flag kMU\_FlagsUpdatingFlag is cleared before calling this function.

```
* while (kMU_FlagsUpdatingFlag & MU_GetStatusFlags(base))
{
} Wait for previous MU flags updating.
*
* MU_SetFlagsNonBlocking(base, 0U); Set the mU flags.
*
```

Parameters

|              |                             |
|--------------|-----------------------------|
| <i>base</i>  | MU peripheral base address. |
| <i>flags</i> | The 3-bit MU flags to set.  |

### 25.5.8 void MU\_SetFlags ( MU\_Type \* *base*, uint32\_t *flags* )

This function blocks setting the 3-bit MU flags. Every time the 3-bit MU flags are changed, the status flag kMU\_FlagsUpdatingFlag asserts indicating the 3-bit MU flags are updating to the other side. After the 3-bit MU flags are updated, the status flag kMU\_FlagsUpdatingFlag is cleared by hardware. During the flags updating period, the flags cannot be changed. This function waits for the MU status flag kMU\_FlagsUpdatingFlag cleared and sets the 3-bit MU flags.

Parameters

|              |                             |
|--------------|-----------------------------|
| <i>base</i>  | MU peripheral base address. |
| <i>flags</i> | The 3-bit MU flags to set.  |

### 25.5.9 static uint32\_t MU\_GetFlags ( MU\_Type \* *base* ) [inline], [static]

This function gets the current 3-bit MU flags on the current side.

Parameters

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

Returns

*flags* Current value of the 3-bit flags.

### 25.5.10 static uint32\_t MU\_GetStatusFlags ( MU\_Type \* *base* ) [inline], [static]

This function returns the bit mask of the MU status flags. See \_mu\_status\_flags.

```
* uint32_t flags;
* flags = MU_GetStatusFlags(base); Get all status flags.
* if (kMU_Tx0EmptyFlag & flags)
* {
* The TX0 register is empty. Message can be sent.
* MU_SendMsgNonBlocking(base, kMU_MsgReg0, MSG0_VAL);
* }
```

```
* if (kMU_Tx1EmptyFlag & flags)
{
 The TX1 register is empty. Message can be sent.
 MU_SendMsgNonBlocking(base, kMU_MsgReg1, MSG1_VAL);
}
```

## Parameters

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

## Returns

Bit mask of the MU status flags, see `_mu_status_flags`.

### 25.5.11 static uint32\_t MU\_GetInterruptsPending ( MU\_Type \* *base* ) [inline], [static]

This function returns the bit mask of the pending MU IRQs.

## Parameters

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

## Returns

Bit mask of the MU IRQs pending.

### 25.5.12 static void MU\_ClearStatusFlags ( MU\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

This function clears the specific MU status flags. The flags to clear should be passed in as bit mask. See `_mu_status_flags`.

```
* Clear general interrupt 0 and general interrupt 1 pending flags.
* MU_ClearStatusFlags(base, kMU_GenInt0Flag |
 kMU_GenInt1Flag);
*
```

## Parameters

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i> | MU peripheral base address.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <i>mask</i> | <p>Bit mask of the MU status flags. See <code>_mu_status_flags</code>. The following flags are cleared by hardware, this function could not clear them.</p> <ul style="list-style-type: none"> <li>• <code>kMU_Tx0EmptyFlag</code></li> <li>• <code>kMU_Tx1EmptyFlag</code></li> <li>• <code>kMU_Tx2EmptyFlag</code></li> <li>• <code>kMU_Tx3EmptyFlag</code></li> <li>• <code>kMU_Rx0FullFlag</code></li> <li>• <code>kMU_Rx1FullFlag</code></li> <li>• <code>kMU_Rx2FullFlag</code></li> <li>• <code>kMU_Rx3FullFlag</code></li> <li>• <code>kMU_EventPendingFlag</code></li> <li>• <code>kMU_FlagsUpdatingFlag</code></li> <li>• <code>kMU_OtherSideInResetFlag</code></li> </ul> |

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

This function enables the specific MU interrupts. The interrupts to enable should be passed in as bit mask. See `_mu_interrupt_enable`.

```
* Enable general interrupt 0 and TX0 empty interrupt.
* MU_EnableInterrupts(base, kMU_GenInt0InterruptEnable |
 kMU_Tx0EmptyInterruptEnable);
*
```

## Parameters

|             |                                                                        |
|-------------|------------------------------------------------------------------------|
| <i>base</i> | MU peripheral base address.                                            |
| <i>mask</i> | Bit mask of the MU interrupts. See <code>_mu_interrupt_enable</code> . |

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

This function disables the specific MU interrupts. The interrupts to disable should be passed in as bit mask. See `_mu_interrupt_enable`.

```
* Disable general interrupt 0 and TX0 empty interrupt.
* MU_DisableInterrupts(base, kMU_GenInt0InterruptEnable |
 kMU_Tx0EmptyInterruptEnable);
*
```

## Parameters

|             |                                                                        |
|-------------|------------------------------------------------------------------------|
| <i>base</i> | MU peripheral base address.                                            |
| <i>mask</i> | Bit mask of the MU interrupts. See <code>_mu_interrupt_enable</code> . |

**25.5.15 status\_t MU\_TriggerInterrupts ( MU\_Type \* *base*, uint32\_t *mask* )**

This function triggers the specific interrupts to the other core. The interrupts to trigger are passed in as bit mask. See `_mu_interrupt_trigger`. The MU should not trigger an interrupt to the other core when the previous interrupt has not been processed by the other core. This function checks whether the previous interrupts have been processed. If not, it returns an error.

```
* if (kStatus_Success != MU_TriggerInterrupts(base,
 kMU_GenInt0InterruptTrigger |
 kMU_GenInt2InterruptTrigger))
{
 Previous general purpose interrupt 0 or general purpose interrupt 2
 has not been processed by the other core.
}
```

## Parameters

|             |                                                                                 |
|-------------|---------------------------------------------------------------------------------|
| <i>base</i> | MU peripheral base address.                                                     |
| <i>mask</i> | Bit mask of the interrupts to trigger. See <code>_mu_interrupt_trigger</code> . |

## Return values

|                        |                                              |
|------------------------|----------------------------------------------|
| <i>kStatus_Success</i> | Interrupts have been triggered successfully. |
| <i>kStatus_Fail</i>    | Previous interrupts have not been accepted.  |

**25.5.16 static void MU\_ClearNmi ( MU\_Type \* *base* ) [inline], [static]**

This function clears non-maskable interrupt (NMI) sent by the other core.

## Parameters

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

**25.5.17 void MU\_BootCoreB ( MU\_Type \* *base*, mu\_core\_boot\_mode\_t *mode* )**

This function sets the B side core's boot configuration and releases the core from reset.

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | MU peripheral base address. |
| <i>mode</i> | Core B boot mode.           |

Note

Only MU side A can use this function.

#### **25.5.18 static void MU\_HoldCoreBReset ( MU\_Type \* *base* ) [inline], [static]**

This function causes the core of B side to be held in reset following any reset event.

Parameters

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

Note

Only A side could call this function.

#### **25.5.19 void MU\_BootOtherCore ( MU\_Type \* *base*, mu\_core\_boot\_mode\_t *mode* )**

This function boots the other core with a boot configuration.

Parameters

|             |                             |
|-------------|-----------------------------|
| <i>base</i> | MU peripheral base address. |
| <i>mode</i> | The other core boot mode.   |

#### **25.5.20 static void MU\_HoldOtherCoreReset ( MU\_Type \* *base* ) [inline], [static]**

This function causes the other core to be held in reset following any reset event.

## Parameters

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

### 25.5.21 static void MU\_ResetBothSides ( MU\_Type \* *base* ) [inline], [static]

This function resets the MU for both A side and B side. Before reset, it is recommended to interrupt processor B, because this function may affect the ongoing processor B programs.

## Parameters

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

## Note

For some platforms, only MU side A could use this function, check reference manual for details.

### 25.5.22 void MU\_HardwareResetOtherCore ( MU\_Type \* *base*, bool *waitReset*, bool *holdReset*, mu\_core\_boot\_mode\_t *bootMode* )

This function resets the other core, the other core could mask the hardware reset by calling MU\_MaskHardwareReset. The hardware reset mask feature is only available for some platforms. This function could be used together with MU\_BootOtherCore to control the other core reset workflow.

Example 1: Reset the other core, and no hold reset

```
* MU_HardwareResetOtherCore(MU_A, true, false, bootMode);
*
```

In this example, the core at MU side B will reset with the specified boot mode.

Example 2: Reset the other core and hold it, then boot the other core later.

```
* Here the other core enters reset, and the reset is hold
* MU_HardwareResetOtherCore(MU_A, true, true, modeDontCare);
* Current core boot the other core when necessary.
* MU_BootOtherCore(MU_A, bootMode);
*
```

Parameters

|                  |                                                                                                                                                                                                                                                                                                                |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>base</i>      | MU peripheral base address.                                                                                                                                                                                                                                                                                    |
| <i>waitReset</i> | <p>Wait the other core enters reset.</p> <ul style="list-style-type: none"> <li>• true: Wait until the other core enters reset, if the other core has masked the hardware reset, then this function will be blocked.</li> <li>• false: Don't wait the reset.</li> </ul>                                        |
| <i>holdReset</i> | <p>Hold the other core reset or not.</p> <ul style="list-style-type: none"> <li>• true: Hold the other core in reset, this function returns directly when the other core enters reset.</li> <li>• false: Don't hold the other core in reset, this function waits until the other core out of reset.</li> </ul> |
| <i>bootMode</i>  | Boot mode of the other core, if <i>holdReset</i> is true, this parameter is useless.                                                                                                                                                                                                                           |

### 25.5.23 static void MU\_SetClockOnOtherCoreEnable ( MU\_Type \* *base*, bool *enable* ) [inline], [static]

This function enables or disables the platform clock on the other core when that core enters a stop mode. If disabled, the platform clock for the other core is disabled when it enters stop mode. If enabled, the platform clock keeps running on the other core in stop mode, until this core also enters stop mode.

Parameters

|               |                                                |
|---------------|------------------------------------------------|
| <i>base</i>   | MU peripheral base address.                    |
| <i>enable</i> | Enable or disable the clock on the other core. |

### 25.5.24 static mu\_power\_mode\_t MU\_GetOtherCorePowerMode ( MU\_Type \* *base* ) [inline], [static]

This function gets the power mode of the other core.

Parameters

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

Returns

Power mode of the other core.

# Chapter 26

## PMC0: Power Management Controller

### 26.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Power Management Controller (PMC) module of MCUXpresso SDK devices. The Power Management Controller (PMC) can be divided in two parts: PMC 0 and PMC 1. The PMC 0 controls the Core 0, its SoG and RAM, and the PMC 1 controls the Core 1, its SoG and RAM. This driver is for PMC 0 only.

The PMC 0 has:

- the high-power (HP) and low-power (LP) Core Regulator;
- the high-power (HP) and low-power (LP) Array Regulator;
- the high-power (HP) and low-power (LP) 1.2V Low Voltage Detector (LVD) monitor (in regulators input);
- the high-power (HP) and low-power (LP) 1.2V High Voltage Detector (HVD) monitor (in regulators input);
- the bandgap;
- the forward bias (FBB) and the reverse back bias (RBB). In addition, the PMC has a 1.8 V POR (Power-On Reset) monitor to assure the voltage level in the Always-On power domain would be in the correct range to the correct functionality of the internal digital and analog blocks. Both PMCs receive requests from the MSMC to change the current power mode. Each PMC allows the customer to choose what features are enabled or disabled for each power mode using the PMC registers.

### 26.2 Typical use case

#### 26.2.1 Turn on the PMC 1 using LDO Regulator

After a POR event, when the PMC 0 is during RUN mode and the PMC 1 is turned off. The procedure to turn on the PMC 1 using the internal LDO Regulator.

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pmc0

#### 26.2.2 Turn on the PMC 1 using the PMIC

After a POR event, when the PMC 0 is during RUN mode and the PMC 1 is turned off. The procedure to turn on the PMC 1 using the external PMIC

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pmc0

### 26.2.3 Turn off the LDO Regulator

When the PMC 1 is during RUN mode, the LDO Regulator can be programmed to be turned off in the next transition from RUN to VLLS power mode. As in VLLS the regulator is disconnected from the load by the switches (switches are OFF), a external regulator can assume the power supply (PMIC).

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pmc0

### 26.2.4 Turn on the LDO Regulator

When the PMC 1 is during VLLS mode, the LDO Regulator can be turned on in a transition to RUN mode.

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pmc0

### 26.2.5 Change the Core Regulator voltage level in PMC 0 RUN or HSRUN mode

To change the Core Regulator voltage level when the PMC 0 is in RUN mode:

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pmc0

To change the Core Regulator voltage level when the PMC 0 is in HSRUN mode:

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pmc0

### 26.2.6 Change the SRAMs power mode during PMC 0 RUN mode

To change the SRAMs power mode during the PMC 0 RUN mode.

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/pmc0

## Files

- file [fsl\\_pmc0.h](#)

## Data Structures

- struct [pmc0\\_hsrn\\_mode\\_config\\_t](#)  
*PMC 0 HSRUN mode configuration.* [More...](#)
- struct [pmc0\\_run\\_mode\\_config\\_t](#)  
*PMC 0 RUN mode configuration.* [More...](#)
- struct [pmc0\\_vlpr\\_mode\\_config\\_t](#)  
*PMC 0 VLPR mode configuration.* [More...](#)
- struct [pmc0\\_stop\\_mode\\_config\\_t](#)  
*PMC 0 STOP mode configuration.* [More...](#)

- struct `pmc0_vlps_mode_config_t`  
*PMC 0 VLPS mode configuration. [More...](#)*
- struct `pmc0_lls_mode_config_t`  
*PMC 0 LLS mode configuration. [More...](#)*
- struct `pmc0_vlls_mode_config_t`  
*PMC 0 VLLS mode configuration. [More...](#)*
- struct `pmc0_bias_config_t`  
*PMC 0 bias configuration. [More...](#)*

## Macros

- `#define CORE_REGULATOR_VOLT_LEVEL_MAX 50U`  
*MAX valid values of Core Regulator Voltage Level.*

## Enumerations

- enum `pmc0_high_volt_detect_monitor_select_t` {
   
`kPMC0_HighVoltDetectLowPowerMonitor = 0U,`
  
`kPMC0_HighVoltDetectHighPowerMonitor = 1U }`
  
*High Voltage Detect Monitor Select.*
- enum `pmc0_low_volt_detect_monitor_select_t` {
   
`kPMC0_LowVoltDetectLowPowerMonitor = 0U,`
  
`kPMC0_LowVoltDetectHighPowerMonitor = 1U }`
  
*Low Voltage Detect Monitor Select.*
- enum `pmc0_core_regulator_select_t` {
   
`kPMC0_CoreLowPowerRegulator = 0U,`
  
`kPMC0_CoreHighPowerRegulator = 1U }`
  
*Core Regulator Select.*
- enum `pmc0_array_regulator_select_t` {
   
`kPMC0_ArrayLowPowerRegulator = 0U,`
  
`kPMC0_ArrayHighPowerRegulator = 1U }`
  
*Array Regulator Select.*
- enum `pmc0_vlls_array_regulator_select_t` {
   
`kPMC0_VllsArrayRegulatorOff = 0U,`
  
`kPMC0_VllsArrayLowPowerRegulator = 2U,`
  
`kPMC0_VllsArrayHighPowerRegulator = 3U }`
  
*VLLS mode array Regulator Select.*
- enum `pmc0_fbb_p_well_voltage_level_select_t` {
   
`kPMC0_FbbPWellNoBiasCondition = 0U,`
  
`kPMC0_FbbPWellVoltageLevelAt50Mv = 1U,`
  
`kPMC0_FbbPWellVoltageLevelAt150Mv = 2U,`
  
`kPMC0_FbbPWellVoltageLevelAt100Mv = 3U,`
  
`kPMC0_FbbPWellVoltageLevelAt350Mv = 4U,`
  
`kPMC0_FbbPWellVoltageLevelAt300Mv = 5U,`
  
`kPMC0_FbbPWellVoltageLevelAt200Mv = 6U,`
  
`kPMC0_FbbPWellVoltageLevelAt250Mv = 7U }`
  
*FBB P-Well voltage level select.*

- enum pmc0\_fbb\_n\_well\_voltage\_level\_select\_t {
   
kPMC0\_FbbNWellNoBiasCondition = 0U,
   
kPMC0\_FbbNWellVoltageLevelAtMinus50Mv = 1U,
   
kPMC0\_FbbNWellVoltageLevelAtMinus150Mv = 2U,
   
kPMC0\_FbbNWellVoltageLevelAtMinus100Mv = 3U,
   
kPMC0\_FbbNWellVoltageLevelAtMinus350Mv = 4U,
   
kPMC0\_FbbNWellVoltageLevelAtMinus300Mv = 5U,
   
kPMC0\_FbbNWellVoltageLevelAtMinus200Mv = 6U,
   
kPMC0\_FbbNWellVoltageLevelAtMinus250Mv = 7U }
   
*FBB N-Well voltage level select.*
- enum pmc0\_rbb\_p\_well\_voltage\_level\_select\_t {
   
kPMC0\_RBBPWellVoltageLevelAtMinus0\_5V = 0U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus0\_6V = 1U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus0\_7V = 2U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus0\_8V = 3U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus0\_9V = 4U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus1\_0V = 5U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus1\_1V = 6U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus1\_2V = 7U,
   
kPMC0\_RBBPWellVoltageLevelAtMinus1\_3V = 8U }
   
*RBB P-Well voltage level select.*
- enum pmc0\_rbb\_n\_well\_voltage\_level\_select\_t {
   
kPMC0\_RBBNWellVoltageLevelAt0\_5V = 0U,
   
kPMC0\_RBBNWellVoltageLevelAt0\_6V = 1U,
   
kPMC0\_RBBNWellVoltageLevelAt0\_7V = 2U,
   
kPMC0\_RBBNWellVoltageLevelAt0\_8V = 3U,
   
kPMC0\_RBBNWellVoltageLevelAt0\_9V = 4U,
   
kPMC0\_RBBNWellVoltageLevelAt1\_0V = 5U,
   
kPMC0\_RBBNWellVoltageLevelAt1\_1V = 6U,
   
kPMC0\_RBBNWellVoltageLevelAt1\_2V = 7U,
   
kPMC0\_RBBNWellVoltageLevelAt1\_3V = 8U }
   
*RBB N-Well voltage level select.*
- enum \_pmc0\_status\_flags {
   
kPMC0\_LowVoltDetectEventFlag,
   
kPMC0\_LowVoltDetectValueFlag = PMC0\_STATUS\_LVDV\_MASK,
   
kPMC0\_HighVoltDetectEventFlag,
   
kPMC0\_HighVoltDetectValueFlag = PMC0\_STATUS\_HVDV\_MASK,
   
kPMC0\_CoreRegulatorVoltLevelFlag = PMC0\_STATUS\_COREVLF\_MASK,
   
kPMC0\_SramFlag,
   
kPMC0\_PMC1VoltageSourceFlag }
- *PMC 0 status flags.*
- enum \_pmc0\_power\_mode\_status\_flags {

```
kPMC0_HSRUNModeStatusFlags = 0U,
kPMC0_RUNModeStatusFlags = 1U,
kPMC0_STOPModeStatusFlags = 2U,
kPMC0_VLPRModeStatusFlags = 3U,
kPMC0_VLPSSModeStatusFlags = 4U,
kPMC0_LLSSModeStatusFlags = 5U,
kPMC0_VLLSSModeStatusFlags = 6U }
```

*PMC 0 power mode status flags.*

## Driver version

- #define **FSL\_PMC0\_DRIVER\_VERSION** (MAKE\_VERSION(2, 1, 0))  
*PMC 0 driver version.*

## Power Management Controller Control APIs

- static void **PMC0\_ConfigureHsrnMode** (const **pmc0\_hsrn\_mode\_config\_t** \*config)  
*Configure the HSRUN power mode.*
- static void **PMC0\_ConfigureRunMode** (const **pmc0\_run\_mode\_config\_t** \*config)  
*Configure the RUN power mode.*
- static void **PMC0\_ConfigureVlprMode** (const **pmc0\_vlpr\_mode\_config\_t** \*config)  
*Configure the VLPR power mode.*
- static void **PMC0\_ConfigureStopMode** (const **pmc0\_stop\_mode\_config\_t** \*config)  
*Configure the STOP power mode.*
- static void **PMC0\_ConfigureVlpsMode** (const **pmc0\_vlps\_mode\_config\_t** \*config)  
*Configure the VLPS power mode.*
- static void **PMC0\_ConfigureLlsMode** (const **pmc0\_lls\_mode\_config\_t** \*config)  
*Configure the LLS power mode.*
- static void **PMC0\_ConfigureVllsMode** (const **pmc0\_vlls\_mode\_config\_t** \*config)  
*Configure the VLLS power mode.*
- static uint32\_t **PMC0\_GetPMC0PowerModeStatusFlags** (void)  
*Get current power mode of PMC 0.*
- static bool **PMC0\_GetPMC0PowerTransitionStatus** (void)  
*Get the status of PMC 0 power mode transition.*
- static uint32\_t **PMC0\_GetPMC1PowerModeStatusFlags** (void)  
*Get current power mode of PMC 1.*
- static bool **PMC0\_GetPMC1PowerTransitionStatus** (void)  
*Get the status of PMC 1 power mode transition.*
- static uint32\_t **PMC0\_GetStatusFlags** (void)  
*Gets PMC 0 status flags.*
- static void **PMC0\_EnableLowVoltDetectInterrupt** (void)  
*Enables the 1.2V Low-Voltage Detector interrupt.*
- static void **PMC0\_DisableLowVoltDetectInterrupt** (void)  
*Disables the 1.2V Low-Voltage Detector interrupt.*
- static void **PMC0\_ClearLowVoltDetectFlag** (void)  
*Clears the 1.2V Low-Voltage Detector flag.*
- static void **PMC0\_EnableHighVoltDetectInterrupt** (void)  
*Enables the 1.8V High-Voltage Detector interrupt.*
- static void **PMC0\_DisableHighVoltDetectInterrupt** (void)  
*Disables the 1.8V High-Voltage Detector interrupt.*

- static void **PMC0\_ClearHighVoltDetectFlag** (void)  
*Clears the 1.8V High-Voltage Detector flag.*
- static void **PMC0\_EnableLowVoltDetectReset** (bool enable)  
*Enables the 1.2V Low-Voltage Detector reset.*
- static void **PMC0\_EnableHighVoltDetectReset** (bool enable)  
*Enables the 1.8V High-Voltage Detector reset.*
- static void **PMC0\_ClearPadsIsolation** (void)  
*Releases/clears the isolation in the PADS.*
- static void **PMC0\_PowerOnPmc1** (void)  
*Powers on PMC 1.*
- static void **PMC0\_EnableWaitLdoOkSignal** (bool enable)  
*Enables to wait LDO OK signal.*
- static void **PMC0\_EnablePmc1LdoRegulator** (bool enable)  
*Enables PMC 1 LDO Regulator.*
- static void **PMC0\_EnablePmc1RBBMode** (bool enable)  
*Enable the PMC 1 RBB(reverse back bias) mode.*
- static void **PMC0\_SetBiasConfig** (const **pmc0\_bias\_config\_t** \*config)  
*Configure the PMC 0 bias voltage level and enable/disable pull-down.*
- static void **PMC0\_ConfigureSramBankPowerDown** (uint32\_t bankMask)  
*Configures PMC 0 SRAM bank power down.*
- static void **PMC0\_ConfigureSramBankPowerDownStopMode** (uint32\_t bankMask)  
*Configures PMC 0 SRAM bank power down in stop modes.*
- static void **PMC0\_ConfigureSramBankPowerDownStandbyMode** (uint32\_t bankMask)  
*Configures PMC 0 SRAM bank power down in Standby Mode.*
- static void **PMC0\_EnableTemperatureSensor** (bool enable)  
*Enable/disable internal temperature sensor.*
- static void **PMC0\_SetTemperatureSensorMode** (uint8\_t mode)  
*Set temperature sensor mode.*

## 26.3 Data Structure Documentation

### 26.3.1 struct pmc0\_hsrn\_mode\_config\_t

#### Data Fields

- uint32\_t **\_\_pad0\_\_**: 16  
*Reserved.*
- uint32\_t **coreRegulatorVoltLevel**: 6  
*Core Regulator Voltage Level.*
- uint32\_t **\_\_pad1\_\_**: 2  
*Reserved.*
- uint32\_t **enableForwardBias**: 1  
*Enable forward bias.*
- uint32\_t **\_\_pad2\_\_**: 7  
*Reserved.*

**Field Documentation**

- (1) `uint32_t pmc0_hsrn_mode_config_t::__pad0__`
- (2) `uint32_t pmc0_hsrn_mode_config_t::coreRegulatorVoltLevel`
- (3) `uint32_t pmc0_hsrn_mode_config_t::__pad1__`
- (4) `uint32_t pmc0_hsrn_mode_config_t::enableForwardBias`
- (5) `uint32_t pmc0_hsrn_mode_config_t::__pad2__`

**26.3.2 struct pmc0\_run\_mode\_config\_t****Data Fields**

- `uint32_t __pad0__`: 16  
*Reserved.*
- `uint32_t coreRegulatorVoltLevel`: 6  
*Core Regulator Voltage Level.*
- `uint32_t __pad1__`: 10  
*Reserved.*

**Field Documentation**

- (1) `uint32_t pmc0_run_mode_config_t::__pad0__`
- (2) `uint32_t pmc0_run_mode_config_t::coreRegulatorVoltLevel`
- (3) `uint32_t pmc0_run_mode_config_t::__pad1__`

**26.3.3 struct pmc0\_vlpr\_mode\_config\_t****Data Fields**

- `uint32_t arrayRegulatorSelect`: 1  
*Array Regulator Select.*
- `uint32_t __pad0__`: 1  
*Reserved.*
- `uint32_t coreRegulatorSelect`: 1  
*Core Regulator Select.*
- `uint32_t __pad1__`: 1  
*Reserved.*
- `uint32_t lvdMonitorSelect`: 1  
*1.2V LVD Monitor Select.*
- `uint32_t hvdMonitorSelect`: 1  
*1.2V HVD Monitor Select.*
- `uint32_t __pad2__`: 1  
*Reserved.*

- `uint32_t enableForceHpBandgap: 1`  
*Enable force HP band-gap.*
- `uint32_t __pad3__: 8`  
*Reserved.*
- `uint32_t coreRegulatorVoltLevel: 6`  
*Core Regulator Voltage Level.*
- `uint32_t __pad4__: 6`  
*Reserved.*
- `uint32_t enableReverseBackBias: 1`  
*Enable reverse back bias.*
- `uint32_t __pad5__: 3`  
*Reserved.*

## Field Documentation

(1) `uint32_t pmc0_vlpr_mode_config_t::arrayRegulatorSelect`

`pmc0_array_regulator_select_t`

(2) `uint32_t pmc0_vlpr_mode_config_t::__pad0__`

(3) `uint32_t pmc0_vlpr_mode_config_t::coreRegulatorSelect`

`pmc0_core_regulator_select_t`

(4) `uint32_t pmc0_vlpr_mode_config_t::__pad1__`

(5) `uint32_t pmc0_vlpr_mode_config_t::lvdMonitorSelect`

`pmc0_low_volt_detect_monitor_select_t`

(6) `uint32_t pmc0_vlpr_mode_config_t::hvdMonitorSelect`

`pmc0_high_volt_detect_monitor_select_t`

- (7) `uint32_t pmc0_vlpr_mode_config_t::__pad2__`
- (8) `uint32_t pmc0_vlpr_mode_config_t::enableForceHpBandgap`
- (9) `uint32_t pmc0_vlpr_mode_config_t::__pad3__`
- (10) `uint32_t pmc0_vlpr_mode_config_t::coreRegulatorVoltLevel`
- (11) `uint32_t pmc0_vlpr_mode_config_t::__pad4__`
- (12) `uint32_t pmc0_vlpr_mode_config_t::enableReverseBackBias`
- (13) `uint32_t pmc0_vlpr_mode_config_t::__pad5__`

### 26.3.4 struct pmc0\_stop\_mode\_config\_t

#### Data Fields

- `uint32_t __pad0__`: 16  
*Reserved.*
- `uint32_t coreRegulatorVoltLevel`: 6  
*Core Regulator Voltage Level.*
- `uint32_t __pad1__`: 10  
*Reserved.*

#### Field Documentation

- (1) `uint32_t pmc0_stop_mode_config_t::__pad0__`
- (2) `uint32_t pmc0_stop_mode_config_t::coreRegulatorVoltLevel`
- (3) `uint32_t pmc0_stop_mode_config_t::__pad1__`

### 26.3.5 struct pmc0\_vlps\_mode\_config\_t

#### Data Fields

- `uint32_t arrayRegulatorSelect`: 1  
*Array Regulator Select.*
- `uint32_t __pad0__`: 1  
*Reserved.*
- `uint32_t coreRegulatorSelect`: 1  
*Core Regulator Select.*
- `uint32_t __pad1__`: 1  
*Reserved.*
- `uint32_t lvdMonitorSelect`: 1  
*1.2V LVD Monitor Select.*
- `uint32_t hvdMonitorSelect`: 1  
*1.2V HVD Monitor Select.*

- `uint32_t __pad2__`: 1  
*Reserved.*
- `uint32_t enableForceHpBandgap`: 1  
*Enable force HP band-gap.*
- `uint32_t __pad3__`: 8  
*Reserved.*
- `uint32_t coreRegulatorVoltLevel`: 6  
*Core Regulator Voltage Level.*
- `uint32_t __pad4__`: 6  
*Reserved.*
- `uint32_t enableReverseBackBias`: 1  
*Enable reverse back bias.*
- `uint32_t __pad5__`: 3  
*Reserved.*

## Field Documentation

- (1) `uint32_t pmc0_vlps_mode_config_t::arrayRegulatorSelect`  
`pmc0_array_regulator_select_t`
- (2) `uint32_t pmc0_vlps_mode_config_t::__pad0__`
- (3) `uint32_t pmc0_vlps_mode_config_t::coreRegulatorSelect`  
`pmc0_core_regulator_select_t`
- (4) `uint32_t pmc0_vlps_mode_config_t::__pad1__`
- (5) `uint32_t pmc0_vlps_mode_config_t::lvdMonitorSelect`  
`pmc0_low_volt_detect_monitor_select_t`
- (6) `uint32_t pmc0_vlps_mode_config_t::hvdMonitorSelect`  
`pmc0_high_volt_detect_monitor_select_t`

- (7) `uint32_t pmc0_vlps_mode_config_t::__pad2__`
- (8) `uint32_t pmc0_vlps_mode_config_t::enableForceHpBandgap`
- (9) `uint32_t pmc0_vlps_mode_config_t::__pad3__`
- (10) `uint32_t pmc0_vlps_mode_config_t::coreRegulatorVoltLevel`
- (11) `uint32_t pmc0_vlps_mode_config_t::__pad4__`
- (12) `uint32_t pmc0_vlps_mode_config_t::enableReverseBackBias`
- (13) `uint32_t pmc0_vlps_mode_config_t::__pad5__`

### 26.3.6 struct pmc0\_lls\_mode\_config\_t

#### Data Fields

- `uint32_t arrayRegulatorSelect: 1`  
*Array Regulator Select.*
- `uint32_t __pad0__: 1`  
*Reserved.*
- `uint32_t coreRegulatorSelect: 1`  
*Core Regulator Select.*
- `uint32_t __pad1__: 1`  
*Reserved.*
- `uint32_t lvdMonitorSelect: 1`  
*1.2V LVD Monitor Select.*
- `uint32_t hvdMonitorSelect: 1`  
*1.2V HVD Monitor Select.*
- `uint32_t __pad2__: 1`  
*Reserved.*
- `uint32_t enableForceHpBandgap: 1`  
*Enable force HP band-gap.*
- `uint32_t __pad3__: 8`  
*Reserved.*
- `uint32_t coreRegulatorVoltLevel: 6`  
*Core Regulator Voltage Level.*
- `uint32_t __pad4__: 6`  
*Reserved.*
- `uint32_t enableReverseBackBias: 1`  
*Enable reverse back bias.*
- `uint32_t __pad5__: 3`  
*Reserved.*

#### Field Documentation

- (1) `uint32_t pmc0_lls_mode_config_t::arrayRegulatorSelect`

`pmc0_array_regulator_select_t`

```

(2) uint32_t pmc0_lls_mode_config_t::__pad0__
(3) uint32_t pmc0_lls_mode_config_t::coreRegulatorSelect
 pmc0_core_regulator_select_t
(4) uint32_t pmc0_lls_mode_config_t::__pad1__
(5) uint32_t pmc0_lls_mode_config_t::lvdMonitorSelect
 pmc0_low_volt_detect_monitor_select_t
(6) uint32_t pmc0_lls_mode_config_t::hvdMonitorSelect
 pmc0_high_volt_detect_monitor_select_t
(7) uint32_t pmc0_lls_mode_config_t::__pad2__
(8) uint32_t pmc0_lls_mode_config_t::enableForceHpBandgap
(9) uint32_t pmc0_lls_mode_config_t::__pad3__
(10) uint32_t pmc0_lls_mode_config_t::coreRegulatorVoltLevel
(11) uint32_t pmc0_lls_mode_config_t::__pad4__
(12) uint32_t pmc0_lls_mode_config_t::enableReverseBackBias
(13) uint32_t pmc0_lls_mode_config_t::__pad5__

```

### 26.3.7 struct pmc0\_vlls\_mode\_config\_t

#### Data Fields

- uint32\_t arrayRegulatorSelect: 2  
*Array Regulator Select.*
- uint32\_t \_\_pad0\_\_: 2  
*Reserved.*
- uint32\_t lvdMonitorSelect: 1  
*1.2V LVD Monitor Select.*
- uint32\_t hvdMonitorSelect: 1  
*1.2V HVD Monitor Select.*
- uint32\_t \_\_pad1\_\_: 1  
*Reserved.*
- uint32\_t enableForceHpBandgap: 1  
*Enable force HP band-gap.*
- uint32\_t \_\_pad2\_\_: 24  
*Reserved.*

**Field Documentation**

- (1) `uint32_t pmc0_vlls_mode_config_t::arrayRegulatorSelect`  
`pmc0_vlls_array_regulator_select_t`
- (2) `uint32_t pmc0_vlls_mode_config_t::__pad0__`
- (3) `uint32_t pmc0_vlls_mode_config_t::lvdMonitorSelect`  
`pmc0_low_volt_detect_monitor_select_t`
- (4) `uint32_t pmc0_vlls_mode_config_t::hvdMonitorSelect`  
`pmc0_high_volt_detect_monitor_select_t`
- (5) `uint32_t pmc0_vlls_mode_config_t::__pad1__`
- (6) `uint32_t pmc0_vlls_mode_config_t::enableForceHpbBandgap`
- (7) `uint32_t pmc0_vlls_mode_config_t::__pad2__`

**26.3.8 struct pmc0\_bias\_config\_t****Data Fields**

- `uint32_t __pad0__: 4`  
*Reserved.*
- `uint32_t RBBPWellVoltageLevelSelect: 4`  
*Select PMC0 RBB P-Well voltage level.*
- `uint32_t __pad1__: 3`  
*Reserved.*
- `uint32_t DisableRBBPullDown: 1`  
*Disable RBB pull-down.*
- `uint32_t FBBNWellVoltageLevelSelect: 4`  
*Select PMC0 FBB N-Well voltage level.*
- `uint32_t __pad2__: 4`  
*Reserved.*
- `uint32_t FBBPWellVoltageLevelSelect: 4`  
*Select PMC0 FBB P-Well voltage level.*
- `uint32_t __pad3__: 4`  
*Reserved.*

**Field Documentation**

- (1) `uint32_t pmc0_bias_config_t::__pad0__`
- (2) `uint32_t pmc0_bias_config_t::RBBPWellVoltageLevelSelect`  
`pmc0_rbb_p_well_voltage_level_select_t`

- (3) `uint32_t pmc0_bias_config_t::__pad1__`
- (4) `uint32_t pmc0_bias_config_t::DisableRBBPullDown`

'1' means to disable pull-down. '0' means to enable pull-down.

- (5) `uint32_t pmc0_bias_config_t::FBBNWellVoltageLevelSelect`

`pmc0_fbb_n_well_voltage_level_select_t`

- (6) `uint32_t pmc0_bias_config_t::__pad2__`

- (7) `uint32_t pmc0_bias_config_t::FBBPWellVoltageLevelSelect`

`pmc0_fbb_p_well_voltage_level_select_t`

- (8) `uint32_t pmc0_bias_config_t::__pad3__`

## 26.4 Enumeration Type Documentation

### 26.4.1 enum pmc0\_high\_volt\_detect\_monitor\_select\_t

Enumerator

*kPMC0\_HighVoltDetectLowPowerMonitor* LP monitor is selected.

*kPMC0\_HighVoltDetectHighPowerMonitor* HP monitor is selected.

### 26.4.2 enum pmc0\_low\_volt\_detect\_monitor\_select\_t

Enumerator

*kPMC0\_LowVoltDetectLowPowerMonitor* LP monitor is selected.

*kPMC0\_LowVoltDetectHighPowerMonitor* HP monitor is selected.

### 26.4.3 enum pmc0\_core\_regulator\_select\_t

Enumerator

*kPMC0\_CoreLowPowerRegulator* Core LP regulator is selected.

*kPMC0\_CoreHighPowerRegulator* Core HP regulator is selected.

### 26.4.4 enum pmc0\_array\_regulator\_select\_t

Enumerator

*kPMC0\_ArrayLowPowerRegulator* Array LP regulator is selected.

***kPMC0\_ArrayHighPowerRegulator*** Array HP regulator is selected.

#### 26.4.5 enum pmc0\_vlls\_array\_regulator\_select\_t

Enumerator

***kPMC0\_VllsArrayRegulatorOff*** Array regulator is selected OFF. This is selectable only for VLLS mode.

***kPMC0\_VllsArrayLowPowerRegulator*** Array LP regulator is selected.

***kPMC0\_VllsArrayHighPowerRegulator*** Array HP regulator is selected.

#### 26.4.6 enum pmc0\_fbb\_p\_well\_voltage\_level\_select\_t

Enumerator

***kPMC0\_FbbPWellNoBiasCondition*** No BIAS condition is selected.

***kPMC0\_FbbPWellVoltageLevelAt50Mv*** Voltage level at 50mV is selected.

***kPMC0\_FbbPWellVoltageLevelAt150Mv*** Voltage level at 150mV is selected.

***kPMC0\_FbbPWellVoltageLevelAt100Mv*** Voltage level at 100mV is selected.

***kPMC0\_FbbPWellVoltageLevelAt350Mv*** Voltage level at 350mV is selected.

***kPMC0\_FbbPWellVoltageLevelAt300Mv*** Voltage level at 300mV is selected.

***kPMC0\_FbbPWellVoltageLevelAt200Mv*** Voltage level at 200mV is selected.

***kPMC0\_FbbPWellVoltageLevelAt250Mv*** Voltage level at 250mV is selected.

#### 26.4.7 enum pmc0\_fbb\_n\_well\_voltage\_level\_select\_t

Enumerator

***kPMC0\_FbbNWellNoBiasCondition*** No BIAS condition is selected.

***kPMC0\_FbbNWellVoltageLevelAtMinus50Mv*** Voltage level at -50mV is selected.

***kPMC0\_FbbNWellVoltageLevelAtMinus150Mv*** Voltage level at -150mV is selected.

***kPMC0\_FbbNWellVoltageLevelAtMinus100Mv*** Voltage level at -100mV is selected.

***kPMC0\_FbbNWellVoltageLevelAtMinus350Mv*** Voltage level at -350mV is selected.

***kPMC0\_FbbNWellVoltageLevelAtMinus300Mv*** Voltage level at -300mV is selected.

***kPMC0\_FbbNWellVoltageLevelAtMinus200Mv*** Voltage level at -200mV is selected.

***kPMC0\_FbbNWellVoltageLevelAtMinus250Mv*** Voltage level at -250mV is selected.

## 26.4.8 enum pmc0\_rbb\_p\_well\_voltage\_level\_select\_t

Enumerator

- kPMC0\_RBBPWellVoltageLevelAtMinus0\_5V* Voltage level at -0.5V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus0\_6V* Voltage level at -0.6V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus0\_7V* Voltage level at -0.7V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus0\_8V* Voltage level at -0.8V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus0\_9V* Voltage level at -0.9V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus1\_0V* Voltage level at -1.0V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus1\_1V* Voltage level at -1.1V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus1\_2V* Voltage level at -1.2V is selected.
- kPMC0\_RBBPWellVoltageLevelAtMinus1\_3V* Voltage level at -1.3V is selected.

## 26.4.9 enum pmc0\_rbb\_n\_well\_voltage\_level\_select\_t

Enumerator

- kPMC0\_RBBNWellVoltageLevelAt0\_5V* Voltage level at 0.5V is selected.
- kPMC0\_RBBNWellVoltageLevelAt0\_6V* Voltage level at 0.6V is selected.
- kPMC0\_RBBNWellVoltageLevelAt0\_7V* Voltage level at 0.7V is selected.
- kPMC0\_RBBNWellVoltageLevelAt0\_8V* Voltage level at 0.8V is selected.
- kPMC0\_RBBNWellVoltageLevelAt0\_9V* Voltage level at 0.9V is selected.
- kPMC0\_RBBNWellVoltageLevelAt1\_0V* Voltage level at 1.0V is selected.
- kPMC0\_RBBNWellVoltageLevelAt1\_1V* Voltage level at 1.1V is selected.
- kPMC0\_RBBNWellVoltageLevelAt1\_2V* Voltage level at 1.2V is selected.
- kPMC0\_RBBNWellVoltageLevelAt1\_3V* Voltage level at 1.3V is selected.

## 26.4.10 enum \_pmc0\_status\_flags

Enumerator

- kPMC0\_LowVoltDetectEventFlag* 1.2V Low-Voltage Detector Flag, sets when low-voltage event was detected.
- kPMC0\_LowVoltDetectValueFlag* 1.2V Low-Voltage Detector Value, sets when current value of the 1.2V LVD monitor output is 1.
- kPMC0\_HighVoltDetectEventFlag* 1.8V High-Voltage Detector Flag, sets when high-voltage event was detected.
- kPMC0\_HighVoltDetectValueFlag* 1.8V High-Voltage Detector Value, sets when current value of the 1.8V HVD monitor output is 1.
- kPMC0\_CoreRegulatorVoltLevelFlag* Core Regulator Voltage Level Flag, sets when core regulator voltage level is changing (not stable).

***kPMC0\_SramFlag*** SRAM Flag, sets when a change mode request is being processed in the SRAMs.

***kPMC0\_PMC1VoltageSourceFlag*** This flag indicates what is the voltage source selected to supply the PMC 1 and where the sense point of the PMC 1's LVD/HVD is placed. '0' means internal LDO supplies the PMC 1. '1' means external PMIC supplies the PMC 1.

#### 26.4.11 enum \_pmc0\_power\_mode\_status\_flags

Enumerator

***kPMC0\_HSRUNModeStatusFlags*** The PMC 0 is in HSRUN mode.

***kPMC0\_RUNModeStatusFlags*** The PMC 0 is in RUN mode.

***kPMC0\_STOPModeStatusFlags*** The PMC 0 is in STOP mode.

***kPMC0\_VLPRModeStatusFlags*** The PMC 0 is in VLPR mode.

***kPMC0\_VLPSModeStatusFlags*** The PMC 0 is in VLPS mode.

***kPMC0\_LLSSModeStatusFlags*** The PMC 0 is in LLSS mode.

***kPMC0\_VLLSSModeStatusFlags*** The PMC 0 is in VLLS mode.

### 26.5 Function Documentation

#### 26.5.1 static void PMC0\_ConfigureHsrunMode ( const pmc0\_hsrn\_mode\_config\_t \* config ) [inline], [static]

This function configures the HSRUN power mode, including the core regulator voltage Level setting, enable forward bias or not.

Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>config</i> | Low-Voltage detect configuration structure. |
|---------------|---------------------------------------------|

#### 26.5.2 static void PMC0\_ConfigureRunMode ( const pmc0\_run\_mode\_config\_t \* config ) [inline], [static]

This function configures the RUN power mode, including the core regulator voltage Level setting.

Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>config</i> | Low-Voltage detect configuration structure. |
|---------------|---------------------------------------------|

### 26.5.3 static void PMC0\_ConfigureVlprMode ( const pmc0\_vlpr\_mode\_config\_t \* *config* ) [inline], [static]

This function configures the VLPR power mode, including the core regulator voltage Level setting, enable reverse back bias or not, turn on force HP band-gap or not, select of HVD/LVD monitor and select of core/array regulator.

Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>config</i> | Low-Voltage detect configuration structure. |
|---------------|---------------------------------------------|

### 26.5.4 static void PMC0\_ConfigureStopMode ( const pmc0\_stop\_mode\_config\_t \* *config* ) [inline], [static]

This function configures the STOP power mode, including the core regulator voltage Level setting.

Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>config</i> | Low-Voltage detect configuration structure. |
|---------------|---------------------------------------------|

### 26.5.5 static void PMC0\_ConfigureVlpsMode ( const pmc0\_vlps\_mode\_config\_t \* *config* ) [inline], [static]

This function configures the VLPS power mode, including the core regulator voltage Level setting, enable reverse back bias or not, turn on force HP band-gap or not, select of HVD/LVD monitor and select of core/array regulator.

Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>config</i> | Low-Voltage detect configuration structure. |
|---------------|---------------------------------------------|

### 26.5.6 static void PMC0\_ConfigureLlsMode ( const pmc0\_lls\_mode\_config\_t \* *config* ) [inline], [static]

This function configures the LLS power mode, including the core regulator voltage Level setting, enable reverse back bias or not, turn on force HP band-gap or not, select of HVD/LVD monitor and select of

core/array regulator.

## Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>config</i> | Low-Voltage detect configuration structure. |
|---------------|---------------------------------------------|

**26.5.7 static void PMC0\_ConfigureVllsMode ( const pmc0\_vlls\_mode\_config\_t \* config ) [inline], [static]**

This function configures the VLLS power mode, including turn on force HP band-gap or not, select of HVD/LVD monitor and select of core/array regulator.

The select of array regulator is different from the other mode configurations. PMC 0 VLLS config has three array regulator select options where the others have only the latter two, see [pmc0\\_vlls\\_array\\_regulator\\_select\\_t](#). Three array regulator select options in PMC 0 VLLS config are shown below:

- Regulator is off (diffrentiator)
- LP Regulator is on
- HP Regulator is on

## Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>config</i> | Low-Voltage detect configuration structure. |
|---------------|---------------------------------------------|

**26.5.8 static uint32\_t PMC0\_GetPMC0PowerModeStatusFlags ( void ) [inline], [static]**

```
* if(kPMCO_HSRUNModeStatusFlags ==
* PMCO_GetPMC0PowerModeStatusFlags(void))
* {
* ...
* }
```

## Returns

PMC 0 current power mode status flags in the \_pmc0\_power\_mode\_status\_flags.

**26.5.9 static bool PMC0\_GetPMC0PowerTransitionStatus ( void ) [inline], [static]**

## Returns

If return 'true', which means PMC 0 is in a power mode transition. If return 'false', which means PMC 0 is not in a power mode transition.

### 26.5.10 static uint32\_t PMC0\_GetPMC1PowerModeStatusFlags ( void ) [inline], [static]

```
* if(kPMC0_HSRUNModeStatusFlags ==
* PMC0_GetPMC1PowerModeStatusFlags(void))
* {
* ...
* }
```

Returns

PMC 1 current power mode status flags in the \_pmc0\_power\_mode\_status\_flags.

### 26.5.11 static bool PMC0\_GetPMC1PowerTransitionStatus ( void ) [inline], [static]

Returns

If return 'true', which means PMC 1 is in a power mode transition. If return 'false', which means PMC 1 is not in a power mode transition.

### 26.5.12 static uint32\_t PMC0\_GetStatusFlags ( void ) [inline], [static]

This function gets all PMC 0 status flags. The flags are returned as the logical OR value of the enumerators [\\_pmc0\\_status\\_flags](#). To check for a specific status, compare the return value with enumerators in the [\\_pmc0\\_status\\_flags](#). For example, to check whether core regulator voltage level is changing:

```
* if (kPMC0_CoreRegulatorVoltLevelFlag &
* PMC0_GetStatusFlags(void))
* {
* ...
* }
```

Returns

PMC 0 status flags which are ORed by the enumerators in the [\\_pmc0\\_status\\_flags](#).

### 26.5.13 static void PMC0\_EnableLowVoltDetectInterrupt ( void ) [inline], [static]

This function enables the 1.2V Low-Voltage Detector interrupt.

**26.5.14 static void PMC0\_DisableLowVoltDetectInterrupt( void ) [inline],  
[static]**

This function disables the 1.2V Low-Voltage Detector interrupt.

**26.5.15 static void PMC0\_ClearLowVoltDetectFlag( void ) [inline],  
[static]**

This function enables the 1.2V Low-Voltage Detector flag.

**26.5.16 static void PMC0\_EnableHighVoltDetectInterrupt( void ) [inline],  
[static]**

This function enables the 1.8V High-Voltage Detector interrupt.

**26.5.17 static void PMC0\_DisableHighVoltDetectInterrupt( void ) [inline],  
[static]**

This function disables the 1.8V High-Voltage Detector interrupt.

**26.5.18 static void PMC0\_ClearHighVoltDetectFlag( void ) [inline],  
[static]**

This function enables the 1.8V High-Voltage Detector flag.

**26.5.19 static void PMC0\_EnableLowVoltDetectReset( bool enable ) [inline],  
[static]**

This function enables 1.2V Low-Voltage Detector reset.

Parameters

|               |                                                                                                 |
|---------------|-------------------------------------------------------------------------------------------------|
| <i>enable</i> | Switcher of 1.2V Low-Voltage Detector reset feature. "true" means to enable, "false" means not. |
|---------------|-------------------------------------------------------------------------------------------------|

**26.5.20 static void PMC0\_EnableHighVoltDetectReset( bool enable ) [inline],  
[static]**

This function enables 1.8V High-Voltage Detector reset.

Parameters

|               |                                                                                                  |
|---------------|--------------------------------------------------------------------------------------------------|
| <i>enable</i> | Switcher of 1.8V High-Voltage Detector reset feature. "true" means to enable, "false" means not. |
|---------------|--------------------------------------------------------------------------------------------------|

### 26.5.21 static void PMC0\_ClearPadsIsolation( void ) [inline], [static]

This function releases/clears the isolation in the PADS.

The isolation in the pads only will be asserted during LLS/VLLS. On LLS exit, the isolation will release automatically. ISOACK must be set after a VLLS to RUN mode transition has completed.

### 26.5.22 static void PMC0\_PowerOnPmc1( void ) [inline], [static]

This function powers on PMC 1.

When this bit field is asserted the PMC 1 is powered on. This bit would take action only once. This bit will be rearmed after a POR event only. NOTE: USB PHY-related interrupt (NVIC/GIC) and wake-up channels (AWIC/WKPU) must be disabled before turning PMC1 on.

### 26.5.23 static void PMC0\_EnableWaitLdoOkSignal( bool *enable* ) [inline], [static]

This function enables to wait LDO OK signal.

Parameters

|               |                                                                                    |
|---------------|------------------------------------------------------------------------------------|
| <i>enable</i> | Switcher of wait LDO OK signal feature. "true" means to enable, "false" means not. |
|---------------|------------------------------------------------------------------------------------|

### 26.5.24 static void PMC0\_EnablePmc1LdoRegulator( bool *enable* ) [inline], [static]

This function enables PMC 1 LDO Regulator.

Parameters

|               |                                                                             |
|---------------|-----------------------------------------------------------------------------|
| <i>enable</i> | Switcher of PMC 1 LDO Regulator. "true" means to enable, "false" means not. |
|---------------|-----------------------------------------------------------------------------|

### 26.5.25 static void PMC0\_EnablePmc1RBBMode ( bool *enable* ) [inline], [static]

This function enables PMC1 RBB mode. Since this circuit when enabled has current consumption. It is recommended to use it just in high temperatures when the leakage reduction is much higher than the current consumption.

Parameters

|               |                                                                           |
|---------------|---------------------------------------------------------------------------|
| <i>enable</i> | Switcher of PMC1 RBB mode. "true" means to enable, "false" means disable. |
|---------------|---------------------------------------------------------------------------|

### 26.5.26 static void PMC0\_SetBiasConfig ( const pmc0\_bias\_config\_t \* *config* ) [inline], [static]

This function change the RBB&FBB voltage level and RBB pull-down.

Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>config</i> | PMC 0 bias configuration structure. |
|---------------|-------------------------------------|

### 26.5.27 static void PMC0\_ConfigureSramBankPowerDown ( uint32\_t *bankMask* ) [inline], [static]

This function configures PMC 0 SRAM bank power down.

The bit i controls the power mode of the PMC 0 SRAM bank i. PMC0\_SRAM\_PD[i] = 1'b0 - PMC 0 SRAM bank i is not affected. PMC0\_SRAM\_PD[i] = 1'b1 - PMC 0 SRAM bank i is in ASD or ARRAY\_SHUTDOWN during all modes, except VLLS. During VLLS is in POWER\_DOWN mode.

Example

```
* // Enable band 0 and 1 in ASD or ARRAY_SHUTDOWN during all modes except VLLS
* PMC0_ConfigureSramBankPowerDown(0x3U);
*
```

## Parameters

|                 |                                                                      |
|-----------------|----------------------------------------------------------------------|
| <i>bankMask</i> | The bands to enable. Logical OR of all bits of band index to enable. |
|-----------------|----------------------------------------------------------------------|

### 26.5.28 static void PMC0\_ConfigureSramBankPowerDownStopMode ( uint32\_t *bankMask* ) [inline], [static]

This function configures PMC 0 SRAM bank power down in stop modes.

The bit i controls the PMC 0 SRAM bank i. PMC0\_SRAM\_PDS[i] = 1'b0 - PMC 0 SRAM bank i is not affected. PMC0\_SRAM\_PDS[i] = 1'b1 - PMC 0 SRAM bank i is in ASD or ARRAY\_SHUTDOWN mode during STOP, VLPS and LLS modes. During VLLS is in POWER\_DOWN mode.

## Example

```
* // Enable band 0 and 1 in ASD or ARRAY_SHUTDOWN during STOP, VLPS and LLS modes
* PMC0_ConfigureSramBankPowerDownStopMode(0x3U);
*
```

## Parameters

|                 |                                                                      |
|-----------------|----------------------------------------------------------------------|
| <i>bankMask</i> | The bands to enable. Logical OR of all bits of band index to enable. |
|-----------------|----------------------------------------------------------------------|

### 26.5.29 static void PMC0\_ConfigureSramBankPowerDownStandbyMode ( uint32\_t *bankMask* ) [inline], [static]

This function configures PMC 0 SRAM bank power down in Standby Mode.

The bit i controls the PMC 0 SRAM bank i. PMC0\_SRAM\_STDY[i] = 1'b0 - PMC 0 SRAM bank i is not affected. PMC0\_SRAM\_STDY[i] = 1'b1 - PMC 0 SRAM bank i is in STANDBY mode during all modes (except VLLS and LLS).

## Example

```
* // Enable band 0 and 1 in STANDBY mode except VLLS and LLS
* PMC0_ConfigureSramBankPowerDownStandbyMode(0x3U);
*
```

## Parameters

|                 |                                                                      |
|-----------------|----------------------------------------------------------------------|
| <i>bankMask</i> | The bands to enable. Logical OR of all bits of band index to enable. |
|-----------------|----------------------------------------------------------------------|

### 26.5.30 static void PMC0\_EnableTemperatureSensor ( bool *enable* ) [inline], [static]

Parameters

|               |                                                                                                                                                                                                                   |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>enable</i> | Used to enable/disable internal temperature sensor. <ul style="list-style-type: none"><li>• <b>true</b> Enable internal temperature sensor.</li><li>• <b>false</b> Disable internal temperature sensor.</li></ul> |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 26.5.31 static void PMC0\_SetTemperatureSensorMode ( `uint8_t mode` ) [`inline`], [`static`]

Parameters

|             |                                     |
|-------------|-------------------------------------|
| <i>mode</i> | The temperature sensor mode to set. |
|-------------|-------------------------------------|

# Chapter 27

## PORT: Port Control and Interrupts

### 27.1 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_MuxAlt0 = 0U,
kPORT_MuxAlt1 = 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, 4, 1)`)  
*PORT driver version.*

## Configuration

- static void `PORT_SetPinConfig` (PORT\_Type \*base, uint32\_t pin, const `port_pin_config_t` \*config)  
*Sets the port PCR register.*
- static void `PORT_SetMultiplePinsConfig` (PORT\_Type \*base, uint32\_t mask, const `port_pin_config_t` \*config)  
*Sets the port PCR register for multiple pins.*
- static void `PORT_SetPinMux` (PORT\_Type \*base, uint32\_t pin, `port_mux_t` mux)  
*Configures the pin muxing.*
- static void `PORT_EnablePinsDigitalFilter` (PORT\_Type \*base, uint32\_t mask, bool enable)  
*Enables the digital filter in one port, each bit of the 32-bit register represents one pin.*
- static void `PORT_SetDigitalFilterConfig` (PORT\_Type \*base, const `port_digital_filter_config_t` \*config)  
*Sets the digital filter in one port, each bit of the 32-bit register represents one pin.*

## Interrupt

- static void `PORT_SetPinInterruptConfig` (PORT\_Type \*base, uint32\_t pin, `port_interrupt_t` config)  
*Configures the port pin interrupt/DMA request.*
- static 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.*

## 27.2 Data Structure Documentation

### 27.2.1 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.*

### 27.2.2 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].*

## 27.3 Macro Definition Documentation

### 27.3.1 #define FSL\_PORT\_DRIVER\_VERSION (MAKE\_VERSION(2, 4, 1))

## 27.4 Enumeration Type Documentation

### 27.4.1 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.

### 27.4.2 enum \_port\_passive\_filter\_enable

Enumerator

***kPORT\_PassiveFilterDisable*** Passive input filter is disabled.

***kPORT\_PassiveFilterEnable*** Passive input filter is enabled.

### 27.4.3 enum \_port\_drive\_strength

Enumerator

***kPORT\_LowDriveStrength*** Low-drive strength is configured.

***kPORT\_HighDriveStrength*** High-drive strength is configured.

### 27.4.4 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.

## 27.4.5 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\_MuxAlt0*** Chip-specific.

***kPORT\_MuxAlt1*** Chip-specific.

***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.

## 27.4.6 enum port\_interrupt\_t

Enumerator

***kPORT\_InterruptOrDMA.Disabled*** Interrupt/DMA request is disabled.

***kPORT\_DMA.RisingEdge*** DMA request on rising edge.

***kPORT\_DMA.FallingEdge*** DMA request on falling edge.

***kPORT\_DMA.EitherEdge*** DMA request on either edge.

***kPORT\_Flag.RisingEdge*** Flag sets on rising edge.

***kPORT\_Flag.FallingEdge*** Flag sets on falling edge.

***kPORT\_Flag.EitherEdge*** Flag sets on either edge.

***kPORT\_Interrupt.LogicZero*** Interrupt when logic zero.

***kPORT\_Interrupt.RisingEdge*** Interrupt on rising edge.

***kPORT\_Interrupt.FallingEdge*** Interrupt on falling edge.

***kPORT\_Interrupt.EitherEdge*** Interrupt on either edge.

***kPORT\_Interrupt.LogicOne*** Interrupt when logic one.

***kPORT\_ActiveHighTriggerOutputEnable*** Enable active high-trigger output.

***kPORT\_ActiveLowTriggerOutputEnable*** Enable active low-trigger output.

### 27.4.7 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.

## 27.5 Function Documentation

### 27.5.1 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. |

### 27.5.2 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. |

### 27.5.3 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"> <li>• <a href="#">kPORT_PinDisabledOrAnalog</a>: Pin disabled or work in analog function.</li> <li>• <a href="#">kPORT_MuxAsGpio</a> : Set as GPIO.</li> <li>• <a href="#">kPORT_MuxAlt2</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt3</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt4</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt5</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt6</a> : chip-specific.</li> <li>• <a href="#">kPORT_MuxAlt7</a> : chip-specific.</li> </ul> |

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

### 27.5.4 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. |

### 27.5.5 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. |

### 27.5.6 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> | <p>PORT pin interrupt configuration.</p> <ul style="list-style-type: none"> <li>• <a href="#">kPORT_InterruptOrDMADisabled</a>: Interrupt/DMA request disabled.</li> <li>• <a href="#">kPORT_DMARisingEdge</a> : DMA request on rising edge(if the DMA requests exit).</li> <li>• <a href="#">kPORT_DMAPFallingEdge</a>: DMA request on falling edge(if the DMA requests exit).</li> <li>• <a href="#">kPORT_DMAEitherEdge</a> : DMA request on either edge(if the DMA requests exit).</li> <li>• <a href="#">kPORT_FlagRisingEdge</a> : Flag sets on rising edge(if the Flag states exit).</li> <li>• <a href="#">kPORT_FlagFallingEdge</a> : Flag sets on falling edge(if the Flag states exit).</li> <li>• <a href="#">kPORT_FlagEitherEdge</a> : Flag sets on either edge(if the Flag states exit).</li> <li>• <a href="#">kPORT_InterruptLogicZero</a> : Interrupt when logic zero.</li> <li>• <a href="#">kPORT_InterruptRisingEdge</a> : Interrupt on rising edge.</li> <li>• <a href="#">kPORT_InterruptFallingEdge</a>: Interrupt on falling edge.</li> <li>• <a href="#">kPORT_InterruptEitherEdge</a> : Interrupt on either edge.</li> <li>• <a href="#">kPORT_InterruptLogicOne</a> : Interrupt when logic one.</li> <li>• <a href="#">kPORT_ActiveHighTriggerOutputEnable</a> : Enable active high-trigger output (if the trigger states exit).</li> <li>• <a href="#">kPORT_ActiveLowTriggerOutputEnable</a> : Enable active low-trigger output (if the trigger states exit).</li> </ul> |

### 27.5.7 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"> <li>• <a href="#">kPORT_LowDriveStrength</a> = 0U - Low-drive strength is configured.</li> <li>• <a href="#">kPORT_HighDriveStrength</a> = 1U - High-drive strength is configured.</li> </ul> |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 27.5.8 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.

### 27.5.9 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 28

## QSPI: Quad Serial Peripheral Interface

### 28.1 Overview

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

QSPI driver includes functional APIs and EDMA transactional APIs.

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

Transactional APIs are transaction target high level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the `qspi_handle_t` as the first parameter. Initialize the handle by calling the [QSPI\\_TransferTxCreateHandleEDMA\(\)](#) or [QSPI\\_TransferRxCreateHandleEDMA\(\)](#) API.

### Modules

- [Quad Serial Peripheral Interface Driver](#)
- [Quad Serial Peripheral Interface EDMA Driver](#)

## 28.2 Quad Serial Peripheral Interface Driver

### 28.2.1 Overview

#### Data Structures

- struct `qspi_dqs_config_t`  
*DQS configure features. [More...](#)*
- struct `qspi_flash_timing_t`  
*Flash timing configuration. [More...](#)*
- struct `qspi_config_t`  
*QSPI configuration structure. [More...](#)*
- struct `qspi_flash_config_t`  
*External flash configuration items. [More...](#)*
- struct `qspi_transfer_t`  
*Transfer structure for QSPI. [More...](#)*
- struct `ip_command_config_t`  
*16-bit access reg for IPCR register [More...](#)*

#### Macros

- #define `QSPI_LUT_SEQ`(cmd0, pad0, op0, cmd1, pad1, op1)  
*Macro functions for LUT table.*
- #define `QSPI_CMD` (0x1U)  
*Macro for QSPI LUT command.*
- #define `QSPI_PAD_1` (0x0U)  
*Macro for QSPI PAD.*

#### Enumerations

- enum {
   
  `kStatus_QSPI_Idle` = MAKE\_STATUS(kStatusGroup\_QSPI, 0),
   
  `kStatus_QSPI_Busy` = MAKE\_STATUS(kStatusGroup\_QSPI, 1),
   
  `kStatus_QSPI_Error` = MAKE\_STATUS(kStatusGroup\_QSPI, 2) }
   
*Status structure of QSPI.*
- enum `qspi_read_area_t` {
   
  `kQSPI_ReadAHB` = 0x0U,
   
  `kQSPI_ReadIP` }
   
*QSPI read data area, from IP FIFO or AHB buffer.*
- enum `qspi_command_seq_t` {
   
  `kQSPI_IPSeq` = QuadSPI\_SPTRCLR\_IPPTRC\_MASK,
   
  `kQSPI_BufferSeq` = QuadSPI\_SPTRCLR\_BFPTRC\_MASK }
   
*QSPI command sequence type.*
- enum `qspi_fifo_t` {
   
  `kQSPI_TxFifo` = QuadSPI\_MCR\_CLR\_TXF\_MASK,
   
  `kQSPI_RxFifo` = QuadSPI\_MCR\_CLR\_RXF\_MASK,
   
  `kQSPI_AllFifo` = QuadSPI\_MCR\_CLR\_TXF\_MASK | QuadSPI\_MCR\_CLR\_RXF\_MASK }

- *QSPI buffer type.*  
• enum `qspi_endianness_t` {  
  kQSPI\_64BigEndian = 0x0U,  
  kQSPI\_32LittleEndian,  
  kQSPI\_32BigEndian,  
  kQSPI\_64LittleEndian }  
*QSPI transfer endianess.*
- enum `_qspi_error_flags` {  
  kQSPI\_DataLearningFail = (int)QuadSPI\_FR\_DLPFF\_MASK,  
  kQSPI\_TxBufferFill = QuadSPI\_FR\_TBFF\_MASK,  
  kQSPI\_TxBufferUnderrun = QuadSPI\_FR\_TBUF\_MASK,  
  kQSPI\_IllegalInstruction = QuadSPI\_FR\_ILLINE\_MASK,  
  kQSPI\_RxBufferOverflow = QuadSPI\_FR\_RBOF\_MASK,  
  kQSPI\_RxBufferDrain = QuadSPI\_FR\_RBDF\_MASK,  
  kQSPI\_AHBSquenceError = QuadSPI\_FR\_ABSEF\_MASK,  
  kQSPI\_AHBIllegalTransaction = QuadSPI\_FR\_AITEF\_MASK,  
  kQSPI\_AHBIllegalBurstSize = QuadSPI\_FR\_AIBSEF\_MASK,  
  kQSPI\_AHBBufferOverflow = QuadSPI\_FR\_ABOF\_MASK,  
  kQSPI\_IPCommandTriggerDuringAHBAccess = QuadSPI\_FR\_IPAEF\_MASK,  
  kQSPI\_IPCommandTriggerDuringIPAccess = QuadSPI\_FR\_IPIEF\_MASK,  
  kQSPI\_IPCommandTriggerDuringAHBGrant = QuadSPI\_FR\_IPGEF\_MASK,  
  kQSPI\_IPCommandTransactionFinished = QuadSPI\_FR\_TFF\_MASK,  
  kQSPI\_FlagAll = (int)0x8C83F8D1U }
- *QSPI error flags.*  
• enum `_qspi_flags` {

```

kQSPI_DataLearningSamplePoint = (int)QuadSPI_SR_DLPSMP_MASK,
kQSPI_TxBufferFull = QuadSPI_SR_TXFULL_MASK,
kQSPI_TxDMA = QuadSPI_SR_TXDMA_MASK,
kQSPI_TxWatermark = QuadSPI_SR_TXWA_MASK,
kQSPI_TxBufferEnoughData = QuadSPI_SR_TXEDA_MASK,
kQSPI_RxDMA = QuadSPI_SR_RXDMA_MASK,
kQSPI_RxBufferFull = QuadSPI_SR_RXFULL_MASK,
kQSPI_RxWatermark = QuadSPI_SR_RXWE_MASK,
kQSPI_AHB3BufferFull = QuadSPI_SR_AHB3FUL_MASK,
kQSPI_AHB2BufferFull = QuadSPI_SR_AHB2FUL_MASK,
kQSPI_AHB1BufferFull = QuadSPI_SR_AHB1FUL_MASK,
kQSPI_AHB0BufferFull = QuadSPI_SR_AHB0FUL_MASK,
kQSPI_AHB3BufferNotEmpty = QuadSPI_SR_AHB3NE_MASK,
kQSPI_AHB2BufferNotEmpty = QuadSPI_SR_AHB2NE_MASK,
kQSPI_AHB1BufferNotEmpty = QuadSPI_SR_AHB1NE_MASK,
kQSPI_AHB0BufferNotEmpty = QuadSPI_SR_AHB0NE_MASK,
kQSPI_AHBTransactionPending = QuadSPI_SR_AHBTRN_MASK,
kQSPI_AHBCCommandPriorityGranted = QuadSPI_SR_AHBGNT_MASK,
kQSPI_AHBAccess = QuadSPI_SR_AHB_ACC_MASK,
kQSPI_IPAccess = QuadSPI_SR_IP_ACC_MASK,
kQSPI_Busy = QuadSPI_SR_BUSY_MASK,
kQSPI_StateAll = (int)0xEF897FE7U }

```

*QSPI state bit.*

- enum `_qspi_interrupt_enable` {
 

```

kQSPI_DataLearningFailInterruptEnable,
kQSPI_TxBufferFillInterruptEnable = QuadSPI_RSER_TBFIE_MASK,
kQSPI_TxBufferUnderrunInterruptEnable = QuadSPI_RSER_TBUIE_MASK,
kQSPI_IllegalInstructionInterruptEnable,
kQSPI_RxBufferOverflowInterruptEnable = QuadSPI_RSER_RBOIE_MASK,
kQSPI_RxBufferDrainInterruptEnable = QuadSPI_RSER_RBDIE_MASK,
kQSPI_AHBSequenceErrorInterruptEnable = QuadSPI_RSER_ABSEIE_MASK,
kQSPI_AHBIlegalTransactionInterruptEnable,
kQSPI_AHBIlegalBurstSizeInterruptEnable,
kQSPI_AHBBufferOverflowInterruptEnable = QuadSPI_RSER_ABOIE_MASK,
kQSPI_IPCommandTriggerDuringAHBAccessInterruptEnable,
kQSPI_IPCommandTriggerDuringIPAccessInterruptEnable,
kQSPI_IPCommandTriggerDuringAHBGrantInterruptEnable,
kQSPI_IPCommandTransactionFinishedInterruptEnable,
kQSPI_AllInterruptEnable = (int)0x8C83F8D1U }

```

*QSPI interrupt enable.*

- enum `_qspi_dma_enable` {
 

```

kQSPI_TxBufferFillDMAEnable = QuadSPI_RSER_TBFDE_MASK,
kQSPI_RxBufferDrainDMAEnable = QuadSPI_RSER_RBDDE_MASK,
kQSPI_AlIIDDMAEnable = QuadSPI_RSER_TBFDE_MASK | QuadSPI_RSER_RBDDE_MASK
}

```

- *QSPI DMA request flag.*  
enum `qspi_dqs_phrase_shift_t` {  
  kQSPI\_DQSNoPhraseShift = 0x0U,  
  kQSPI\_DQSPhraseShift45Degree,  
  kQSPI\_DQSPhraseShift90Degree,  
  kQSPI\_DQSPhraseShift135Degree }
  - *Phrase shift number for DQS mode.*  
enum `qspi_dqs_read_sample_clock_t` {  
  kQSPI\_ReadSampleClkInternalLoopback = 0x0U,  
  kQSPI\_ReadSampleClkLoopbackFromDqsPad = 0x1U,  
  kQSPI\_ReadSampleClkExternalInputFromDqsPad = 0x2U }
- Qspi read sampling option.*

## Driver version

- #define `FSL_QSPI_DRIVER_VERSION` (MAKE\_VERSION(2, 2, 3))  
*QSPI driver version 2.2.3.*

## Initialization and deinitialization

- `uint32_t QSPIGetInstance (QuadSPI_Type *base)`  
*Get the instance number for QSPI.*
- `void QSPI_Init (QuadSPI_Type *base, qspi_config_t *config, uint32_t srcClock_Hz)`  
*Initializes the QSPI module and internal state.*
- `void QSPI_GetDefaultQspiConfig (qspi_config_t *config)`  
*Gets default settings for QSPI.*
- `void QSPI_Deinit (QuadSPI_Type *base)`  
*Deinitializes the QSPI module.*
- `void QSPI_SetFlashConfig (QuadSPI_Type *base, qspi_flash_config_t *config)`  
*Configures the serial flash parameter.*
- `void QSPI_SetDqsConfig (QuadSPI_Type *base, qspi_dqs_config_t *config)`  
*Configures the serial flash DQS parameter.*
- `void QSPI_SoftwareReset (QuadSPI_Type *base)`  
*Software reset for the QSPI logic.*
- `static void QSPI_Enable (QuadSPI_Type *base, bool enable)`  
*Enables or disables the QSPI module.*

## Status

- `static uint32_t QSPI_GetStatusFlags (QuadSPI_Type *base)`  
*Gets the state value of QSPI.*
- `static uint32_t QSPI_GetErrorStatusFlags (QuadSPI_Type *base)`  
*Gets QSPI error status flags.*
- `static void QSPI_ClearErrorFlag (QuadSPI_Type *base, uint32_t mask)`  
*Clears the QSPI error flags.*

## Interrupts

- static void [QSPI\\_EnableInterrupts](#) (QuadSPI\_Type \*base, uint32\_t mask)  
*Enables the QSPI interrupts.*
- static void [QSPI\\_DisableInterrupts](#) (QuadSPI\_Type \*base, uint32\_t mask)  
*Disables the QSPI interrupts.*

## DMA Control

- static void [QSPI\\_EnableDMA](#) (QuadSPI\_Type \*base, uint32\_t mask, bool enable)  
*Enables the QSPI DMA source.*
- static uint32\_t [QSPI\\_GetTxDataRegisterAddress](#) (QuadSPI\_Type \*base)  
*Gets the Tx data register address.*
- uint32\_t [QSPI\\_GetRxDataRegisterAddress](#) (QuadSPI\_Type \*base)  
*Gets the Rx data register address used for DMA operation.*

## Bus Operations

- static void [QSPI\\_SetIPCommandAddress](#) (QuadSPI\_Type \*base, uint32\_t addr)  
*Sets the IP command address.*
- static void [QSPI\\_SetIPCommandSize](#) (QuadSPI\_Type \*base, uint32\_t size)  
*Sets the IP command size.*
- void [QSPI\\_ExecuteIPCommand](#) (QuadSPI\_Type \*base, uint32\_t index)  
*Executes IP commands located in LUT table.*
- void [QSPI\\_ExecuteAHBCommand](#) (QuadSPI\_Type \*base, uint32\_t index)  
*Executes AHB commands located in LUT table.*
- void [QSPI\\_UpdateLUT](#) (QuadSPI\_Type \*base, uint32\_t index, uint32\_t \*cmd)  
*Updates the LUT table.*
- static void [QSPI\\_ClearFifo](#) (QuadSPI\_Type \*base, uint32\_t mask)  
*Clears the QSPI FIFO logic.*
- static void [QSPI\\_ClearCommandSequence](#) (QuadSPI\_Type \*base, [qspi\\_command\\_seq\\_t](#) seq)  
*@ brief Clears the command sequence for the IP/buffer command.*
- static void [QSPI\\_EnableDDRMode](#) (QuadSPI\_Type \*base, bool enable)  
*Enable or disable DDR mode.*
- void [QSPI\\_SetReadDataArea](#) (QuadSPI\_Type \*base, [qspi\\_read\\_area\\_t](#) area)  
*@ brief Set the RX buffer readout area.*
- void [QSPI\\_WriteBlocking](#) (QuadSPI\_Type \*base, uint32\_t \*buffer, size\_t size)  
*Sends a buffer of data bytes using a blocking method.*
- static void [QSPI\\_WriteData](#) (QuadSPI\_Type \*base, uint32\_t data)  
*Writes data into FIFO.*
- void [QSPI\\_ReadBlocking](#) (QuadSPI\_Type \*base, uint32\_t \*buffer, size\_t size)  
*Receives a buffer of data bytes using a blocking method.*
- uint32\_t [QSPI\\_ReadData](#) (QuadSPI\_Type \*base)  
*Receives data from data FIFO.*

## Transactional

- static void [QSPI\\_TransferSendBlocking](#) (QuadSPI\_Type \*base, [qspi\\_transfer\\_t](#) \*xfer)  
*Writes data to the QSPI transmit buffer.*
- static void [QSPI\\_TransferReceiveBlocking](#) (QuadSPI\_Type \*base, [qspi\\_transfer\\_t](#) \*xfer)  
*Reads data from the QSPI receive buffer in polling way.*

## 28.2.2 Data Structure Documentation

### 28.2.2.1 struct qspi\_dqs\_config\_t

#### Data Fields

- uint32\_t [portADelayTapNum](#)  
*Delay chain tap number selection for QSPI port A DQS.*
- [qspi\\_dqs\\_phrase\\_shift\\_t](#) [shift](#)  
*Phase shift for internal DQS generation.*
- [qspi\\_dqs\\_read\\_sample\\_clock\\_t](#) [rxSampleClock](#)  
*Read sample clock for Dqs.*
- bool [enableDQSClkInverse](#)  
*Enable inverse clock for internal DQS generation.*

#### Field Documentation

(1) [qspi\\_dqs\\_read\\_sample\\_clock\\_t](#) [qspi\\_dqs\\_config\\_t::rxSampleClock](#)

### 28.2.2.2 struct qspi\_flash\_timing\_t

#### Data Fields

- uint32\_t [dataHoldTime](#)  
*Serial flash data in hold time.*
- uint32\_t [CSHoldTime](#)  
*Serial flash CS hold time in terms of serial flash clock cycles.*
- uint32\_t [CSSetupTime](#)  
*Serial flash CS setup time in terms of serial flash clock cycles.*

### 28.2.2.3 struct qspi\_config\_t

#### Data Fields

- uint32\_t [clockSource](#)  
*Clock source for QSPI module.*
- uint32\_t [baudRate](#)  
*Serial flash clock baud rate.*
- uint8\_t [txWatermark](#)  
*QSPI transmit watermark value.*
- uint8\_t [rxWatermark](#)

*QSPI receive watermark value.*

- `uint32_t AHBbufferSize` [FSL\_FEATURE\_QSPI\_AHB\_BUFFER\_COUNT]  
*AHB buffer size.*
- `uint8_t AHBbufferMaster` [FSL\_FEATURE\_QSPI\_AHB\_BUFFER\_COUNT]  
*AHB buffer master.*
- `bool enableAHBbuffer3AllMaster`  
*Is AHB buffer3 for all master.*
- `qspi_read_area_t area`  
*Which area Rx data readout.*
- `bool enableQspi`  
*Enable QSPI after initialization.*

## Field Documentation

(1) `uint8_t qspi_config_t::rxWatermark`

(2) `uint32_t qspi_config_t::AHBbufferSize[FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]`

(3) `uint8_t qspi_config_t::AHBbufferMaster[FSL_FEATURE_QSPI_AHB_BUFFER_COUNT]`

(4) `bool qspi_config_t::enableAHBbuffer3AllMaster`

### 28.2.2.4 struct qspi\_flash\_config\_t

#### Data Fields

- `uint32_t flashA1Size`  
*Flash A1 size.*
- `uint32_t flashA2Size`  
*Flash A2 size.*
- `uint32_t lookuptable` [FSL\_FEATURE\_QSPI\_LUT\_DEPTH]  
*Flash command in LUT.*
- `uint32_t dataHoldTime`  
*Data line hold time.*
- `uint32_t CSHoldTime`  
*CS line hold time.*
- `uint32_t CSSetupTime`  
*CS line setup time.*
- `uint32_t columnSpace`  
*Column space size.*
- `uint32_t dataLearnValue`  
*Data Learn value if enable data learn.*
- `qspi_endianness_t endian`  
*Flash data endianness.*
- `bool enableWordAddress`  
*If enable word address.*

**Field Documentation**

- (1) `uint32_t qspi_flash_config_t::dataHoldTime`
- (2) `qspi_endianness_t qspi_flash_config_t::endian`
- (3) `bool qspi_flash_config_t::enableWordAddress`

**28.2.2.5 struct qspi\_transfer\_t****Data Fields**

- `uint32_t * data`  
*Pointer to data to transmit.*
- `size_t dataSize`  
*Bytes to be transmit.*

**28.2.2.6 struct ip\_command\_config\_t****28.2.3 Macro Definition Documentation****28.2.3.1 #define FSL\_QSPI\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 3))****28.2.4 Enumeration Type Documentation****28.2.4.1 anonymous enum**

Enumerator

- kStatus\_QSPI\_Idle* QSPI is in idle state.
- kStatus\_QSPI\_Busy* QSPI is busy.
- kStatus\_QSPI\_Error* Error occurred during QSPI transfer.

**28.2.4.2 enum qspi\_read\_area\_t**

Enumerator

- kQSPI\_ReadAHB* QSPI read from AHB buffer.
- kQSPI\_ReadIP* QSPI read from IP FIFO.

**28.2.4.3 enum qspi\_command\_seq\_t**

Enumerator

- kQSPI\_IPSeq* IP command sequence.

*kQSPI\_BufferSeq* Buffer command sequence.

#### 28.2.4.4 enum qspi\_fifo\_t

Enumerator

*kQSPI\_TxFifo* QSPI Tx FIFO.

*kQSPI\_RxFifo* QSPI Rx FIFO.

*kQSPI\_AllFifo* QSPI all FIFO, including Tx and Rx.

#### 28.2.4.5 enum qspi\_endianness\_t

Enumerator

*kQSPI\_64BigEndian* 64 bits big endian

*kQSPI\_32LittleEndian* 32 bit little endian

*kQSPI\_32BigEndian* 32 bit big endian

*kQSPI\_64LittleEndian* 64 bit little endian

#### 28.2.4.6 enum \_qspi\_error\_flags

Enumerator

*kQSPI\_DataLearningFail* Data learning pattern failure flag.

*kQSPI\_TxBufferFill* Tx buffer fill flag.

*kQSPI\_TxBufferUnderrun* Tx buffer underrun flag.

*kQSPI\_IllegalInstruction* Illegal instruction error flag.

*kQSPI\_RxBufferOverflow* Rx buffer overflow flag.

*kQSPI\_RxBufferDrain* Rx buffer drain flag.

*kQSPI\_AHBSequenceError* AHB sequence error flag.

*kQSPI\_AHBIllegalTransaction* AHB illegal transaction error flag.

*kQSPI\_AHBIllegalBurstSize* AHB illegal burst error flag.

*kQSPI\_AHBBufferOverflow* AHB buffer overflow flag.

*kQSPI\_IPCommandTriggerDuringAHBAccess* IP command trigger during AHB access error.

*kQSPI\_IPCommandTriggerDuringIPAccess* IP command trigger cannot be executed.

*kQSPI\_IPCommandTriggerDuringAHBGrant* IP command trigger during AHB grant error.

*kQSPI\_IPCommandTransactionFinished* IP command transaction finished flag.

*kQSPI\_FlagAll* All error flag.

### 28.2.4.7 enum \_qspi\_flags

Enumerator

*kQSPI\_DataLearningSamplePoint* Data learning sample point.  
*kQSPI\_TxBufferFull* Tx buffer full flag.  
*kQSPI\_TxDMA* Tx DMA is requested or running.  
*kQSPI\_TxWatermark* Tx buffer watermark available.  
*kQSPI\_TxBufferEnoughData* Tx buffer enough data available.  
*kQSPI\_RxDMA* Rx DMA is requesting or running.  
*kQSPI\_RxBufferFull* Rx buffer full.  
*kQSPI\_RxWatermark* Rx buffer watermark exceeded.  
*kQSPI\_AHB3BufferFull* AHB buffer 3 full.  
*kQSPI\_AHB2BufferFull* AHB buffer 2 full.  
*kQSPI\_AHB1BufferFull* AHB buffer 1 full.  
*kQSPI\_AHB0BufferFull* AHB buffer 0 full.  
*kQSPI\_AHB3BufferNotEmpty* AHB buffer 3 not empty.  
*kQSPI\_AHB2BufferNotEmpty* AHB buffer 2 not empty.  
*kQSPI\_AHB1BufferNotEmpty* AHB buffer 1 not empty.  
*kQSPI\_AHB0BufferNotEmpty* AHB buffer 0 not empty.  
*kQSPI\_AHBTransactionPending* AHB access transaction pending.  
*kQSPI\_AHBCCommandPriorityGranted* AHB command priority granted.  
*kQSPI\_AHBAccess* AHB access.  
*kQSPI\_IPAccess* IP access.  
*kQSPI\_Busy* Module busy.  
*kQSPI\_StateAll* All flags.

### 28.2.4.8 enum \_qspi\_interrupt\_enable

Enumerator

*kQSPI\_DataLearningFailInterruptEnable* Data learning pattern failure interrupt enable.  
*kQSPI\_TxBufferFillInterruptEnable* Tx buffer fill interrupt enable.  
*kQSPI\_TxBufferUnderrunInterruptEnable* Tx buffer underrun interrupt enable.  
*kQSPI\_IllegalInstructionInterruptEnable* Illegal instruction error interrupt enable.  
*kQSPI\_RxBufferOverflowInterruptEnable* Rx buffer overflow interrupt enable.  
*kQSPI\_RxBufferDrainInterruptEnable* Rx buffer drain interrupt enable.  
*kQSPI\_AHBSequenceErrorInterruptEnable* AHB sequence error interrupt enable.  
*kQSPI\_AHBIlegalTransactionInterruptEnable* AHB illegal transaction error interrupt enable.  
*kQSPI\_AHBIlegalBurstSizeInterruptEnable* AHB illegal burst error interrupt enable.  
*kQSPI\_AHBBufferOverflowInterruptEnable* AHB buffer overflow interrupt enable.  
*kQSPI\_IPCommandTriggerDuringAHBAccesInterruptEnable* IP command trigger during AHB access error.  
*kQSPI\_IPCommandTriggerDuringIPAccessInterruptEnable* IP command trigger cannot be executed.

*kQSPI\_IPCommandTriggerDuringAHBGrantInterruptEnable* IP command trigger during AHB grant error.

*kQSPI\_IPCommandTransactionFinishedInterruptEnable* IP command transaction finished interrupt enable.

*kQSPI\_AllInterruptEnable* All error interrupt enable.

#### 28.2.4.9 enum \_qspi\_dma\_enable

Enumerator

*kQSPI\_TxBufferFillDMAEnable* Tx buffer fill DMA.

*kQSPI\_RxBufferDrainDMAEnable* Rx buffer drain DMA.

*kQSPI\_AllDDMAEnable* All DMA source.

#### 28.2.4.10 enum qspi\_dqs\_phrase\_shift\_t

Enumerator

*kQSPI\_DQSNoPhraseShift* No phase shift.

*kQSPI\_DQSPhraseShift45Degree* Select 45 degree phase shift.

*kQSPI\_DQSPhraseShift90Degree* Select 90 degree phase shift.

*kQSPI\_DQSPhraseShift135Degree* Select 135 degree phase shift.

#### 28.2.4.11 enum qspi\_dqs\_read\_sample\_clock\_t

Enumerator

*kQSPI\_ReadSampleClkInternalLoopback* Read sample clock adopts internal loopback mode.

*kQSPI\_ReadSampleClkLoopbackFromDqsPad* Dummy Read strobe generated by QSPI Controller and loopback from DQS pad.

*kQSPI\_ReadSampleClkExternalInputFromDqsPad* Flash provided Read strobe and input from D-QS pad.

### 28.2.5 Function Documentation

#### 28.2.5.1 uint32\_t QSPI\_GetInstance ( QuadSPI\_Type \* *base* )

Parameters

|             |                    |
|-------------|--------------------|
| <i>base</i> | QSPI base pointer. |
|-------------|--------------------|

#### 28.2.5.2 void QSPI\_Init ( QuadSPI\_Type \* *base*, qspi\_config\_t \* *config*, uint32\_t *srcClock\_Hz* )

This function enables the clock for QSPI and also configures the QSPI with the input configure parameters. Users should call this function before any QSPI operations.

Parameters

|                    |                                    |
|--------------------|------------------------------------|
| <i>base</i>        | Pointer to QuadSPI Type.           |
| <i>config</i>      | QSPI configure structure.          |
| <i>srcClock_Hz</i> | QSPI source clock frequency in Hz. |

#### 28.2.5.3 void QSPI\_GetDefaultQspiConfig ( qspi\_config\_t \* *config* )

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>config</i> | QSPI configuration structure. |
|---------------|-------------------------------|

#### 28.2.5.4 void QSPI\_Deinit ( QuadSPI\_Type \* *base* )

Clears the QSPI state and QSPI module registers.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
|-------------|--------------------------|

#### 28.2.5.5 void QSPI\_SetFlashConfig ( QuadSPI\_Type \* *base*, qspi\_flash\_config\_t \* *config* )

This function configures the serial flash relevant parameters, such as the size, command, and so on. The flash configuration value cannot have a default value. The user needs to configure it according to the QSPI features.

Parameters

|               |                                 |
|---------------|---------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.        |
| <i>config</i> | Flash configuration parameters. |

#### 28.2.5.6 void QSPI\_SetDqsConfig ( QuadSPI\_Type \* *base*, qspi\_dqs\_config\_t \* *config* )

This function configures the serial flash DQS relevant parameters, such as the delay chain tap number, . DQS shift phase, whether need to inverse and the rxc sample clock selection.

Parameters

|               |                               |
|---------------|-------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.      |
| <i>config</i> | Dqs configuration parameters. |

#### 28.2.5.7 void QSPI\_SoftwareReset ( QuadSPI\_Type \* *base* )

This function sets the software reset flags for both AHB and buffer domain and resets both AHB buffer and also IP FIFOs.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
|-------------|--------------------------|

#### 28.2.5.8 static void QSPI\_Enable ( QuadSPI\_Type \* *base*, bool *enable* ) [inline], [static]

Parameters

|               |                                              |
|---------------|----------------------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.                     |
| <i>enable</i> | True means enable QSPI, false means disable. |

#### 28.2.5.9 static uint32\_t QSPI\_GetStatusFlags ( QuadSPI\_Type \* *base* ) [inline], [static]

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
|-------------|--------------------------|

Returns

status flag, use status flag to AND `_qspi_flags` could get the related status.

#### 28.2.5.10 static uint32\_t QSPI\_GetErrorStatusFlags ( QuadSPI\_Type \* *base* ) [inline], [static]

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
|-------------|--------------------------|

Returns

status flag, use status flag to AND `_qspi_error_flags` could get the related status.

#### 28.2.5.11 static void QSPI\_ClearErrorFlag ( QuadSPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                                                                                           |
|-------------|-------------------------------------------------------------------------------------------|
| <i>base</i> | Pointer to QuadSPI Type.                                                                  |
| <i>mask</i> | Which kind of QSPI flags to be cleared, a combination of <code>_qspi_error_flags</code> . |

#### 28.2.5.12 static void QSPI\_EnableInterrupts ( QuadSPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
| <i>mask</i> | QSPI interrupt source.   |

#### 28.2.5.13 static void QSPI\_DisableInterrupts ( QuadSPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
| <i>mask</i> | QSPI interrupt source.   |

#### 28.2.5.14 static void QSPI\_EnableDMA ( QuadSPI\_Type \* *base*, uint32\_t *mask*, bool *enable* ) [inline], [static]

Parameters

|               |                                             |
|---------------|---------------------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.                    |
| <i>mask</i>   | QSPI DMA source.                            |
| <i>enable</i> | True means enable DMA, false means disable. |

#### 28.2.5.15 static uint32\_t QSPI\_GetTxDataRegisterAddress ( QuadSPI\_Type \* *base* ) [inline], [static]

It is used for DMA operation.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
|-------------|--------------------------|

Returns

QSPI Tx data register address.

#### 28.2.5.16 uint32\_t QSPI\_GetRxDataRegisterAddress ( QuadSPI\_Type \* *base* )

This function returns the Rx data register address or Rx buffer address according to the Rx read area settings.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
|-------------|--------------------------|

Returns

QSPI Rx data register address.

28.2.5.17 **static void QSPI\_SetIPCommandAddress ( QuadSPI\_Type \* *base*, uint32\_t *addr* ) [inline], [static]**

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
| <i>addr</i> | IP command address.      |

**28.2.5.18 static void QSPI\_SetIPCommandSize ( QuadSPI\_Type \* *base*, uint32\_t *size* )  
[inline], [static]**

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
| <i>size</i> | IP command size.         |

**28.2.5.19 void QSPI\_ExecuteIPCommand ( QuadSPI\_Type \* *base*, uint32\_t *index* )**

Parameters

|              |                                              |
|--------------|----------------------------------------------|
| <i>base</i>  | Pointer to QuadSPI Type.                     |
| <i>index</i> | IP command located in which LUT table index. |

**28.2.5.20 void QSPI\_ExecuteAHBCommand ( QuadSPI\_Type \* *base*, uint32\_t *index* )**

Parameters

|              |                                               |
|--------------|-----------------------------------------------|
| <i>base</i>  | Pointer to QuadSPI Type.                      |
| <i>index</i> | AHB command located in which LUT table index. |

**28.2.5.21 void QSPI\_UpdateLUT ( QuadSPI\_Type \* *base*, uint32\_t *index*, uint32\_t \* *cmd* )**

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
|-------------|--------------------------|

|              |                                                                            |
|--------------|----------------------------------------------------------------------------|
| <i>index</i> | Which LUT index needs to be located. It should be an integer divided by 4. |
| <i>cmd</i>   | Command sequence array.                                                    |

**28.2.5.22 static void QSPI\_ClearFifo ( QuadSPI\_Type \* *base*, uint32\_t *mask* ) [inline], [static]**

Parameters

|             |                                        |
|-------------|----------------------------------------|
| <i>base</i> | Pointer to QuadSPI Type.               |
| <i>mask</i> | Which kind of QSPI FIFO to be cleared. |

**28.2.5.23 static void QSPI\_ClearCommandSequence ( QuadSPI\_Type \* *base*, qspi\_command\_seq\_t *seq* ) [inline], [static]**

This function can reset the command sequence.

Parameters

|             |                                                                           |
|-------------|---------------------------------------------------------------------------|
| <i>base</i> | QSPI base address.                                                        |
| <i>seq</i>  | Which command sequence need to reset, IP command, buffer command or both. |

**28.2.5.24 static void QSPI\_EnableDDRMode ( QuadSPI\_Type \* *base*, bool *enable* ) [inline], [static]**

Parameters

|               |                                                           |
|---------------|-----------------------------------------------------------|
| <i>base</i>   | QSPI base pointer                                         |
| <i>enable</i> | True means enable DDR mode, false means disable DDR mode. |

**28.2.5.25 void QSPI\_SetReadDataArea ( QuadSPI\_Type \* *base*, qspi\_read\_area\_t *area* )**

This function can set the RX buffer readout, from AHB bus or IP Bus.

Parameters

|             |                                                               |
|-------------|---------------------------------------------------------------|
| <i>base</i> | QSPI base address.                                            |
| <i>area</i> | QSPI Rx buffer readout area. AHB bus buffer or IP bus buffer. |

**28.2.5.26 void QSPI\_WriteBlocking ( QuadSPI\_Type \* *base*, uint32\_t \* *buffer*, size\_t *size* )**

## Note

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

## Parameters

|               |                                  |
|---------------|----------------------------------|
| <i>base</i>   | QSPI base pointer                |
| <i>buffer</i> | The data bytes to send           |
| <i>size</i>   | The number of data bytes to send |

**28.2.5.27 static void QSPI\_WriteData ( QuadSPI\_Type \* *base*, uint32\_t *data* ) [inline], [static]**

## Parameters

|             |                        |
|-------------|------------------------|
| <i>base</i> | QSPI base pointer      |
| <i>data</i> | The data bytes to send |

**28.2.5.28 void QSPI\_ReadBlocking ( QuadSPI\_Type \* *base*, uint32\_t \* *buffer*, size\_t *size* )**

## Note

This function blocks via polling until all bytes have been sent. Users shall notice that this receive size shall not bigger than 64 bytes. As this interface is used to read flash status registers. For flash contents read, please use AHB bus read, this is much more efficiency.

## Parameters

|               |                                     |
|---------------|-------------------------------------|
| <i>base</i>   | QSPI base pointer                   |
| <i>buffer</i> | The data bytes to send              |
| <i>size</i>   | The number of data bytes to receive |

**28.2.5.29 uint32\_t QSPI\_ReadData ( QuadSPI\_Type \* *base* )**

Parameters

|             |                   |
|-------------|-------------------|
| <i>base</i> | QSPI base pointer |
|-------------|-------------------|

Returns

The data in the FIFO.

#### 28.2.5.30 static void QSPI\_TransferSendBlocking ( QuadSPI\_Type \* *base*, qspi\_transfer\_t \* *xfer* ) [inline], [static]

This function writes a continuous data to the QSPI transmit FIFO. This function is a block function and can return only when finished. This function uses polling methods.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
| <i>xfer</i> | QSPI transfer structure. |

#### 28.2.5.31 static void QSPI\_TransferReceiveBlocking ( QuadSPI\_Type \* *base*, qspi\_transfer\_t \* *xfer* ) [inline], [static]

This function reads continuous data from the QSPI receive buffer/FIFO. This function is a blocking function and can return only when finished. This function uses polling methods. Users shall notice that this receive size shall not bigger than 64 bytes. As this interface is used to read flash status registers. For flash contents read, please use AHB bus read, this is much more efficiency.

Parameters

|             |                          |
|-------------|--------------------------|
| <i>base</i> | Pointer to QuadSPI Type. |
| <i>xfer</i> | QSPI transfer structure. |

## 28.3 Quad Serial Peripheral Interface EDMA Driver

### 28.3.1 Overview

#### Data Structures

- struct `qspi_edma_handle_t`  
*QSPI DMA transfer handle, users should not touch the content of the handle. [More...](#)*

#### TypeDefs

- typedef void(\* `qspi_edma_callback_t`)(QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle, `status_t` status, void \*userData)  
*QSPI eDMA transfer callback function for finish and error.*

#### Driver version

- #define `FSL_QSPI_EDMA_DRIVER_VERSION` (`MAKE_VERSION(2, 2, 2)`)  
*QSPI EDMA driver version 2.2.2.*

#### eDMA Transactional

- void `QSPI_TransferTxCreateHandleEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle, `qspi_edma_callback_t` callback, void \*userData, `edma_handle_t` \*dmaHandle)  
*Initializes the QSPI handle for send which is used in transactional functions and set the callback.*
- void `QSPI_TransferRxCreateHandleEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle, `qspi_edma_callback_t` callback, void \*userData, `edma_handle_t` \*dmaHandle)  
*Initializes the QSPI handle for receive which is used in transactional functions and set the callback.*
- `status_t QSPI_TransferSendEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle, `qspi_transfer_t` \*xfer)  
*Transfers QSPI data using an eDMA non-blocking method.*
- `status_t QSPI_TransferReceiveEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle, `qspi_transfer_t` \*xfer)  
*Receives data using an eDMA non-blocking method.*
- void `QSPI_TransferAbortSendEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle)  
*Aborts the sent data using eDMA.*
- void `QSPI_TransferAbortReceiveEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle)  
*Aborts the receive data using eDMA.*
- `status_t QSPI_TransferGetSendCountEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle, `size_t` \*count)  
*Gets the transferred counts of send.*
- `status_t QSPI_TransferGetReceiveCountEDMA` (QuadSPI\_Type \*base, qspi\_edma\_handle\_t \*handle, `size_t` \*count)  
*Gets the status of the receive transfer.*

## 28.3.2 Data Structure Documentation

### 28.3.2.1 struct \_qspi\_edma\_handle

#### Data Fields

- `edma_handle_t * dmaHandle`  
*eDMA handler for QSPI send.*
- `size_t transferSize`  
*Bytes need to transfer.*
- `uint8_t nbytes`  
*eDMA minor byte transfer count initially configured.*
- `uint8_t count`  
*The transfer data count in a DMA request.*
- `uint32_t state`  
*Internal state for QSPI eDMA transfer.*
- `qspi_edma_callback_t callback`  
*Callback for users while transfer finish or error occurred.*
- `void * userData`  
*User callback parameter.*

#### Field Documentation

- (1) `size_t qspi_edma_handle_t::transferSize`
- (2) `uint8_t qspi_edma_handle_t::nbytes`

## 28.3.3 Macro Definition Documentation

### 28.3.3.1 #define FSL\_QSPI\_EDMA\_DRIVER\_VERSION (MAKE\_VERSION(2, 2, 2))

## 28.3.4 Function Documentation

### 28.3.4.1 void QSPI\_TransferTxCreateHandleEDMA ( QuadSPI\_Type \* base, qspi\_edma\_handle\_t \* handle, qspi\_edma\_callback\_t callback, void \* userData, edma\_handle\_t \* dmaHandle )

Parameters

|                     |                                                      |
|---------------------|------------------------------------------------------|
| <code>base</code>   | QSPI peripheral base address                         |
| <code>handle</code> | Pointer to <code>qspi_edma_handle_t</code> structure |

|                  |                                              |
|------------------|----------------------------------------------|
| <i>callback</i>  | QSPI callback, NULL means no callback.       |
| <i>userData</i>  | User callback function data.                 |
| <i>dmaHandle</i> | User requested eDMA handle for eDMA transfer |

**28.3.4.2 void QSPI\_TransferRxCreateHandleEDMA ( QuadSPI\_Type \* *base*, qspi\_edma\_handle\_t \* *handle*, qspi\_edma\_callback\_t *callback*, void \* *userData*, edma\_handle\_t \* *dmaHandle* )**

Parameters

|                  |                                              |
|------------------|----------------------------------------------|
| <i>base</i>      | QSPI peripheral base address                 |
| <i>handle</i>    | Pointer to qspi_edma_handle_t structure      |
| <i>callback</i>  | QSPI callback, NULL means no callback.       |
| <i>userData</i>  | User callback function data.                 |
| <i>dmaHandle</i> | User requested eDMA handle for eDMA transfer |

**28.3.4.3 status\_t QSPI\_TransferSendEDMA ( QuadSPI\_Type \* *base*, qspi\_edma\_handle\_t \* *handle*, qspi\_transfer\_t \* *xfer* )**

This function writes data to the QSPI transmit FIFO. This function is non-blocking.

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.                |
| <i>handle</i> | Pointer to qspi_edma_handle_t structure |
| <i>xfer</i>   | QSPI transfer structure.                |

**28.3.4.4 status\_t QSPI\_TransferReceiveEDMA ( QuadSPI\_Type \* *base*, qspi\_edma\_handle\_t \* *handle*, qspi\_transfer\_t \* *xfer* )**

This function receive data from the QSPI receive buffer/FIFO. This function is non-blocking. Users shall notice that this receive size shall not bigger than 64 bytes. As this interface is used to read flash status registers. For flash contents read, please use AHB bus read, this is much more efficiency.

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.                |
| <i>handle</i> | Pointer to qspi_edma_handle_t structure |
| <i>xfer</i>   | QSPI transfer structure.                |

#### **28.3.4.5 void QSPI\_TransferAbortSendEDMA ( QuadSPI\_Type \* *base*, qspi\_edma\_handle\_t \* *handle* )**

This function aborts the sent data using eDMA.

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | QSPI peripheral base address.           |
| <i>handle</i> | Pointer to qspi_edma_handle_t structure |

#### **28.3.4.6 void QSPI\_TransferAbortReceiveEDMA ( QuadSPI\_Type \* *base*, qspi\_edma\_handle\_t \* *handle* )**

This function abort receive data which using eDMA.

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | QSPI peripheral base address.           |
| <i>handle</i> | Pointer to qspi_edma_handle_t structure |

#### **28.3.4.7 status\_t QSPI\_TransferGetSendCountEDMA ( QuadSPI\_Type \* *base*, qspi\_edma\_handle\_t \* *handle*, size\_t \* *count* )**

Parameters

|               |                                          |
|---------------|------------------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.                 |
| <i>handle</i> | Pointer to qspi_edma_handle_t structure. |
| <i>count</i>  | Bytes sent.                              |

Return values

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

#### 28.3.4.8 `status_t QSPI_TransferGetReceiveCountEDMA ( QuadSPI_Type * base, qspi_edma_handle_t * handle, size_t * count )`

Parameters

|               |                                         |
|---------------|-----------------------------------------|
| <i>base</i>   | Pointer to QuadSPI Type.                |
| <i>handle</i> | Pointer to qspi_edma_handle_t structure |
| <i>count</i>  | Bytes received.                         |

Return values

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

# Chapter 29

## SAI: Serial Audio Interface

### 29.1 Overview

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

SAI driver includes functional APIs and transactional APIs.

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

Transactional APIs target high-level APIs. Transactional APIs can be used to enable the peripheral and in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are a critical requirement, see the transactional API implementation and write a custom code. All transactional APIs use the `sai_handle_t` as the first parameter. Initialize the handle by calling the [SAI\\_TransferTxCreateHandle\(\)](#) or [SAI\\_TransferRxCreateHandle\(\)](#) API.

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

### 29.2 Typical configurations

#### Bit width configuration

SAI driver support 8/16/24/32bits stereo/mono raw audio data transfer. SAI EDMA driver support 8/16/32bits stereo/mono raw audio data transfer, since the EDMA doesn't support 24bit data width, so application should pre-convert the 24bit data to 32bit. SAI DMA driver support 8/16/32bits stereo/mono raw audio data transfer, since the EDMA doesn't support 24bit data width, so application should pre-convert the 24bit data to 32bit. SAI SDMA driver support 8/16/24/32bits stereo/mono raw audio data transfer.

#### Frame configuration

SAI driver support I2S, DSP, Left justified, Right justified, TDM mode. Application can call the api directly: `SAI_GetClassicI2SConfig` `SAI_GetLeftJustifiedConfig` `SAI_GetRightJustifiedConfig` `SAI_GetTDMConfig` `SAI_GetDSPConfig`

## 29.3 Typical use case

### 29.3.1 SAI Send/receive using an interrupt method

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/sai

### 29.3.2 SAI Send/receive using a DMA method

Refer to the driver examples codes located at <SDK\_ROOT>/boards/<BOARD>/driver\_examples/sai

## Modules

- [SAI Driver](#)
- [SAI EDMA Driver](#)

## 29.4 SAI Driver

### 29.4.1 Overview

#### Data Structures

- struct [sai\\_config\\_t](#)  
*SAI user configuration structure. [More...](#)*
- struct [sai\\_transfer\\_format\\_t](#)  
*sai transfer format [More...](#)*
- struct [sai\\_bit\\_clock\\_t](#)  
*sai bit clock configurations [More...](#)*
- struct [sai\\_frame\\_sync\\_t](#)  
*sai frame sync configurations [More...](#)*
- struct [sai\\_serial\\_data\\_t](#)  
*sai serial data configurations [More...](#)*
- struct [sai\\_transceiver\\_t](#)  
*sai transceiver configurations [More...](#)*
- struct [sai\\_transfer\\_t](#)  
*SAI transfer structure. [More...](#)*
- struct [sai\\_handle\\_t](#)  
*SAI handle structure. [More...](#)*

#### Macros

- #define [SAI\\_XFER\\_QUEUE\\_SIZE](#) (4U)  
*SAI transfer queue size, user can refine it according to use case.*
- #define [FSL\\_SAI\\_HAS\\_FIFO\\_EXTEND\\_FEATURE](#) 0  
*sai fifo feature*

#### Typedefs

- typedef void(\* [sai\\_transfer\\_callback\\_t](#) )(I2S\_Type \*base, sai\_handle\_t \*handle, [status\\_t](#) status, void \*userData)  
*SAI transfer callback prototype.*

#### Enumerations

- enum {
   
 kStatus\_SAI\_TxBusy = MAKE\_STATUS(kStatusGroup\_SAI, 0),
   
 kStatus\_SAI\_RxBusy = MAKE\_STATUS(kStatusGroup\_SAI, 1),
   
 kStatus\_SAI\_TxError = MAKE\_STATUS(kStatusGroup\_SAI, 2),
   
 kStatus\_SAI\_RxError = MAKE\_STATUS(kStatusGroup\_SAI, 3),
   
 kStatus\_SAI\_QueueFull = MAKE\_STATUS(kStatusGroup\_SAI, 4),
   
 kStatus\_SAI\_TxIdle = MAKE\_STATUS(kStatusGroup\_SAI, 5),
   
 kStatus\_SAI\_RxIdle = MAKE\_STATUS(kStatusGroup\_SAI, 6) }

- *\_sai\_status\_t, SAI return status.*
- enum {
   
kSAI\_Channel0Mask = 1 << 0U,  
 kSAI\_Channel1Mask = 1 << 1U,  
 kSAI\_Channel2Mask = 1 << 2U,  
 kSAI\_Channel3Mask = 1 << 3U,  
 kSAI\_Channel4Mask = 1 << 4U,  
 kSAI\_Channel5Mask = 1 << 5U,  
 kSAI\_Channel6Mask = 1 << 6U,  
 kSAI\_Channel7Mask = 1 << 7U }
- *\_sai\_channel\_mask,.sai channel mask value, actual channel numbers is depend soc specific*
- enum **sai\_protocol\_t** {
   
kSAI\_BusLeftJustified = 0x0U,  
 kSAI\_BusRightJustified,  
 kSAI\_BusI2S,  
 kSAI\_BusPCMA,  
 kSAI\_BusPCMB }
- *Define the SAI bus type.*
- enum **sai\_master\_slave\_t** {
   
kSAI\_Master = 0x0U,  
 kSAI\_Slave = 0x1U,  
 kSAI\_Bclk\_Master\_FrameSync\_Slave = 0x2U,  
 kSAI\_Bclk\_Slave\_FrameSync\_Master = 0x3U }
- *Master or slave mode.*
- enum **sai\_mono\_stereo\_t** {
   
kSAI\_Stereo = 0x0U,  
 kSAI\_MonoRight,  
 kSAI\_MonoLeft }
- *Mono or stereo audio format.*
- enum **sai\_data\_order\_t** {
   
kSAI\_DataLSB = 0x0U,  
 kSAI\_DataMSB }
- *SAI data order, MSB or LSB.*
- enum **sai\_clock\_polarity\_t** {
   
kSAI\_PolarityActiveHigh = 0x0U,  
 kSAI\_PolarityActiveLow = 0x1U,  
 kSAI\_SampleOnFallingEdge = 0x0U,  
 kSAI\_SampleOnRisingEdge = 0x1U }
- *SAI clock polarity, active high or low.*
- enum **sai\_sync\_mode\_t** {
   
kSAI\_ModeAsync = 0x0U,  
 kSAI\_ModeSync }
- *Synchronous or asynchronous mode.*
- enum **sai\_mclk\_source\_t** {
   
kSAI\_MclkSourceSysclk = 0x0U,  
 kSAI\_MclkSourceSelect1,  
 kSAI\_MclkSourceSelect2,

- ```

    kSAI_MclkSourceSelect3 }

    Mater clock source.
• enum sai_bclk_source_t {
    kSAI_BclkSourceBusclk = 0x0U,
    kSAI_BclkSourceMclkOption1 = 0x1U,
    kSAI_BclkSourceMclkOption2 = 0x2U,
    kSAI_BclkSourceMclkOption3 = 0x3U,
    kSAI_BclkSourceMclkDiv = 0x1U,
    kSAI_BclkSourceOtherSai0 = 0x2U,
    kSAI_BclkSourceOtherSai1 = 0x3U }

    Bit clock source.
• enum {
    kSAI_WordStartInterruptEnable,
    kSAI_SyncErrorInterruptEnable = I2S_TCSR_SEIE_MASK,
    kSAI_FIFOWarningInterruptEnable = I2S_TCSR_FWIE_MASK,
    kSAI_FIFOErrorInterruptEnable = I2S_TCSR_FEIE_MASK }

    _sai_interrupt_enable_t, The SAI interrupt enable flag
• enum { kSAI_FIFOWarningDMAEnable = I2S_TCSR_FWDE_MASK }

    _sai_dma_enable_t, The DMA request sources
• enum {
    kSAI_WordStartFlag = I2S_TCSR_WSF_MASK,
    kSAI_SyncErrorFlag = I2S_TCSR_SEF_MASK,
    kSAI_FIFOErrorFlag = I2S_TCSR_FEF_MASK,
    kSAI_FIFOWarningFlag = I2S_TCSR_FWF_MASK }

    _sai_flags, The SAI status flag
• enum sai_reset_type_t {
    kSAI_ResetTypeSoftware = I2S_TCSR_SR_MASK,
    kSAI_ResetTypeFIFO = I2S_TCSR_FR_MASK,
    kSAI_ResetAll = I2S_TCSR_SR_MASK | I2S_TCSR_FR_MASK }

    The reset type.
• enum sai_sample_rate_t {
    kSAI_SampleRate8KHz = 8000U,
    kSAI_SampleRate11025Hz = 11025U,
    kSAI_SampleRate12KHz = 12000U,
    kSAI_SampleRate16KHz = 16000U,
    kSAI_SampleRate22050Hz = 22050U,
    kSAI_SampleRate24KHz = 24000U,
    kSAI_SampleRate32KHz = 32000U,
    kSAI_SampleRate44100Hz = 44100U,
    kSAI_SampleRate48KHz = 48000U,
    kSAI_SampleRate96KHz = 96000U,
    kSAI_SampleRate192KHz = 192000U,
    kSAI_SampleRate384KHz = 384000U }

    Audio sample rate.
• enum sai_word_width_t {

```

```

kSAI_WordWidth8bits = 8U,
kSAI_WordWidth16bits = 16U,
kSAI_WordWidth24bits = 24U,
kSAI_WordWidth32bits = 32U }

Audio word width.
• enum sai_transceiver_type_t {
    kSAI_Transmitter = 0U,
    kSAI_Receiver = 1U }
sai transceiver type
• enum sai_frame_sync_len_t {
    kSAI_FrameSyncLenOneBitClk = 0U,
    kSAI_FrameSyncLenPerWordWidth = 1U }
sai frame sync len

```

Driver version

- #define **FSL_SAI_DRIVER_VERSION** (MAKE_VERSION(2, 3, 8))
Version 2.3.8.

Initialization and deinitialization

- void **SAI_TxInit** (I2S_Type *base, const **sai_config_t** *config)
Initializes the SAI Tx peripheral.
- void **SAI_RxInit** (I2S_Type *base, const **sai_config_t** *config)
Initializes the SAI Rx peripheral.
- void **SAI_TxGetDefaultConfig** (**sai_config_t** *config)
Sets the SAI Tx configuration structure to default values.
- void **SAI_RxGetDefaultConfig** (**sai_config_t** *config)
Sets the SAI Rx configuration structure to default values.
- void **SAI_Init** (I2S_Type *base)
Initializes the SAI peripheral.
- void **SAI_Deinit** (I2S_Type *base)
De-initializes the SAI peripheral.
- void **SAI_TxReset** (I2S_Type *base)
Resets the SAI Tx.
- void **SAI_RxReset** (I2S_Type *base)
Resets the SAI Rx.
- void **SAI_TxEnable** (I2S_Type *base, bool enable)
Enables/disables the SAI Tx.
- void **SAI_RxEnable** (I2S_Type *base, bool enable)
Enables/disables the SAI Rx.
- static void **SAI_TxSetBitClockDirection** (I2S_Type *base, **sai_master_slave_t** masterSlave)
Set Rx bit clock direction.
- static void **SAI_RxSetBitClockDirection** (I2S_Type *base, **sai_master_slave_t** masterSlave)
Set Rx bit clock direction.
- static void **SAI_RxSetFrameSyncDirection** (I2S_Type *base, **sai_master_slave_t** masterSlave)
Set Rx frame sync direction.

- static void **SAI_TxSetFrameSyncDirection** (I2S_Type *base, **sai_master_slave_t** masterSlave)
Set Tx frame sync direction.
- void **SAI_TxSetBitClockRate** (I2S_Type *base, uint32_t sourceClockHz, uint32_t sampleRate, uint32_t bitWidth, uint32_t channelNumbers)
Transmitter bit clock rate configurations.
- void **SAI_RxSetBitClockRate** (I2S_Type *base, uint32_t sourceClockHz, uint32_t sampleRate, uint32_t bitWidth, uint32_t channelNumbers)
Receiver bit clock rate configurations.
- void **SAI_TxSetBitclockConfig** (I2S_Type *base, **sai_master_slave_t** masterSlave, **sai_bit_clock_t** *config)
Transmitter Bit clock configurations.
- void **SAI_RxSetBitclockConfig** (I2S_Type *base, **sai_master_slave_t** masterSlave, **sai_bit_clock_t** *config)
Receiver Bit clock configurations.
- void **SAI_TxSetFrameSyncConfig** (I2S_Type *base, **sai_master_slave_t** masterSlave, **sai_frame_sync_t** *config)
SAI transmitter Frame sync configurations.
- void **SAI_RxSetFrameSyncConfig** (I2S_Type *base, **sai_master_slave_t** masterSlave, **sai_frame_sync_t** *config)
SAI receiver Frame sync configurations.
- void **SAI_TxSetSerialDataConfig** (I2S_Type *base, **sai_serial_data_t** *config)
SAI transmitter Serial data configurations.
- void **SAI_RxSetSerialDataConfig** (I2S_Type *base, **sai_serial_data_t** *config)
SAI receiver Serial data configurations.
- void **SAI_TxSetConfig** (I2S_Type *base, **sai_transceiver_t** *config)
SAI transmitter configurations.
- void **SAI_RxSetConfig** (I2S_Type *base, **sai_transceiver_t** *config)
SAI receiver configurations.
- void **SAI_GetClassicI2SConfig** (**sai_transceiver_t** *config, **sai_word_width_t** bitWidth, **sai_mono_stereo_t** mode, uint32_t saiChannelMask)
Get classic I2S mode configurations.
- void **SAI_GetLeftJustifiedConfig** (**sai_transceiver_t** *config, **sai_word_width_t** bitWidth, **sai_mono_stereo_t** mode, uint32_t saiChannelMask)
Get left justified mode configurations.
- void **SAI_GetRightJustifiedConfig** (**sai_transceiver_t** *config, **sai_word_width_t** bitWidth, **sai_mono_stereo_t** mode, uint32_t saiChannelMask)
Get right justified mode configurations.
- void **SAI_GetTDMConfig** (**sai_transceiver_t** *config, **sai_frame_sync_len_t** frameSyncWidth, **sai_word_width_t** bitWidth, uint32_t dataWordNum, uint32_t saiChannelMask)
Get TDM mode configurations.
- void **SAI_GetDSPConfig** (**sai_transceiver_t** *config, **sai_frame_sync_len_t** frameSyncWidth, **sai_word_width_t** bitWidth, **sai_mono_stereo_t** mode, uint32_t saiChannelMask)
Get DSP mode configurations.

Status

- static uint32_t **SAI_TxGetStatusFlag** (I2S_Type *base)
Gets the SAI Tx status flag state.

- static void **SAI_TxClearStatusFlags** (I2S_Type *base, uint32_t mask)
Clears the SAI Tx status flag state.
- static uint32_t **SAI_RxGetStatusFlag** (I2S_Type *base)
Gets the SAI Tx status flag state.
- static void **SAI_RxClearStatusFlags** (I2S_Type *base, uint32_t mask)
Clears the SAI Rx status flag state.
- void **SAI_TxSoftwareReset** (I2S_Type *base, sai_reset_type_t resetType)
Do software reset or FIFO reset .
- void **SAI_RxSoftwareReset** (I2S_Type *base, sai_reset_type_t resetType)
Do software reset or FIFO reset .
- void **SAI_TxSetChannelFIFOMask** (I2S_Type *base, uint8_t mask)
Set the Tx channel FIFO enable mask.
- void **SAI_RxSetChannelFIFOMask** (I2S_Type *base, uint8_t mask)
Set the Rx channel FIFO enable mask.
- void **SAI_TxSetDataOrder** (I2S_Type *base, sai_data_order_t order)
Set the Tx data order.
- void **SAI_RxSetDataOrder** (I2S_Type *base, sai_data_order_t order)
Set the Rx data order.
- void **SAI_TxSetBitClockPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Tx data order.
- void **SAI_RxSetBitClockPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Rx data order.
- void **SAI_TxSetFrameSyncPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Tx data order.
- void **SAI_RxSetFrameSyncPolarity** (I2S_Type *base, sai_clock_polarity_t polarity)
Set the Rx data order.

Interrupts

- static void **SAI_TxEnableInterrupts** (I2S_Type *base, uint32_t mask)
Enables the SAI Tx interrupt requests.
- static void **SAI_RxEnableInterrupts** (I2S_Type *base, uint32_t mask)
Enables the SAI Rx interrupt requests.
- static void **SAI_TxDisableInterrupts** (I2S_Type *base, uint32_t mask)
Disables the SAI Tx interrupt requests.
- static void **SAI_RxDisableInterrupts** (I2S_Type *base, uint32_t mask)
Disables the SAI Rx interrupt requests.

DMA Control

- static void **SAI_TxEnableDMA** (I2S_Type *base, uint32_t mask, bool enable)
Enables/disables the SAI Tx DMA requests.
- static void **SAI_RxEnableDMA** (I2S_Type *base, uint32_t mask, bool enable)
Enables/disables the SAI Rx DMA requests.
- static uintptr_t **SAI_TxGetDataRegisterAddress** (I2S_Type *base, uint32_t channel)
Gets the SAI Tx data register address.
- static uintptr_t **SAI_RxGetDataRegisterAddress** (I2S_Type *base, uint32_t channel)
Gets the SAI Rx data register address.

Bus Operations

- void **SAI_TxSetFormat** (I2S_Type *base, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- void **SAI_RxSetFormat** (I2S_Type *base, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- void **SAI_WriteBlocking** (I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Sends data using a blocking method.
- void **SAI_WriteMultiChannelBlocking** (I2S_Type *base, uint32_t channel, uint32_t channelMask, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Sends data to multi channel using a blocking method.
- static void **SAI_WriteData** (I2S_Type *base, uint32_t channel, uint32_t data)
Writes data into SAI FIFO.
- void **SAI_ReadBlocking** (I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Receives data using a blocking method.
- void **SAI_ReadMultiChannelBlocking** (I2S_Type *base, uint32_t channel, uint32_t channelMask, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Receives multi channel data using a blocking method.
- static uint32_t **SAI_ReadData** (I2S_Type *base, uint32_t channel)
Reads data from the SAI FIFO.

Transactional

- void **SAI_TransferTxCreateHandle** (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
Initializes the SAI Tx handle.
- void **SAI_TransferRxCreateHandle** (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
Initializes the SAI Rx handle.
- void **SAI_TransferTxSetConfig** (I2S_Type *base, sai_handle_t *handle, sai_transceiver_t *config)
SAI transmitter transfer configurations.
- void **SAI_TransferRxSetConfig** (I2S_Type *base, sai_handle_t *handle, sai_transceiver_t *config)
SAI receiver transfer configurations.
- status_t **SAI_TransferTxSetFormat** (I2S_Type *base, sai_handle_t *handle, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- status_t **SAI_TransferRxSetFormat** (I2S_Type *base, sai_handle_t *handle, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- status_t **SAI_TransferSendNonBlocking** (I2S_Type *base, sai_handle_t *handle, sai_transfer_t *xfer)
Performs an interrupt non-blocking send transfer on SAI.
- status_t **SAI_TransferReceiveNonBlocking** (I2S_Type *base, sai_handle_t *handle, sai_transfer_t *xfer)

Performs an interrupt non-blocking receive transfer on SAI.

- **status_t SAI_TransferGetSendCount** (I2S_Type *base, sai_handle_t *handle, size_t *count)
Gets a set byte count.
- **status_t SAI_TransferGetReceiveCount** (I2S_Type *base, sai_handle_t *handle, size_t *count)
Gets a received byte count.
- **void SAI_TransferAbortSend** (I2S_Type *base, sai_handle_t *handle)
Aborts the current send.
- **void SAI_TransferAbortReceive** (I2S_Type *base, sai_handle_t *handle)
Aborts the current IRQ receive.
- **void SAI_TransferTerminateSend** (I2S_Type *base, sai_handle_t *handle)
Terminate all SAI send.
- **void SAI_TransferTerminateReceive** (I2S_Type *base, sai_handle_t *handle)
Terminate all SAI receive.
- **void SAI_TransferTxHandleIRQ** (I2S_Type *base, sai_handle_t *handle)
Tx interrupt handler.
- **void SAI_TransferRxHandleIRQ** (I2S_Type *base, sai_handle_t *handle)
Tx interrupt handler.

29.4.2 Data Structure Documentation

29.4.2.1 struct sai_config_t

Data Fields

- **sai_protocol_t protocol**
Audio bus protocol in SAI.
- **sai_sync_mode_t syncMode**
SAI sync mode, control Tx/Rx clock sync.
- **sai_bclk_source_t bclkSource**
Bit Clock source.
- **sai_master_slave_t masterSlave**
Master or slave.

29.4.2.2 struct sai_transfer_format_t

Data Fields

- **uint32_t sampleRate_Hz**
Sample rate of audio data.
- **uint32_t bitWidth**
Data length of audio data, usually 8/16/24/32 bits.
- **sai_mono_stereo_t stereo**
Mono or stereo.
- **uint8_t channel**
Transfer start channel.
- **uint8_t channelMask**
enabled channel mask value, reference _sai_channel_mask
- **uint8_t endChannel**

- `uint8_t channelNums`
end channel number
- `sai_protocol_t protocol`
Total enabled channel numbers.
- `bool isFrameSyncCompact`
Which audio protocol used.
- `bool isFrameSyncCompact`
True means Frame sync length is configurable according to bitWidth, false means frame sync length is 64 times of bit clock.

Field Documentation

(1) `bool sai_transfer_format_t::isFrameSyncCompact`

29.4.2.3 struct sai_bit_clock_t

Data Fields

- `bool bclkSrcSwap`
bit clock source swap
- `bool bclkInputDelay`
bit clock actually used by the transmitter is delayed by the pad output delay, this has effect of decreasing the data input setup time, but increasing the data output valid time .
- `sai_clock_polarity_t bclkPolarity`
bit clock polarity
- `sai_bclk_source_t bclkSource`
bit Clock source

Field Documentation

(1) `bool sai_bit_clock_t::bclkInputDelay`

29.4.2.4 struct sai_frame_sync_t

Data Fields

- `uint8_t frameSyncWidth`
frame sync width in number of bit clocks
- `bool frameSyncEarly`
TRUE is frame sync assert one bit before the first bit of frame FALSE is frame sync assert with the first bit of the frame.
- `sai_clock_polarity_t frameSyncPolarity`
frame sync polarity

29.4.2.5 struct sai_serial_data_t

Data Fields

- `sai_data_order_t dataOrder`
configure whether the LSB or MSB is transmitted first
- `uint8_t dataWord0Length`

- `uint8_t dataWordNLength`
configure the number of bits in the first word in each frame
- `uint8_t dataWordLength`
configure the number of bits in the each word in each frame, except the first word
- `uint8_t dataFirstBitShifted`
used to record the data length for dma transfer
- `uint8_t dataWordNum`
Configure the bit index for the first bit transmitted for each word in the frame.
- `uint32_t dataMaskedWord`
configure the number of words in each frame
- `uint32_t dataMaskedWord`
configure whether the transmit word is masked

29.4.2.6 struct sai_transceiver_t

Data Fields

- `sai_serial_data_t serialData`
serial data configurations
- `sai_frame_sync_t frameSync`
ws configurations
- `sai_bit_clock_t bitClock`
bit clock configurations
- `sai_master_slave_t masterSlave`
transceiver is master or slave
- `sai_sync_mode_t syncMode`
transceiver sync mode
- `uint8_t startChannel`
Transfer start channel.
- `uint8_t channelMask`
enabled channel mask value, reference _sai_channel_mask
- `uint8_t endChannel`
end channel number
- `uint8_t channelNums`
Total enabled channel numbers.

29.4.2.7 struct sai_transfer_t

Data Fields

- `uint8_t * data`
Data start address to transfer.
- `size_t dataSize`
Transfer size.

Field Documentation

- (1) `uint8_t* sai_transfer_t::data`
- (2) `size_t sai_transfer_t::dataSize`

29.4.2.8 struct _sai_handle

Data Fields

- `I2S_Type * base`
base address
- `uint32_t state`
Transfer status.
- `sai_transfer_callback_t callback`
Callback function called at transfer event.
- `void * userData`
Callback parameter passed to callback function.
- `uint8_t bitWidth`
Bit width for transfer, 8/16/24/32 bits.
- `uint8_t channel`
Transfer start channel.
- `uint8_t channelMask`
enabled channel mask value, refernece _sai_channel_mask
- `uint8_t endChannel`
end channel number
- `uint8_t channelNums`
Total enabled channel numbers.
- `sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]`
Transfer queue storing queued transfer.
- `size_t transferSize [SAI_XFER_QUEUE_SIZE]`
Data bytes need to transfer.
- `volatile uint8_t queueUser`
Index for user to queue transfer.
- `volatile uint8_t queueDriver`
Index for driver to get the transfer data and size.

29.4.3 Macro Definition Documentation

29.4.3.1 #define SAI_XFER_QUEUE_SIZE (4U)

29.4.4 Enumeration Type Documentation

29.4.4.1 anonymous enum

Enumerator

`kStatus_SAI_TxBusy` SAI Tx is busy.

kStatus_SAI_RxBusy SAI Rx is busy.
kStatus_SAI_TxError SAI Tx FIFO error.
kStatus_SAI_RxError SAI Rx FIFO error.
kStatus_SAI_QueueFull SAI transfer queue is full.
kStatus_SAI_TxIdle SAI Tx is idle.
kStatus_SAI_RxIdle SAI Rx is idle.

29.4.4.2 anonymous enum

Enumerator

kSAI_Channel0Mask channel 0 mask value
kSAI_Channel1Mask channel 1 mask value
kSAI_Channel2Mask channel 2 mask value
kSAI_Channel3Mask channel 3 mask value
kSAI_Channel4Mask channel 4 mask value
kSAI_Channel5Mask channel 5 mask value
kSAI_Channel6Mask channel 6 mask value
kSAI_Channel7Mask channel 7 mask value

29.4.4.3 enum sai_protocol_t

Enumerator

kSAI_BusLeftJustified Uses left justified format.
kSAI_BusRightJustified Uses right justified format.
kSAI_BusI2S Uses I2S format.
kSAI_BusPCMA Uses I2S PCM A format.
kSAI_BusPCMB Uses I2S PCM B format.

29.4.4.4 enum sai_master_slave_t

Enumerator

kSAI_Master Master mode include bclk and frame sync.
kSAI_Slave Slave mode include bclk and frame sync.
kSAI_Bclk_Master_FrameSync_Slave bclk in master mode, frame sync in slave mode
kSAI_Bclk_Slave_FrameSync_Master bclk in slave mode, frame sync in master mode

29.4.4.5 enum sai_mono_stereo_t

Enumerator

kSAI_Stereo Stereo sound.

kSAI_MonoRight Only Right channel have sound.

kSAI_MonoLeft Only left channel have sound.

29.4.4.6 enum sai_data_order_t

Enumerator

kSAI_DataLSB LSB bit transferred first.

kSAI_DataMSB MSB bit transferred first.

29.4.4.7 enum sai_clock_polarity_t

Enumerator

kSAI_PolarityActiveHigh Drive outputs on rising edge.

kSAI_PolarityActiveLow Drive outputs on falling edge.

kSAI_SampleOnFallingEdge Sample inputs on falling edge.

kSAI_SampleOnRisingEdge Sample inputs on rising edge.

29.4.4.8 enum sai_sync_mode_t

Enumerator

kSAI_ModeAsync Asynchronous mode.

kSAI_ModeSync Synchronous mode (with receiver or transmit)

29.4.4.9 enum sai_mclk_source_t

Enumerator

kSAI_MclkSourceSysclk Master clock from the system clock.

kSAI_MclkSourceSelect1 Master clock from source 1.

kSAI_MclkSourceSelect2 Master clock from source 2.

kSAI_MclkSourceSelect3 Master clock from source 3.

29.4.4.10 enum sai_bclk_source_t

Enumerator

kSAI_BclkSourceBusclk Bit clock using bus clock.

kSAI_BclkSourceMclkOption1 Bit clock MCLK option 1.

kSAI_BclkSourceMclkOption2 Bit clock MCLK option2.
kSAI_BclkSourceMclkOption3 Bit clock MCLK option3.
kSAI_BclkSourceMclkDiv Bit clock using master clock divider.
kSAI_BclkSourceOtherSai0 Bit clock from other SAI device.
kSAI_BclkSourceOtherSai1 Bit clock from other SAI device.

29.4.4.11 anonymous enum

Enumerator

kSAI_WordStartInterruptEnable Word start flag, means the first word in a frame detected.
kSAI_SyncErrorInterruptEnable Sync error flag, means the sync error is detected.
kSAI_FIFOWarningInterruptEnable FIFO warning flag, means the FIFO is empty.
kSAI_FIFOErrorInterruptEnable FIFO error flag.

29.4.4.12 anonymous enum

Enumerator

kSAI_FIFOWarningDMAEnable FIFO warning caused by the DMA request.

29.4.4.13 anonymous enum

Enumerator

kSAI_WordStartFlag Word start flag, means the first word in a frame detected.
kSAI_SyncErrorFlag Sync error flag, means the sync error is detected.
kSAI_FIFOErrorFlag FIFO error flag.
kSAI_FIFOWarningFlag FIFO warning flag.

29.4.4.14 enum sai_reset_type_t

Enumerator

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

29.4.4.15 enum sai_sample_rate_t

Enumerator

kSAI_SampleRate8KHz Sample rate 8000 Hz.
kSAI_SampleRate11025Hz Sample rate 11025 Hz.

kSAI_SampleRate12KHz Sample rate 12000 Hz.
kSAI_SampleRate16KHz Sample rate 16000 Hz.
kSAI_SampleRate22050Hz Sample rate 22050 Hz.
kSAI_SampleRate24KHz Sample rate 24000 Hz.
kSAI_SampleRate32KHz Sample rate 32000 Hz.
kSAI_SampleRate44100Hz Sample rate 44100 Hz.
kSAI_SampleRate48KHz Sample rate 48000 Hz.
kSAI_SampleRate96KHz Sample rate 96000 Hz.
kSAI_SampleRate192KHz Sample rate 192000 Hz.
kSAI_SampleRate384KHz Sample rate 384000 Hz.

29.4.4.16 enum sai_word_width_t

Enumerator

kSAI_WordWidth8bits Audio data width 8 bits.
kSAI_WordWidth16bits Audio data width 16 bits.
kSAI_WordWidth24bits Audio data width 24 bits.
kSAI_WordWidth32bits Audio data width 32 bits.

29.4.4.17 enum sai_transceiver_type_t

Enumerator

kSAI_Transmitter sai transmitter
kSAI_Receiver sai receiver

29.4.4.18 enum sai_frame_sync_len_t

Enumerator

kSAI_FrameSyncLenOneBitClk 1 bit clock frame sync len for DSP mode
kSAI_FrameSyncLenPerWordWidth Frame sync length decided by word width.

29.4.5 Function Documentation

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

Deprecated Do not use this function. It has been superceded by [SAI_Init](#)

Ungates the SAI clock, resets the module, and configures SAI Tx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI_TxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAIM module can cause a hard fault because the clock is not enabled.

Parameters

<i>base</i>	SAI base pointer
<i>config</i>	SAI configuration structure.

29.4.5.2 void SAI_RxInit (I2S_Type * *base*, const sai_config_t * *config*)

Deprecated Do not use this function. It has been superceded by [SAI_Init](#)

Ungates the SAI clock, resets the module, and configures the SAI Rx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI_RxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAI module can cause a hard fault because the clock is not enabled.

Parameters

<i>base</i>	SAI base pointer
<i>config</i>	SAI configuration structure.

29.4.5.3 void SAI_TxGetDefaultConfig (sai_config_t * *config*)

Deprecated Do not use this function. It has been superceded by [SAI_GetClassicI2SConfig](#), [SAI_GetLeftJustifiedConfig](#), [SAI_GetRightJustifiedConfig](#), [SAI_GetDSPConfig](#), [SAI_GetTDMConfig](#)

This API initializes the configuration structure for use in SAI_TxConfig(). The initialized structure can remain unchanged in SAI_TxConfig(), or it can be modified before calling SAI_TxConfig(). This is an example.

```
sai_config_t config;
SAI_TxGetDefaultConfig(&config);
```

Parameters

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

29.4.5.4 void SAI_RxGetDefaultConfig (sai_config_t * *config*)

Deprecated Do not use this function. It has been superceded by [SAI_GetClassicI2SConfig](#), [SAI_GetLeftJustifiedConfig](#), [SAI_GetRightJustifiedConfig](#), [SAI_GetDSPConfig](#), [SAI_GetTDMConfig](#)

This API initializes the configuration structure for use in SAI_RxConfig(). The initialized structure can remain unchanged in SAI_RxConfig() or it can be modified before calling SAI_RxConfig(). This is an example.

```
sai_config_t config;
SAI_RxGetDefaultConfig(&config);
```

Parameters

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

29.4.5.5 void SAI_Init (I2S_Type * *base*)

This API gates the SAI clock. The SAI module can't operate unless SAI_Init is called to enable the clock.

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

29.4.5.6 void SAI_Deinit (I2S_Type * *base*)

This API gates the SAI clock. The SAI module can't operate unless SAI_TxInit or SAI_RxInit is called to enable the clock.

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

29.4.5.7 void SAI_TxReset (I2S_Type * *base*)

This function enables the software reset and FIFO reset of SAI Tx. After reset, clear the reset bit.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

29.4.5.8 void SAI_RxReset (I2S_Type * *base*)

This function enables the software reset and FIFO reset of SAI Rx. After reset, clear the reset bit.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

29.4.5.9 void SAI_TxEnable (I2S_Type * *base*, bool *enable*)

Parameters

<i>base</i>	SAI base pointer.
<i>enable</i>	True means enable SAI Tx, false means disable.

29.4.5.10 void SAI_RxEnable (I2S_Type * *base*, bool *enable*)

Parameters

<i>base</i>	SAI base pointer.
<i>enable</i>	True means enable SAI Rx, false means disable.

29.4.5.11 static void SAI_TxSetBitClockDirection (I2S_Type * *base*, sai_master_slave_t *masterSlave*) [inline], [static]

Select bit clock direction, master or slave.

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

<i>masterSlave</i>	reference sai_master_slave_t.
--------------------	-------------------------------

29.4.5.12 static void SAI_RxSetBitClockDirection (I2S_Type * *base*, sai_master_slave_t *masterSlave*) [inline], [static]

Select bit clock direction, master or slave.

Parameters

<i>base</i>	SAI base pointer.
<i>masterSlave</i>	reference sai_master_slave_t.

29.4.5.13 static void SAI_RxSetFrameSyncDirection (I2S_Type * *base*, sai_master_slave_t *masterSlave*) [inline], [static]

Select frame sync direction, master or slave.

Parameters

<i>base</i>	SAI base pointer.
<i>masterSlave</i>	reference sai_master_slave_t.

29.4.5.14 static void SAI_TxSetFrameSyncDirection (I2S_Type * *base*, sai_master_slave_t *masterSlave*) [inline], [static]

Select frame sync direction, master or slave.

Parameters

<i>base</i>	SAI base pointer.
<i>masterSlave</i>	reference sai_master_slave_t.

29.4.5.15 void SAI_TxSetBitClockRate (I2S_Type * *base*, uint32_t *sourceClockHz*, uint32_t *sampleRate*, uint32_t *bitWidth*, uint32_t *channelNumbers*)

Parameters

<i>base</i>	SAI base pointer.
<i>sourceClockHz</i>	Bit clock source frequency.
<i>sampleRate</i>	Audio data sample rate.
<i>bitWidth</i>	Audio data bitWidth.
<i>channel-Numbers</i>	Audio channel numbers.

29.4.5.16 void SAI_RxSetBitClockRate (I2S_Type * *base*, uint32_t *sourceClockHz*, uint32_t *sampleRate*, uint32_t *bitWidth*, uint32_t *channelNumbers*)

Parameters

<i>base</i>	SAI base pointer.
<i>sourceClockHz</i>	Bit clock source frequency.
<i>sampleRate</i>	Audio data sample rate.
<i>bitWidth</i>	Audio data bitWidth.
<i>channel-Numbers</i>	Audio channel numbers.

29.4.5.17 void SAI_TxSetBitclockConfig (I2S_Type * *base*, sai_master_slave_t *masterSlave*, sai_bit_clock_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>masterSlave</i>	master or slave.
<i>config</i>	bit clock other configurations, can be NULL in slave mode.

29.4.5.18 void SAI_RxSetBitclockConfig (I2S_Type * *base*, sai_master_slave_t *masterSlave*, sai_bit_clock_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>masterSlave</i>	master or slave.
<i>config</i>	bit clock other configurations, can be NULL in slave mode.

29.4.5.19 void SAI_TxSetFrameSyncConfig (I2S_Type * *base*, sai_master_slave_t *masterSlave*, sai_frame_sync_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>masterSlave</i>	master or slave.
<i>config</i>	frame sync configurations, can be NULL in slave mode.

29.4.5.20 void SAI_RxSetFrameSyncConfig (I2S_Type * *base*, sai_master_slave_t *masterSlave*, sai_frame_sync_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>masterSlave</i>	master or slave.
<i>config</i>	frame sync configurations, can be NULL in slave mode.

29.4.5.21 void SAI_TxSetSerialDataConfig (I2S_Type * *base*, sai_serial_data_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>config</i>	serial data configurations.

29.4.5.22 void SAI_RxSetSerialDataConfig (I2S_Type * *base*, sai_serial_data_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>config</i>	serial data configurations.

29.4.5.23 void SAI_TxSetConfig (I2S_Type * *base*, sai_transceiver_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>config</i>	transmitter configurations.

29.4.5.24 void SAI_RxSetConfig (I2S_Type * *base*, sai_transceiver_t * *config*)

Parameters

<i>base</i>	SAI base pointer.
<i>config</i>	receiver configurations.

29.4.5.25 void SAI_GetClassicI2SConfig (sai_transceiver_t * *config*, sai_word_width_t *bitWidth*, sai_mono_stereo_t *mode*, uint32_t *saiChannelMask*)

Parameters

<i>config</i>	transceiver configurations.
<i>bitWidth</i>	audio data bitWidth.
<i>mode</i>	audio data channel.
<i>saiChannel- Mask</i>	mask value of the channel to be enable.

29.4.5.26 void SAI_GetLeftJustifiedConfig (sai_transceiver_t * *config*, sai_word_width_t *bitWidth*, sai_mono_stereo_t *mode*, uint32_t *saiChannelMask*)

Parameters

<i>config</i>	transceiver configurations.
<i>bitWidth</i>	audio data bitWidth.
<i>mode</i>	audio data channel.
<i>saiChannel-Mask</i>	mask value of the channel to be enable.

**29.4.5.27 void SAI_GetRightJustifiedConfig (*sai_transceiver_t * config*,
sai_word_width_t bitWidth, *sai_mono_stereo_t mode*, *uint32_t saiChannelMask*)**

Parameters

<i>config</i>	transceiver configurations.
<i>bitWidth</i>	audio data bitWidth.
<i>mode</i>	audio data channel.
<i>saiChannel-Mask</i>	mask value of the channel to be enable.

29.4.5.28 void SAI_GetTDMConfig (*sai_transceiver_t * config*, *sai_frame_sync_len_t frameSyncWidth*, *sai_word_width_t bitWidth*, *uint32_t dataWordNum*, *uint32_t saiChannelMask*)

Parameters

<i>config</i>	transceiver configurations.
<i>frameSync-Width</i>	length of frame sync.
<i>bitWidth</i>	audio data word width.
<i>dataWordNum</i>	word number in one frame.
<i>saiChannel-Mask</i>	mask value of the channel to be enable.

29.4.5.29 void SAI_GetDSPConfig (sai_transceiver_t * *config*, sai_frame_sync_len_t *frameSyncWidth*, sai_word_width_t *bitWidth*, sai_mono_stereo_t *mode*, uint32_t *saiChannelMask*)

Note

DSP mode is also called PCM mode which support MODE A and MODE B, DSP/PCM MODE A configuration flow. RX is similiar but uses SAI_RxSetConfig instead of SAI_TxSetConfig:

```
* SAI_GetDSPConfig(config, kSAI_FrameSyncLenOneBitClk, bitWidth,
    kSAI_Stereo, channelMask)
* config->frameSync.frameSyncEarly = true;
* SAI_TxSetConfig(base, config)
*
```

DSP/PCM MODE B configuration flow for TX. RX is similiar but uses SAI_RxSetConfig instead of SAI_TxSetConfig:

```
* SAI_GetDSPConfig(config, kSAI_FrameSyncLenOneBitClk, bitWidth,
    kSAI_Stereo, channelMask)
* SAI_TxSetConfig(base, config)
*
```

Parameters

<i>config</i>	transceiver configurations.
<i>frameSyncWidth</i>	length of frame sync.
<i>bitWidth</i>	audio data bitWidth.
<i>mode</i>	audio data channel.
<i>saiChannelMask</i>	mask value of the channel to enable.

29.4.5.30 static uint32_t SAI_TxGetStatusFlag (I2S_Type * *base*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Returns

SAI Tx status flag value. Use the Status Mask to get the status value needed.

29.4.5.31 static void SAI_TxClearStatusFlags (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>State mask. It can be a combination of the following source if defined:</p> <ul style="list-style-type: none"> • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

29.4.5.32 static uint32_t SAI_RxGetStatusFlag (I2S_Type * *base*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Returns

SAI Rx status flag value. Use the Status Mask to get the status value needed.

29.4.5.33 static void SAI_RxClearStatusFlags (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>State mask. It can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

29.4.5.34 void SAI_TxSoftwareReset (I2S_Type * *base*, sai_reset_type_t *resetType*)

FIFO reset means clear all the data in the FIFO, and make the FIFO pointer both to 0. Software reset means clear the Tx internal logic, including the bit clock, frame count etc. But software reset will not clear any configuration registers like TCR1~TCR5. This function will also clear all the error flags such as FIFO error, sync error etc.

Parameters

<i>base</i>	SAI base pointer
<i>resetType</i>	Reset type, FIFO reset or software reset

29.4.5.35 void SAI_RxSoftwareReset (I2S_Type * *base*, sai_reset_type_t *resetType*)

FIFO reset means clear all the data in the FIFO, and make the FIFO pointer both to 0. Software reset means clear the Rx internal logic, including the bit clock, frame count etc. But software reset will not clear any configuration registers like RCR1~RCR5. This function will also clear all the error flags such as FIFO error, sync error etc.

Parameters

<i>base</i>	SAI base pointer
<i>resetType</i>	Reset type, FIFO reset or software reset

29.4.5.36 void SAI_TxSetChannelFIFOMask (I2S_Type * *base*, uint8_t *mask*)

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	Channel enable mask, 0 means all channel FIFO disabled, 1 means channel 0 enabled, 3 means both channel 0 and channel 1 enabled.

29.4.5.37 void SAI_RxSetChannelFIFOMask (I2S_Type * *base*, uint8_t *mask*)

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	Channel enable mask, 0 means all channel FIFO disabled, 1 means channel 0 enabled, 3 means both channel 0 and channel 1 enabled.

29.4.5.38 void SAI_TxSetDataOrder (I2S_Type * *base*, sai_data_order_t *order*)

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

29.4.5.39 void SAI_RxSetDataOrder (I2S_Type * *base*, sai_data_order_t *order*)

Parameters

<i>base</i>	SAI base pointer
<i>order</i>	Data order MSB or LSB

29.4.5.40 void SAI_TxSetBitClockPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Parameters

<i>base</i>	SAI base pointer
<i>polarity</i>	

29.4.5.41 void SAI_RxSetBitClockPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Parameters

<i>base</i>	SAI base pointer
<i>polarity</i>	

29.4.5.42 void SAI_TxSetFrameSyncPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Parameters

<i>base</i>	SAI base pointer
<i>polarity</i>	

29.4.5.43 void SAI_RxSetFrameSyncPolarity (I2S_Type * *base*, sai_clock_polarity_t *polarity*)

Parameters

<i>base</i>	SAI base pointer
<i>polarity</i>	

29.4.5.44 static void SAI_TxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

**29.4.5.45 static void SAI_RxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

**29.4.5.46 static void SAI_TxDisableInterrupts (I2S_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

**29.4.5.47 static void SAI_RxDisableInterrupts (I2S_Type * *base*, uint32_t *mask*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>interrupt source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

29.4.5.48 static void SAI_TxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>DMA source The parameter can be combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

29.4.5.49 static void SAI_RxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	<p>DMA source The parameter can be a combination of the following sources if defined.</p> <ul style="list-style-type: none"> • kSAI_FIFOWarningDMAEnable • kSAI_FIFORequestDMAEnable

<i>enable</i>	True means enable DMA, false means disable DMA.
---------------	---

29.4.5.50 static uintptr_t SAI_TxGetDataRegisterAddress (I2S_Type * *base*, uint32_t *channel*) [inline], [static]

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

29.4.5.51 static uintptr_t SAI_RxGetDataRegisterAddress (I2S_Type * *base*, uint32_t *channel*) [inline], [static]

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

29.4.5.52 void SAI_TxSetFormat (I2S_Type * *base*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

Deprecated Do not use this function. It has been superceded by [SAI_TxSetConfig](#)

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

Parameters

<i>base</i>	SAI base pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.

29.4.5.53 void SAI_RxSetFormat (I2S_Type * *base*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

Deprecated Do not use this function. It has been superceded by [SAI_RxSetConfig](#)

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

Parameters

<i>base</i>	SAI base pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.

29.4.5.54 void SAI_WriteBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be written.
<i>size</i>	Bytes to be written.

29.4.5.55 void SAI_WriteMultiChannelBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *channelMask*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>channelMask</i>	channel mask.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be written.
<i>size</i>	Bytes to be written.

29.4.5.56 static void SAI_WriteData (I2S_Type * *base*, uint32_t *channel*, uint32_t *data*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>data</i>	Data needs to be written.

29.4.5.57 void SAI_ReadBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be read.
<i>size</i>	Bytes to be read.

29.4.5.58 void SAI_ReadMultiChannelBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *channelMask*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>channelMask</i>	channel mask.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be read.
<i>size</i>	Bytes to be read.

29.4.5.59 static uint32_t SAI_ReadData (I2S_Type * *base*, uint32_t *channel*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.

Returns

Data in SAI FIFO.

**29.4.5.60 void SAI_TransferTxCreateHandle (I2S_Type * *base*, sai_handle_t * *handle*,
sai_transfer_callback_t *callback*, void * *userData*)**

This function initializes the Tx handle for the SAI Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI handle pointer.
<i>callback</i>	Pointer to the user callback function.
<i>userData</i>	User parameter passed to the callback function

29.4.5.61 void SAI_TransferRxCreateHandle (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_callback_t *callback*, void * *userData*)

This function initializes the Rx handle for the SAI Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>callback</i>	Pointer to the user callback function.
<i>userData</i>	User parameter passed to the callback function.

29.4.5.62 void SAI_TransferTxSetConfig (I2S_Type * *base*, sai_handle_t * *handle*, sai_transceiver_t * *config*)

This function initializes the Tx, include bit clock, frame sync, master clock, serial data and fifo configurations.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>config</i>	transmitter configurations.

29.4.5.63 void SAI_TransferRxSetConfig (I2S_Type * *base*, sai_handle_t * *handle*, sai_transceiver_t * *config*)

This function initializes the Rx, include bit clock, frame sync, master clock, serial data and fifo configurations.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>config</i>	receiver configurations.

29.4.5.64 status_t SAI_TransferTxSetFormat (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

Deprecated Do not use this function. It has been superceded by [SAI_TransferTxSetConfig](#)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the <i>masterClockHz</i> in <i>format</i> .

Returns

Status of this function. Return value is the *status_t*.

29.4.5.65 status_t SAI_TransferRxSetFormat (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

Deprecated Do not use this function. It has been superceded by [SAI_TransferRxSetConfig](#)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is one of status_t.

29.4.5.66 status_t SAI_TransferSendNonBlocking (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This API returns immediately after the transfer initiates. Call the SAI_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SAI_Busy, the transfer is finished.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>xfer</i>	Pointer to the sai_transfer_t structure.

Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_TxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

29.4.5.67 status_t SAI_TransferReceiveNonBlocking (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This API returns immediately after the transfer initiates. Call the SAI_RxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_SAI_Busy, the transfer is finished.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>xfer</i>	Pointer to the sai_transfer_t structure.

Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_RxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

29.4.5.68 status_t SAI_TransferGetSendCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count sent.

Return values

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

29.4.5.69 status_t SAI_TransferGetReceiveCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count received.

Return values

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

29.4.5.70 void SAI_TransferAbortSend (I2S_Type * *base*, sai_handle_t * *handle*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.

29.4.5.71 void SAI_TransferAbortReceive (I2S_Type * *base*, sai_handle_t * *handle*)

Note

This API can be called when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.

29.4.5.72 void SAI_TransferTerminateSend (I2S_Type * *base*, sai_handle_t * *handle*)

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortSend.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

29.4.5.73 void SAI_TransferTerminateReceive (I2S_Type * *base*, sai_handle_t * *handle*)

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceive.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

29.4.5.74 void SAI_TransferTxHandleIRQ (I2S_Type * *base*, sai_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure.

29.4.5.75 void SAI_TransferRxHandleIRQ (I2S_Type * *base*, sai_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure.

29.5 SAI EDMA Driver

29.5.1 Overview

Data Structures

- struct `sai_edma_handle_t`
SAI DMA transfer handle, users should not touch the content of the handle. [More...](#)

Typedefs

- typedef void(* `sai_edma_callback_t`)(I2S_Type *base, sai_edma_handle_t *handle, `status_t` status, void *userData)
SAI eDMA transfer callback function for finish and error.

Driver version

- #define `FSL_SAI_EDMA_DRIVER_VERSION` (`MAKE_VERSION(2, 5, 1)`)
Version 2.5.1.

eDMA Transactional

- void `SAI_TransferTxCreateHandleEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_edma_callback_t` callback, void *userData, `edma_handle_t` *txDmaHandle)
Initializes the SAI eDMA handle.
- void `SAI_TransferRxCreateHandleEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_edma_callback_t` callback, void *userData, `edma_handle_t` *rxDmaHandle)
Initializes the SAI Rx eDMA handle.
- void `SAI_TransferTxSetFormatEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_transfer_format_t` *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- void `SAI_TransferRxSetFormatEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_transfer_format_t` *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- void `SAI_TransferTxSetConfigEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_transceiver_t` *saiConfig)
Configures the SAI Tx.
- void `SAI_TransferRxSetConfigEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_transceiver_t` *saiConfig)
Configures the SAI Rx.
- `status_t SAI_TransferSendEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_transfer_t` *xfer)
Performs a non-blocking SAI transfer using DMA.
- `status_t SAI_TransferReceiveEDMA` (I2S_Type *base, sai_edma_handle_t *handle, `sai_transfer_t` *xfer)

- *Performs a non-blocking SAI receive using eDMA.*
status_t SAI_TransferSendLoopEDMA (I2S_Type *base, sai_edma_handle_t *handle, sai_transfer_t *xfer, uint32_t loopTransferCount)
- *Performs a non-blocking SAI loop transfer using eDMA.*
status_t SAI_TransferReceiveLoopEDMA (I2S_Type *base, sai_edma_handle_t *handle, sai_transfer_t *xfer, uint32_t loopTransferCount)
- *Performs a non-blocking SAI loop transfer using eDMA.*
void SAI_TransferTerminateSendEDMA (I2S_Type *base, sai_edma_handle_t *handle)
Terminate all SAI send.
- **void SAI_TransferTerminateReceiveEDMA** (I2S_Type *base, sai_edma_handle_t *handle)
Terminate all SAI receive.
- **void SAI_TransferAbortSendEDMA** (I2S_Type *base, sai_edma_handle_t *handle)
Aborts a SAI transfer using eDMA.
- **void SAI_TransferAbortReceiveEDMA** (I2S_Type *base, sai_edma_handle_t *handle)
Aborts a SAI receive using eDMA.
- **status_t SAI_TransferGetSendCountEDMA** (I2S_Type *base, sai_edma_handle_t *handle, size_t *count)
Gets byte count sent by SAI.
- **status_t SAI_TransferGetReceiveCountEDMA** (I2S_Type *base, sai_edma_handle_t *handle, size_t *count)
Gets byte count received by SAI.
- **uint32_t SAI_TransferGetValidTransferSlotsEDMA** (I2S_Type *base, sai_edma_handle_t *handle)
Gets valid transfer slot.

29.5.2 Data Structure Documentation

29.5.2.1 struct sai_edma_handle

Data Fields

- **edma_handle_t * dmaHandle**
DMA handler for SAI send.
- **uint8_t nbytes**
eDMA minor byte transfer count initially configured.
- **uint8_t bytesPerFrame**
Bytes in a frame.
- **uint8_t channelMask**
Enabled channel mask value, reference _sai_channel_mask.
- **uint8_t channelNums**
total enabled channel nums
- **uint8_t channel**
Which data channel.
- **uint8_t count**
The transfer data count in a DMA request.
- **uint32_t state**
Internal state for SAI eDMA transfer.
- **sai_edma_callback_t callback**
Callback for users while transfer finish or error occurs.
- **void * userData**

- *User callback parameter.*
- `uint8_t tcd [(SAI_XFER_QUEUE_SIZE+1U)*sizeof(edma_tcd_t)]`
TCD pool for eDMA transfer.
- `sai_transfer_t saiQueue [SAI_XFER_QUEUE_SIZE]`
Transfer queue storing queued transfer.
- `size_t transferSize [SAI_XFER_QUEUE_SIZE]`
Data bytes need to transfer.
- `volatile uint8_t queueUser`
Index for user to queue transfer.
- `volatile uint8_t queueDriver`
Index for driver to get the transfer data and size.

Field Documentation

- (1) `uint8_t sai_edma_handle_t::nbytes`
- (2) `uint8_t sai_edma_handle_t::tcd[(SAI_XFER_QUEUE_SIZE+1U)*sizeof(edma_tcd_t)]`
- (3) `sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`
- (4) `volatile uint8_t sai_edma_handle_t::queueUser`

29.5.3 Function Documentation

29.5.3.1 void SAI_TransferTxCreateHandleEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_edma_callback_t *callback*, void * *userData*, edma_handle_t * *txDmaHandle*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>txDmaHandle</i>	eDMA handle pointer, this handle shall be static allocated by users.

29.5.3.2 void SAI_TransferRxCreateHandleEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_edma_callback_t *callback*, void * *userData*, edma_handle_t * *rxDmaHandle*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>rxDmaHandle</i>	eDMA handle pointer, this handle shall be static allocated by users.

29.5.3.3 void SAI_TransferTxSetFormatEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

Deprecated Do not use this function. It has been superceded by [SAI_TransferTxSetConfigEDMA](#)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If bit clock source is master clock, this value should equals to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

29.5.3.4 void SAI_TransferRxSetFormatEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

Deprecated Do not use this function. It has been superceded by [SAI_TransferRxSetConfigEDMA](#)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is the master clock, this value should equal to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

29.5.3.5 void SAI_TransferTxSetConfigEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transceiver_t * *saiConfig*)

Note

SAI eDMA supports data transfer in multiple SAI channels if the FIFO Combine feature is supported. To activate the multi-channel transfer enable SAI channels by filling the channelMask of *sai_transceiver_t* with the corresponding values of *_sai_channel_mask* enum, enable the FIFO Combine mode by assigning *kSAI_FifoCombineModeEnabledOnWrite* to the *fifoCombine* member of *sai_fifo_combine_t* which is a member of *sai_transceiver_t*. This is an example of multi-channel data transfer configuration step.

```
*   sai_transceiver_t config;
*   SAI_GetClassicI2SConfig(&config, kSAI_WordWidth16bits,
*                           kSAI_Stereo, kSAI_Channel0Mask|kSAI_Channel1Mask);
*   config fifo.fifoCombine = kSAI_FifoCombineModeEnabledOnWrite;
*   SAI_TransferTxSetConfigEDMA(I2S0, &edmaHandle, &config);
*
```

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>saiConfig</i>	sai configurations.

29.5.3.6 void SAI_TransferRxSetConfigEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transceiver_t * *saiConfig*)

Note

SAI eDMA supports data transfer in a multiple SAI channels if the FIFO Combine feature is supported. To activate the multi-channel transfer enable SAI channels by filling the channelMask of *sai_transceiver_t* with the corresponding values of _sai_channel_mask enum, enable the FIFO Combine mode by assigning kSAI_FifoCombineModeEnabledOnRead to the fifoCombine member of *sai_fifo_combine_t* which is a member of *sai_transceiver_t*. This is an example of multi-channel data transfer configuration step.

```
*     sai_transceiver_t config;
*     SAI_GetClassicI2SConfig(&config, kSAI_WordWidth16bits,
*                             kSAI_Stereo, kSAI_Channel0Mask|kSAI_Channel1Mask);
*     configfifo.fifoCombine = kSAI_FifoCombineModeEnabledOnRead;
*     SAI_TransferRxSetConfigEDMA(I2S0, &edmaHandle, &config);
*
```

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>saiConfig</i>	sai configurations.

29.5.3.7 status_t SAI_TransferSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This interface returns immediately after the transfer initiates. Call SAI_GetTransferStatus to poll the transfer status and check whether the SAI transfer is finished.

This function support multi channel transfer,

1. for the sai IP support fifo combine mode, application should enable the fifo combine mode, no limitation on channel numbers
2. for the sai IP not support fifo combine mode, sai edma provide another solution which using EDMA modulo feature, but support 2 or 4 channels only.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to the DMA transfer structure.

Return values

<i>kStatus_Success</i>	Start a SAI eDMA send successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_TxBusy</i>	SAI is busy sending data.

29.5.3.8 status_t SAI_TransferReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This interface returns immediately after the transfer initiates. Call the SAI_GetReceiveRemainingBytes to poll the transfer status and check whether the SAI transfer is finished.

This function support multi channel transfer,

1. for the sai IP support fifo combine mode, application should enable the fifo combine mode, no limitation on channel numbers
2. for the sai IP not support fifo combine mode, sai edma provide another solution which using EDMA modulo feature, but support 2 or 4 channels only.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

<i>kStatus_Success</i>	Start a SAI eDMA receive successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

<i>kStatus_RxBusy</i>	SAI is busy receiving data.
-----------------------	-----------------------------

29.5.3.9 status_t SAI_TransferSendLoopEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_t * *xfer*, uint32_t *loopTransferCount*)

Note

This function support loop transfer only,such as A->B->...->A, application must be aware of that the more counts of the loop transfer, then more tcd memory required, as the function use the tcd pool in sai_edma_handle_t, so application could redefine the SAI_XFER_QUEUE_SIZE to determine the proper TCD pool size. This function support one sai channel only.

Once the loop transfer start, application can use function SAI_TransferAbortSendEDMA to stop the loop transfer.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to the DMA transfer structure, should be a array with elements counts ≥ 1 (loopTransferCount).
<i>loopTransferCount</i>	the counts of xfer array.

Return values

<i>kStatus_Success</i>	Start a SAI eDMA send successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

29.5.3.10 status_t SAI_TransferReceiveLoopEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_t * *xfer*, uint32_t *loopTransferCount*)

Note

This function support loop transfer only,such as A->B->...->A, application must be aware of that the more counts of the loop transfer, then more tcd memory required, as the function use the tcd pool in sai_edma_handle_t, so application could redefine the SAI_XFER_QUEUE_SIZE to determine the proper TCD pool size. This function support one sai channel only.

Once the loop transfer start, application can use function SAI_TransferAbortReceiveEDMA to stop the loop transfer.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to the DMA transfer structure, should be a array with elements counts ≥ 1 (loopTransferCount).
<i>loopTransfer-Count</i>	the counts of xfer array.

Return values

<i>kStatus_Success</i>	Start a SAI eDMA receive successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

29.5.3.11 void SAI_TransferTerminateSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortSendEDMA.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

29.5.3.12 void SAI_TransferTerminateReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

This function will clear all transfer slots buffered in the sai queue. If users only want to abort the current transfer slot, please call SAI_TransferAbortReceiveEDMA.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

29.5.3.13 void SAI_TransferAbortSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call SAI_TransferTerminateSendEDMA.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

29.5.3.14 void SAI_TransferAbortReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

This function only aborts the current transfer slots, the other transfer slots' information still kept in the handler. If users want to terminate all transfer slots, just call SAI_TransferTerminateReceiveEDMA.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.

29.5.3.15 status_t SAI_TransferGetSendCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count sent by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is no non-blocking transaction in progress.

29.5.3.16 status_t SAI_TransferGetReceiveCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferIn-Progress</i>	There is no non-blocking transaction in progress.

29.5.3.17 `uint32_t SAI_TransferGetValidTransferSlotsEDMA (I2S_Type * base, sai_edma_handle_t * handle)`

This function can be used to query the valid transfer request slot that the application can submit. It should be called in the critical section, that means the application could call it in the corresponding callback function or disable IRQ before calling it in the application, otherwise, the returned value may not correct.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.

Return values

<i>valid</i>	slot count that application submit.
--------------	-------------------------------------

Chapter 30

SEMA42: Hardware Semaphores Driver

30.1 Overview

The MCUXpresso SDK provides a driver for the SEMA42 module of MCUXpresso SDK devices.

The SEMA42 driver is used for multicore platforms. Before using the SEMA42, call the [SEMA42_Init\(\)](#) function to initialize the module. Note that this function only enables the clock but does not reset the gates because the module might be used by other processors at the same time. To reset the gates, call either the [SEMA42_ResetGate\(\)](#) or [SEMA42_ResetAllGates\(\)](#) functions. The function [SEMA42_Deinit\(\)](#) deinitializes the SEMA42.

The SEMA42 provides two functions to lock the SEMA42 gate. The function [SEMA42_TryLock\(\)](#) tries to lock the gate. If the gate has been locked by another processor, this function returns an error immediately. The function [SEMA42_Lock\(\)](#) is a blocking method, which waits until the gate is free and locks it.

The [SEMA42_Unlock\(\)](#) unlocks the SEMA42 gate. The gate can only be unlocked by the processor which locked it. If the gate is not locked by the current processor, this function takes no effect. The function [SEMA42_GetGateStatus\(\)](#) returns a status whether the gate is unlocked and which processor locks the gate.

The SEMA42 gate can be reset to unlock forcefully. The function [SEMA42_ResetGate\(\)](#) resets a specific gate. The function [SEMA42_ResetAllGates\(\)](#) resets all gates.

30.2 Typical use case

Refer to the driver examples codes located at <SDK_ROOT>/boards/<BOARD>/driver_examples/sema42

Macros

- #define [SEMA42_GATE_NUM_RESET_ALL](#) (64U)
The number to reset all SEMA42 gates.
- #define [SEMA42_GATEn](#)(base, n) (((volatile uint8_t *)(&((base)->GATE3)))[(n) ^ 3U])
SEMA42 gate n register address.

Enumerations

- enum {
 [kStatus_SEMA42_Busy](#) = MAKE_STATUS(kStatusGroup_SEMA42, 0),
 [kStatus_SEMA42_Reseting](#) = MAKE_STATUS(kStatusGroup_SEMA42, 1) }
SEMA42 status return codes.
- enum [sema42_gate_status_t](#) {

```

kSEMA42_Unlocked = 0U,
kSEMA42_LockedByProc0 = 1U,
kSEMA42_LockedByProc1 = 2U,
kSEMA42_LockedByProc2 = 3U,
kSEMA42_LockedByProc3 = 4U,
kSEMA42_LockedByProc4 = 5U,
kSEMA42_LockedByProc5 = 6U,
kSEMA42_LockedByProc6 = 7U,
kSEMA42_LockedByProc7 = 8U,
kSEMA42_LockedByProc8 = 9U,
kSEMA42_LockedByProc9 = 10U,
kSEMA42_LockedByProc10 = 11U,
kSEMA42_LockedByProc11 = 12U,
kSEMA42_LockedByProc12 = 13U,
kSEMA42_LockedByProc13 = 14U,
kSEMA42_LockedByProc14 = 15U }

```

SEMA42 gate lock status.

Functions

- void [SEMA42_Init](#) (SEMA42_Type *base)
Initializes the SEMA42 module.
- void [SEMA42_Deinit](#) (SEMA42_Type *base)
De-initializes the SEMA42 module.
- [status_t SEMA42_TryLock](#) (SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)
Tries to lock the SEMA42 gate.
- void [SEMA42_Lock](#) (SEMA42_Type *base, uint8_t gateNum, uint8_t procNum)
Locks the SEMA42 gate.
- static void [SEMA42_Unlock](#) (SEMA42_Type *base, uint8_t gateNum)
Unlocks the SEMA42 gate.
- static [sema42_gate_status_t SEMA42_GetGateStatus](#) (SEMA42_Type *base, uint8_t gateNum)
Gets the status of the SEMA42 gate.
- [status_t SEMA42_ResetGate](#) (SEMA42_Type *base, uint8_t gateNum)
Resets the SEMA42 gate to an unlocked status.
- static [status_t SEMA42_ResetAllGates](#) (SEMA42_Type *base)
Resets all SEMA42 gates to an unlocked status.

Driver version

- #define [FSL_SEMA42_DRIVER_VERSION](#) (MAKE_VERSION(2, 0, 3))
SEMA42 driver version.

30.3 Macro Definition Documentation

30.3.1 #define SEMA42_GATE_NUM_RESET_ALL (64U)

30.3.2 #define SEMA42_GATEEn(*base*, *n*) (((volatile uint8_t *)(&(*base*)->GATE3))[(*n*) ^ 3U])

The SEMA42 gates are sorted in the order 3, 2, 1, 0, 7, 6, 5, 4, ... not in the order 0, 1, 2, 3, 4, 5, 6, 7, ... The macro SEMA42_GATEEn gets the SEMA42 gate based on the gate index.

The input gate index is XOR'ed with 3U: $0 \wedge 3 = 3$ $1 \wedge 3 = 2$ $2 \wedge 3 = 1$ $3 \wedge 3 = 0$ $4 \wedge 3 = 7$ $5 \wedge 3 = 6$ $6 \wedge 3 = 5$ $7 \wedge 3 = 4$...

30.4 Enumeration Type Documentation

30.4.1 anonymous enum

Enumerator

kStatus_SEMA42_Busy SEMA42 gate has been locked by other processor.

kStatus_SEMA42_Reseting SEMA42 gate reseting is ongoing.

30.4.2 enum sema42_gate_status_t

Enumerator

kSEMA42_Unlocked The gate is unlocked.

kSEMA42_LockedByProc0 The gate is locked by processor 0.

kSEMA42_LockedByProc1 The gate is locked by processor 1.

kSEMA42_LockedByProc2 The gate is locked by processor 2.

kSEMA42_LockedByProc3 The gate is locked by processor 3.

kSEMA42_LockedByProc4 The gate is locked by processor 4.

kSEMA42_LockedByProc5 The gate is locked by processor 5.

kSEMA42_LockedByProc6 The gate is locked by processor 6.

kSEMA42_LockedByProc7 The gate is locked by processor 7.

kSEMA42_LockedByProc8 The gate is locked by processor 8.

kSEMA42_LockedByProc9 The gate is locked by processor 9.

kSEMA42_LockedByProc10 The gate is locked by processor 10.

kSEMA42_LockedByProc11 The gate is locked by processor 11.

kSEMA42_LockedByProc12 The gate is locked by processor 12.

kSEMA42_LockedByProc13 The gate is locked by processor 13.

kSEMA42_LockedByProc14 The gate is locked by processor 14.

30.5 Function Documentation

30.5.1 void SEMA42_Init (SEMA42_Type * *base*)

This function initializes the SEMA42 module. It only enables the clock but does not reset the gates because the module might be used by other processors at the same time. To reset the gates, call either SEMA42_ResetGate or SEMA42_ResetAllGates function.

Parameters

<i>base</i>	SEMA42 peripheral base address.
-------------	---------------------------------

30.5.2 void SEMA42_Deinit (SEMA42_Type * *base*)

This function de-initializes the SEMA42 module. It only disables the clock.

Parameters

<i>base</i>	SEMA42 peripheral base address.
-------------	---------------------------------

30.5.3 status_t SEMA42_TryLock (SEMA42_Type * *base*, uint8_t *gateNum*, uint8_t *procNum*)

This function tries to lock the specific SEMA42 gate. If the gate has been locked by another processor, this function returns an error code.

Parameters

<i>base</i>	SEMA42 peripheral base address.
<i>gateNum</i>	Gate number to lock.
<i>procNum</i>	Current processor number.

Return values

<i>kStatus_Success</i>	Lock the sema42 gate successfully.
<i>kStatus_SEMA42_Busy</i>	Sema42 gate has been locked by another processor.

30.5.4 void SEMA42_Lock (SEMA42_Type * *base*, uint8_t *gateNum*, uint8_t *procNum*)

This function locks the specific SEMA42 gate. If the gate has been locked by other processors, this function waits until it is unlocked and then lock it.

Parameters

<i>base</i>	SEMA42 peripheral base address.
<i>gateNum</i>	Gate number to lock.
<i>procNum</i>	Current processor number.

30.5.5 static void SEMA42_Unlock (SEMA42_Type * *base*, uint8_t *gateNum*) [inline], [static]

This function unlocks the specific SEMA42 gate. It only writes unlock value to the SEMA42 gate register. However, it does not check whether the SEMA42 gate is locked by the current processor or not. As a result, if the SEMA42 gate is not locked by the current processor, this function has no effect.

Parameters

<i>base</i>	SEMA42 peripheral base address.
<i>gateNum</i>	Gate number to unlock.

30.5.6 static sema42_gate_status_t SEMA42_GetGateStatus (SEMA42_Type * *base*, uint8_t *gateNum*) [inline], [static]

This function checks the lock status of a specific SEMA42 gate.

Parameters

<i>base</i>	SEMA42 peripheral base address.
<i>gateNum</i>	Gate number.

Returns

status Current status.

30.5.7 status_t SEMA42_ResetGate (SEMA42_Type * *base*, uint8_t *gateNum*)

This function resets a SEMA42 gate to an unlocked status.

Parameters

<i>base</i>	SEMA42 peripheral base address.
<i>gateNum</i>	Gate number.

Return values

<i>kStatus_Success</i>	SEMA42 gate is reset successfully.
<i>kStatus_SEMA42_Reseting</i>	Some other reset process is ongoing.

30.5.8 static status_t SEMA42_ResetAllGates (SEMA42_Type * *base*) [inline], [static]

This function resets all SEMA42 gate to an unlocked status.

Parameters

<i>base</i>	SEMA42 peripheral base address.
-------------	---------------------------------

Return values

<i>kStatus_Success</i>	SEMA42 is reset successfully.
<i>kStatus_SEMA42_Reseting</i>	Some other reset process is ongoing.

Chapter 31

SNVS: Secure Non-Volatile Storage

31.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Secure Non-Volatile Storage (SNVS) module.

The SNVS module is designed to safely hold security-related data such as cryptographic key, time counter, monotonic counter, and general purpose security information. The SNVS includes a low power section, namely SNVS_LP, that is battery backed by the SVNS (or VBAT) power domain. This enables it to keep this data valid and continue to increment the time counter when the power goes down in the rest of the SoC. The always-powered-up part of the module is isolated from the rest of the logic to ensure that it does not get corrupted when the SoC is powered down. The SNVS is designed to comply with Digital Rights Management (DRM) and other security application rules and requirements. This trusted hardware provides features that allow the system software designer to ensure that the data kept by the device is certifiable. Specially, it incorporates a security monitor that checks for various security conditions. If a security violation is indicated then it invalidates access to its sensitive data, and the secret data, for instance, Zeroizable Secret Key, is zeroized. the SNVS can be also configured to bypass its security features and protection mechanism. In this case it can be used by systems that do not require security.

Modules

- [Secure Non-Volatile Storage High-Power](#)
- [Secure Non-Volatile Storage Low-Power](#)

31.2 Secure Non-Volatile Storage High-Power

31.2.1 Overview

The MCUXpresso SDK provides a Peripheral driver for the Secure Non-Volatile Storage High-Power(S-NVS-HP) module.

The SNVS_HP is in the chip's power-supply domain and thus receives the power along with the rest of the chip. The SNVS_HP provides an interface between the SNVS_LP and the rest of the system; there is no way to access the SNVS_LP registers except through the SNVS_HP. For access to the SNVS_LP registers, the SNVS_HP must be powered up. It uses a register access permission policy to determine whether the access to the particular registers is permitted.

Data Structures

- struct `snvs_hp_rtc_datetime_t`
Structure is used to hold the date and time. [More...](#)
- struct `snvs_hp_rtc_config_t`
SNVS config structure. [More...](#)

Macros

- #define `SNVS_MAKE_HP_SV_FLAG(x)` (1U << (SNVS_HPSVR_SV0_SHIFT + (x)))
Macro to make security violation flag.

Enumerations

- enum `snvs_hp_interrupts_t` {

`kSNVS_RTC_AlarmInterrupt` = SNVS_HPCR_HPTA_EN_MASK,
`kSNVS_RTC_PeriodicInterrupt` = SNVS_HPCR_PI_EN_MASK }

List of SNVS interrupts.
- enum `snvs_hp_status_flags_t` {

`kSNVS_RTC_AlarmInterruptFlag` = SNVS_HPSR_HPTA_MASK,
`kSNVS_RTC_PeriodicInterruptFlag` = SNVS_HPSR_PI_MASK,
`kSNVS_ZMK_ZeroFlag` = (int)SNVS_HPSR_ZMK_ZERO_MASK,
`kSNVS_OTPMK_ZeroFlag` = SNVS_HPSR_OTPMK_ZERO_MASK }

List of SNVS flags.
- enum `snvs_hp_sv_status_flags_t` {

```

kSNVS_LP_ViolationFlag = SNVS_HPSVSR_SW_LPSV_MASK,
kSNVS_ZMK_EccFailFlag = SNVS_HPSVSR_ZMK_ECC_FAIL_MASK,
kSNVS_LP_SoftwareViolationFlag = SNVS_HPSVSR_SW_LPSV_MASK,
kSNVS_FatalSoftwareViolationFlag = SNVS_HPSVSR_SW_FSV_MASK,
kSNVS_SoftwareViolationFlag = SNVS_HPSVSR_SW_SV_MASK,
kSNVS_Violation0Flag = SNVS_HPSVSR_SV0_MASK,
kSNVS_Violation1Flag = SNVS_HPSVSR_SV1_MASK,
kSNVS_Violation2Flag = SNVS_HPSVSR_SV2_MASK,
kSNVS_Violation4Flag = SNVS_HPSVSR_SV4_MASK,
kSNVS_Violation5Flag = SNVS_HPSVSR_SV5_MASK }

```

List of SNVS security violation flags.

- enum `snvs_hp_ssm_state_t` {

kSNVS_SSMInit = 0x00,
 kSNVS_SSMHardFail = 0x01,
 kSNVS_SSMSoftFail = 0x03,
 kSNVS_SSMInitInter = 0x08,
 kSNVS_SSMCheck = 0x09,
 kSNVS_SSMNonSecure = 0x0B,
 kSNVS_SSMTrusted = 0x0D,
 kSNVS_SSMSecure = 0x0F }

List of SNVS Security State Machine State.

Functions

- static void `SNVS_HP_EnableMasterKeySelection` (SNVS_Type *base, bool enable)
Enable or disable master key selection.
- static void `SNVS_HP_ProgramZeroizableMasterKey` (SNVS_Type *base)
Trigger to program Zeroizable Master Key.
- static void `SNVS_HP_ChangeSSMState` (SNVS_Type *base)
Trigger SSM State Transition.
- static void `SNVS_HP_SetSoftwareFatalSecurityViolation` (SNVS_Type *base)
Trigger Software Fatal Security Violation.
- static void `SNVS_HP_SetSoftwareSecurityViolation` (SNVS_Type *base)
Trigger Software Security Violation.
- static `snvs_hp_ssm_state_t` `SNVS_HP_GetSSMState` (SNVS_Type *base)
Get current SSM State.
- static void `SNVS_HP_ResetLP` (SNVS_Type *base)
Reset the SNVS LP section.
- static uint32_t `SNVS_HP_GetStatusFlags` (SNVS_Type *base)
Get the SNVS HP status flags.
- static void `SNVS_HP_ClearStatusFlags` (SNVS_Type *base, uint32_t mask)
Clear the SNVS HP status flags.
- static uint32_t `SNVS_HP_GetSecurityViolationStatusFlags` (SNVS_Type *base)
Get the SNVS HP security violation status flags.
- static void `SNVS_HP_ClearSecurityViolationStatusFlags` (SNVS_Type *base, uint32_t mask)
Clear the SNVS HP security violation status flags.

Driver version

- #define `FSL_SNVS_HP_DRIVER_VERSION` (`MAKE_VERSION(2, 3, 2)`)
Version 2.3.2.

Initialization and deinitialization

- void `SNVS_HP_Init` (SNVS_Type *base)
Initialize the SNVS.
- void `SNVS_HP_Deinit` (SNVS_Type *base)
Deinitialize the SNVS.
- void `SNVS_HP_RTC_Init` (SNVS_Type *base, const `snvs_hp_rtc_config_t` *config)
Ungates the SNVS clock and configures the peripheral for basic operation.
- void `SNVS_HP_RTC_Deinit` (SNVS_Type *base)
Stops the RTC and SRTC timers.
- void `SNVS_HP_RTC_GetDefaultConfig` (`snvs_hp_rtc_config_t` *config)
Fills in the SNVS config struct with the default settings.

Non secure RTC current Time & Alarm

- `status_t SNVS_HP_RTC_SetDatetime` (SNVS_Type *base, const `snvs_hp_rtc_datetime_t` *datetime)
Sets the SNVS RTC date and time according to the given time structure.
- void `SNVS_HP_RTC_GetDatetime` (SNVS_Type *base, `snvs_hp_rtc_datetime_t` *datetime)
Gets the SNVS RTC time and stores it in the given time structure.
- `status_t SNVS_HP_RTC_SetAlarm` (SNVS_Type *base, const `snvs_hp_rtc_datetime_t` *alarmTime)
Sets the SNVS RTC alarm time.
- void `SNVS_HP_RTC_GetAlarm` (SNVS_Type *base, `snvs_hp_rtc_datetime_t` *datetime)
Returns the SNVS RTC alarm time.

Interrupt Interface

- static void `SNVS_HP_RTC_EnableInterrupts` (SNVS_Type *base, uint32_t mask)
Enables the selected SNVS interrupts.
- static void `SNVS_HP_RTC_DisableInterrupts` (SNVS_Type *base, uint32_t mask)
Disables the selected SNVS interrupts.
- uint32_t `SNVS_HP_RTC_GetEnabledInterrupts` (SNVS_Type *base)
Gets the enabled SNVS interrupts.

Status Interface

- uint32_t `SNVS_HP_RTC_GetStatusFlags` (SNVS_Type *base)
Gets the SNVS status flags.
- static void `SNVS_HP_RTC_ClearStatusFlags` (SNVS_Type *base, uint32_t mask)

Clears the SNVS status flags.

Timer Start and Stop

- static void [SNVS_HP_RTC_StartTimer](#) (SNVS_Type *base)
Starts the SNVS RTC time counter.
- static void [SNVS_HP_RTC_StopTimer](#) (SNVS_Type *base)
Stops the SNVS RTC time counter.

High Assurance Counter (HAC)

- static void [SNVS_HP_EnableHighAssuranceCounter](#) (SNVS_Type *base, bool enable)
Enable or disable the High Assurance Counter (HAC)
- static void [SNVS_HP_StartHighAssuranceCounter](#) (SNVS_Type *base, bool start)
Start or stop the High Assurance Counter (HAC)
- static void [SNVS_HP_SetHighAssuranceCounterInitialValue](#) (SNVS_Type *base, uint32_t value)
Set the High Assurance Counter (HAC) initialize value.
- static void [SNVS_HP_LoadHighAssuranceCounter](#) (SNVS_Type *base)
Load the High Assurance Counter (HAC)
- static uint32_t [SNVS_HP_GetHighAssuranceCounter](#) (SNVS_Type *base)
Get the current High Assurance Counter (HAC) value.
- static void [SNVS_HP_ClearHighAssuranceCounter](#) (SNVS_Type *base)
Clear the High Assurance Counter (HAC)
- static void [SNVS_HP_LockHighAssuranceCounter](#) (SNVS_Type *base)
Lock the High Assurance Counter (HAC)

31.2.2 Data Structure Documentation

31.2.2.1 struct snvs_hp_rtc_datetime_t

Data Fields

- uint16_t [year](#)
Range from 1970 to 2099.
- uint8_t [month](#)
Range from 1 to 12.
- uint8_t [day](#)
Range from 1 to 31 (depending on month).
- uint8_t [hour](#)
Range from 0 to 23.
- uint8_t [minute](#)
Range from 0 to 59.
- uint8_t [second](#)
Range from 0 to 59.

Field Documentation

- (1) `uint16_t snvs_hp_rtc_datetime_t::year`
- (2) `uint8_t snvs_hp_rtc_datetime_t::month`
- (3) `uint8_t snvs_hp_rtc_datetime_t::day`
- (4) `uint8_t snvs_hp_rtc_datetime_t::hour`
- (5) `uint8_t snvs_hp_rtc_datetime_t::minute`
- (6) `uint8_t snvs_hp_rtc_datetime_t::second`

31.2.2.2 struct snvs_hp_rtc_config_t

This structure holds the configuration settings for the SNVS peripheral. To initialize this structure to reasonable defaults, call the SNVS_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Data Fields

- bool `rtcCalEnable`
true: RTC calibration mechanism is enabled; false: No calibration is used
- `uint32_t rtcCalValue`
Defines signed calibration value for nonsecure RTC; This is a 5-bit 2's complement value, range from -16 to +15.
- `uint32_t periodicInterruptFreq`
Defines frequency of the periodic interrupt; Range from 0 to 15.

31.2.3 Macro Definition Documentation

31.2.3.1 #define SNVS_MAKE_HP_SV_FLAG(x) (1U << (SNVS_HPSVSR_SV0_SHIFT + (x)))

Macro help to make security violation flag kSNVS_Violation0Flag to kSNVS_Violation5Flag, For example, `SNVS_MAKE_HP_SV_FLAG(0)` is kSNVS_Violation0Flag.

31.2.4 Enumeration Type Documentation

31.2.4.1 enum snvs_hp_interrupts_t

Enumerator

kSNVS_RTC_AlarmInterrupt RTC time alarm.

kSNVS_RTC_PeriodicInterrupt RTC periodic interrupt.

31.2.4.2 enum snvs_hp_status_flags_t

Enumerator

kSNVS_RTC_AlarmInterruptFlag RTC time alarm flag.

kSNVS_RTC_PeriodicInterruptFlag RTC periodic interrupt flag.

kSNVS_ZMK_ZeroFlag The ZMK is zero.

kSNVS OTPMK_ZeroFlag The OTPMK is zero.

31.2.4.3 enum snvs_hp_sv_status_flags_t

Enumerator

kSNVS_LP_ViolationFlag Low Power section Security Violation.

kSNVS_ZMK_EccFailFlag Zeroizable Master Key Error Correcting Code Check Failure.

kSNVS_LP_SoftwareViolationFlag LP Software Security Violation.

kSNVS_FatalSoftwareViolationFlag Software Fatal Security Violation.

kSNVS_SoftwareViolationFlag Software Security Violation.

kSNVS_Violation0Flag Security Violation 0.

kSNVS_Violation1Flag Security Violation 1.

kSNVS_Violation2Flag Security Violation 2.

kSNVS_Violation4Flag Security Violation 4.

kSNVS_Violation5Flag Security Violation 5.

31.2.4.4 enum snvs_hp_ssm_state_t

Enumerator

kSNVS_SSMInit Init.

kSNVS_SSMHardFail Hard Fail.

kSNVS_SSSoftFail Soft Fail.

kSNVS_SSMInitInter Init Intermediate (transition state between Init and Check)

kSNVS_SSMCheck Check.

kSNVS_SSMNonSecure Non-Secure.

kSNVS_SSMTrusted Trusted.

kSNVS_SSMSecure Secure.

31.2.5 Function Documentation

31.2.5.1 void SNVS_HP_Init (SNVS_Type * *base*)

Note

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

Parameters

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

31.2.5.2 void SNVS_HP_Deinit (SNVS_Type * *base*)

Parameters

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

31.2.5.3 void SNVS_HP_RTC_Init (SNVS_Type * *base*, const snvs_hp_rtc_config_t * *config*)

Note

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

Parameters

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

31.2.5.4 void SNVS_HP_RTC_Deinit (SNVS_Type * *base*)

Parameters

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

31.2.5.5 void SNVS_HP_RTC_GetDefaultConfig (snvs_hp_rtc_config_t * *config*)

The default values are as follows.

```
*     config->rtccalenable = false;
*     config->rtccalvalue = 0U;
*     config->PIFreq = 0U;
*
```

Parameters

<i>config</i>	Pointer to the user's SNVS configuration structure.
---------------	---

31.2.5.6 status_t SNVS_HP_RTC_SetDatetime (SNVS_Type * *base*, const snvs_hp_rtc_datetime_t * *datetime*)

Parameters

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

Returns

kStatus_Success: Success in setting the time and starting the SNVS RTC
kStatus_InvalidArgument: Error because the datetime format is incorrect

31.2.5.7 void SNVS_HP_RTC_GetDatetime (SNVS_Type * *base*, snvs_hp_rtc_datetime_t * *datetime*)

Parameters

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

31.2.5.8 status_t SNVS_HP_RTC_SetAlarm (SNVS_Type * *base*, const snvs_hp_rtc_datetime_t * *alarmTime*)

The function sets the RTC alarm. It also checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

<i>base</i>	SNVS peripheral base address
<i>alarmTime</i>	Pointer to the structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the SNVS RTC alarm
kStatus_InvalidArgument: Error because the alarm datetime format is incorrect
kStatus_Fail: Error because the alarm time has already passed

31.2.5.9 void SNVS_HP_RTC_GetAlarm (SNVS_Type * *base*, snvs_hp_RTC_datetime_t * *datetime*)

Parameters

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

31.2.5.10 static void SNVS_HP_RTC_EnableInterrupts (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration :: _snvs_hp_interrupts_t

31.2.5.11 static void SNVS_HP_RTC_DisableInterrupts (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

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

<i>mask</i>	The interrupts to disable. This is a logical OR of members of the enumeration :: _snvs_hp_interrupts_t
-------------	--

31.2.5.12 uint32_t SNVS_HP_RTC_GetEnabledInterrupts (SNVS_Type * *base*)

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration :: _snvs_hp_interrupts_t

31.2.5.13 uint32_t SNVS_HP_RTC_GetStatusFlags (SNVS_Type * *base*)

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration :: _snvs_hp_status_flags_t

31.2.5.14 static void SNVS_HP_RTC_ClearStatusFlags (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration :: _snvs_hp_status_flags_t

31.2.5.15 static void SNVS_HP_RTC_StartTimer (SNVS_Type * *base*) [inline], [static]

Parameters

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

31.2.5.16 static void SNVS_HP_RTC_StopTimer (SNVS_Type * *base*) [inline], [static]

Parameters

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

31.2.5.17 static void SNVS_HP_EnableMasterKeySelection (SNVS_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

31.2.5.18 static void SNVS_HP_ProgramZeroizableMasterKey (SNVS_Type * *base*) [inline], [static]

Parameters

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

31.2.5.19 static void SNVS_HP_ChangeSSMState (SNVS_Type * *base*) [inline], [static]

Trigger state transition of the system security monitor (SSM). It results only the following transitions of the SSM:

- Check State -> Non-Secure (when Non-Secure Boot and not in Fab Configuration)
- Check State -> Trusted (when Secure Boot or in Fab Configuration)
- Trusted State -> Secure
- Secure State -> Trusted
- Soft Fail -> Non-Secure

Parameters

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

31.2.5.20 static void SNVS_HP_SetSoftwareFatalSecurityViolation (SNVS_Type * *base*) [inline], [static]

The result SSM state transition is:

- Check State -> Soft Fail
- Non-Secure State -> Soft Fail
- Trusted State -> Soft Fail
- Secure State -> Soft Fail

Parameters

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

31.2.5.21 static void SNVS_HP_SetSoftwareSecurityViolation (SNVS_Type * *base*) [inline], [static]

The result SSM state transition is:

- Check -> Non-Secure
- Trusted -> Soft Fail
- Secure -> Soft Fail

Parameters

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

31.2.5.22 static snvs_hp_ssm_state_t SNVS_HP_GetSSMState (SNVS_Type * *base*) [inline], [static]

Parameters

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

Returns

Current SSM state

31.2.5.23 static void SNVS_HP_ResetLP (SNVS_Type * *base*) [inline], [static]

Reset the LP section except SRTC and Time alarm.

Parameters

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

31.2.5.24 static void SNVS_HP_EnableHighAssuranceCounter (SNVS_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

31.2.5.25 static void SNVS_HP_StartHighAssuranceCounter (SNVS_Type * *base*, bool *start*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>start</i>	Pass true to start, false to stop.

31.2.5.26 static void SNVS_HP_SetHighAssuranceCounterInitialValue (SNVS_Type * *base*, uint32_t *value*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>value</i>	The initial value to set.

31.2.5.27 static void SNVS_HP_LoadHighAssuranceCounter (SNVS_Type * *base*) [inline], [static]

This function loads the HAC initialize value to counter register.

Parameters

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

31.2.5.28 **static uint32_t SNVS_HP_GetHighAssuranceCounter (SNVS_Type * *base*)**
[**inline**], [**static**]

Parameters

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

Returns

HAC currnet value.

31.2.5.29 static void SNVS_HP_ClearHighAssuranceCounter (SNVS_Type * *base*) [inline], [static]

This function can be called in a functional or soft fail state. When the HAC is enabled:

- If the HAC is cleared in the soft fail state, the SSM transitions to the hard fail state immediately;
- If the HAC is cleared in functional state, the SSM will transition to hard fail immediately after transitioning to soft fail.

Parameters

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

31.2.5.30 static void SNVS_HP_LockHighAssuranceCounter (SNVS_Type * *base*) [inline], [static]

Once locked, the HAC initialize value could not be changed, the HAC enable status could not be changed. This could only be unlocked by system reset.

Parameters

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

31.2.5.31 static uint32_t SNVS_HP_GetStatusFlags (SNVS_Type * *base*) [inline], [static]

The flags are returned as the OR'ed value f the enumeration :: _snvs_hp_status_flags_t.

Parameters

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

Returns

The OR'ed value of status flags.

31.2.5.32 static void SNVS_HP_ClearStatusFlags (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]

The flags to clear are passed in as the OR'ed value of the enumeration :: _snvs_hp_status_flags_t. Only these flags could be cleared using this API.

- [kSNVS_RTC_PeriodicInterruptFlag](#)
- [kSNVS_RTC_AlarmInterruptFlag](#)

Parameters

<i>base</i>	SNVS peripheral base address
<i>mask</i>	OR'ed value of the flags to clear.

31.2.5.33 static uint32_t SNVS_HP_GetSecurityViolationStatusFlags (SNVS_Type * *base*) [inline], [static]

The flags are returned as the OR'ed value of the enumeration :: _snvs_hp_sv_status_flags_t.

Parameters

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

Returns

The OR'ed value of security violation status flags.

31.2.5.34 static void SNVS_HP_ClearSecurityViolationStatusFlags (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]

The flags to clear are passed in as the OR'ed value of the enumeration :: _snvs_hp_sv_status_flags_t. Only these flags could be cleared using this API.

- [kSNVS_ZMK_EccFailFlag](#)
- [kSNVS_Violation0Flag](#)
- [kSNVS_Violation1Flag](#)
- [kSNVS_Violation2Flag](#)
- [kSNVS_Violation3Flag](#)
- [kSNVS_Violation4Flag](#)
- [kSNVS_Violation5Flag](#)

Parameters

<i>base</i>	SNVS peripheral base address
<i>mask</i>	OR'ed value of the flags to clear.

31.3 Secure Non-Volatile Storage Low-Power

31.3.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Secure Non-Volatile Storage Low-Power (S-NVS-LP) module.

The SNVS_LP is a data storage subsystem. Its purpose is to store and protect system data, regardless of the main system power state. The SNVS_LP is in the always-powered-up domain, which is a separate power domain with its own power supply.

Data Structures

- struct `snvs_lp_passive_tamper_t`
Structure is used to configure SNVS LP passive tamper pins. [More...](#)
- struct `snvs_lp_srtc_datetime_t`
Structure is used to hold the date and time. [More...](#)
- struct `snvs_lp_srtc_config_t`
SNVS_LP config structure. [More...](#)

Macros

- #define `SNVS_ZMK_REG_COUNT` 8U /* 8 Zeroizable Master Key registers. */
Define of SNVS_LP Zeroizable Master Key registers.
- #define `SNVS_LP_MAX_TAMPER` kSNVS_ExternalTamper1
Define of SNVS_LP Max possible tamper.

Enumerations

- enum `snvs_lp_srtc_interrupts_t` { `kSNVS_SRTC_AlarmInterrupt` = SNVS_LPCR_LPTA_EN_M-ASK }
List of SNVS_LP interrupts.
- enum `snvs_lp_srtc_status_flags_t` { `kSNVS_SRTC_AlarmInterruptFlag` = SNVS_LPSR_LPTA-MASK }
List of SNVS_LP flags.
- enum `snvs_lp_external_tamper_status_t`
List of SNVS_LP external tampers status.
- enum `snvs_lp_external_tamper_polarity_t`
SNVS_LP external tamper polarity.
- enum `snvs_lp_zmk_program_mode_t` {
`kSNVS_ZMKSoftwareProgram`,
`kSNVS_ZMKHardwareProgram` }
SNVS_LP Zeroizable Master Key programming mode.
- enum `snvs_lp_master_key_mode_t` {
`kSNVS OTPMK` = 0,
`kSNVS_ZMK` = 2,

```
kSNVS_CMK = 3 }

SNVS_LP Master Key mode.
```

Functions

- void **SNVS_LP_SRTC_Init** (SNVS_Type *base, const **snvs_lp_srtc_config_t** *config)
Ungates the SNVS clock and configures the peripheral for basic operation.
- void **SNVS_LP_SRTC_Deinit** (SNVS_Type *base)
Stops the SRTC timer.
- void **SNVS_LP_SRTC_GetDefaultConfig** (**snvs_lp_srtc_config_t** *config)
Fills in the SNVS_LP config struct with the default settings.

Driver version

- #define **FSL_SNVS_LP_DRIVER_VERSION** (**MAKE_VERSION**(2, 4, 5))
Version 2.4.5.

Initialization and deinitialization

- void **SNVS_LP_Init** (SNVS_Type *base)
Ungates the SNVS clock and configures the peripheral for basic operation.
- void **SNVS_LP_Deinit** (SNVS_Type *base)
Deinit the SNVS LP section.

Secure RTC (SRTC) current Time & Alarm

- **status_t SNVS_LP_SRTC_SetDatetime** (SNVS_Type *base, const **snvs_lp_srtc_datetime_t** *datetime)
Sets the SNVS SRTC date and time according to the given time structure.
- void **SNVS_LP_SRTC_GetDatetime** (SNVS_Type *base, **snvs_lp_srtc_datetime_t** *datetime)
Gets the SNVS SRTC time and stores it in the given time structure.
- **status_t SNVS_LP_SRTC_SetAlarm** (SNVS_Type *base, const **snvs_lp_srtc_datetime_t** *alarmTime)
Sets the SNVS SRTC alarm time.
- void **SNVS_LP_SRTC_GetAlarm** (SNVS_Type *base, **snvs_lp_srtc_datetime_t** *datetime)
Returns the SNVS SRTC alarm time.

Interrupt Interface

- static void **SNVS_LP_SRTC_EnableInterrupts** (SNVS_Type *base, uint32_t mask)
Enables the selected SNVS interrupts.
- static void **SNVS_LP_SRTC_DisableInterrupts** (SNVS_Type *base, uint32_t mask)
Disables the selected SNVS interrupts.
- uint32_t **SNVS_LP_SRTC_GetEnabledInterrupts** (SNVS_Type *base)

Gets the enabled SNVS interrupts.

Status Interface

- `uint32_t SNVS_LP_SRTC_GetStatusFlags (SNVS_Type *base)`
Gets the SNVS status flags.
- `static void SNVS_LP_SRTC_ClearStatusFlags (SNVS_Type *base, uint32_t mask)`
Clears the SNVS status flags.

Timer Start and Stop

- `static void SNVS_LP_SRTC_StartTimer (SNVS_Type *base)`
Starts the SNVS SRTC time counter.
- `static void SNVS_LP_SRTC_StopTimer (SNVS_Type *base)`
Stops the SNVS SRTC time counter.

External tampering

- `void SNVS_LP_PassiveTamperPin_GetDefaultConfig (snvs_lp_passive_tamper_t *config)`
Fills in the SNVS tamper pin config struct with the default settings.

Monotonic Counter (MC)

- `static void SNVS_LP_EnableMonotonicCounter (SNVS_Type *base, bool enable)`
Enable or disable the Monotonic Counter.
- `uint64_t SNVS_LP_GetMonotonicCounter (SNVS_Type *base)`
Get the current Monotonic Counter.
- `static void SNVS_LP_IncreaseMonotonicCounter (SNVS_Type *base)`
Increase the Monotonic Counter.

Zeroizable Master Key (ZMK)

- `void SNVS_LP_WriteZeroizableMasterKey (SNVS_Type *base, uint32_t ZMKey[SNVS_ZMK_REG_COUNT])`
Write Zeroizable Master Key (ZMK) to the SNVS registers.
- `static void SNVS_LP_SetZeroizableMasterKeyValid (SNVS_Type *base, bool valid)`
Set Zeroizable Master Key valid.
- `static bool SNVS_LP_GetZeroizableMasterKeyValid (SNVS_Type *base)`
Get Zeroizable Master Key valid status.
- `static void SNVS_LP_SetZeroizableMasterKeyProgramMode (SNVS_Type *base, snvs_lp_zmk_program_mode_t mode)`
Set Zeroizable Master Key programming mode.
- `static void SNVS_LP_EnableZeroizableMasterKeyECC (SNVS_Type *base, bool enable)`
Enable or disable Zeroizable Master Key ECC.

- static void `SNVS_LP_SetMasterKeyMode` (SNVS_Type *base, `snvs_lp_master_key_mode_t mode`)
Set SNVS Master Key mode.

31.3.2 Data Structure Documentation

31.3.2.1 `struct snvs_lp_passive_tamper_t`

31.3.2.2 `struct snvs_lp_srtc_datetime_t`

Data Fields

- `uint16_t year`
Range from 1970 to 2099.
- `uint8_t month`
Range from 1 to 12.
- `uint8_t day`
Range from 1 to 31 (depending on month).
- `uint8_t hour`
Range from 0 to 23.
- `uint8_t minute`
Range from 0 to 59.
- `uint8_t second`
Range from 0 to 59.

Field Documentation

- (1) `uint16_t snvs_lp_srtc_datetime_t::year`
- (2) `uint8_t snvs_lp_srtc_datetime_t::month`
- (3) `uint8_t snvs_lp_srtc_datetime_t::day`
- (4) `uint8_t snvs_lp_srtc_datetime_t::hour`
- (5) `uint8_t snvs_lp_srtc_datetime_t::minute`
- (6) `uint8_t snvs_lp_srtc_datetime_t::second`

31.3.2.3 `struct snvs_lp_srtc_config_t`

This structure holds the configuration settings for the SNVS_LP peripheral. To initialize this structure to reasonable defaults, call the `SNVS_LP_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Data Fields

- bool `srtcCalEnable`
true: SRTC calibration mechanism is enabled; false: No calibration is used
- uint32_t `srtcCalValue`
Defines signed calibration value for SRTC; This is a 5-bit 2's complement value, range from -16 to +15.

31.3.3 Enumeration Type Documentation

31.3.3.1 enum snvs_lp_srtc_interrupts_t

Enumerator

kSNVS_SRTC_AlarmInterrupt SRTC time alarm.

31.3.3.2 enum snvs_lp_srtc_status_flags_t

Enumerator

kSNVS_SRTC_AlarmInterruptFlag SRTC time alarm flag.

31.3.3.3 enum snvs_lp_zmk_program_mode_t

Enumerator

kSNVS_ZMKSoftwareProgram Software programming mode.

kSNVS_ZMKHardwareProgram Hardware programming mode.

31.3.3.4 enum snvs_lp_master_key_mode_t

Enumerator

kSNVS_OTPMK One Time Programmable Master Key.

kSNVS_ZMK Zeroizable Master Key.

kSNVS_CMK Combined Master Key, it is XOR of OTPMK and ZMK.

31.3.4 Function Documentation

31.3.4.1 void SNVS_LP_Init(SNVS_Type * *base*)

Note

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

Parameters

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

31.3.4.2 void SNVS_LP_Deinit (SNVS_Type * *base*)

Parameters

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

31.3.4.3 void SNVS_LP_SRTC_Init (SNVS_Type * *base*, const snvs_lp_srtc_config_t * *config*)

Note

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

Parameters

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

31.3.4.4 void SNVS_LP_SRTC_Deinit (SNVS_Type * *base*)

Parameters

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

31.3.4.5 void SNVS_LP_SRTC_GetDefaultConfig (snvs_lp_srtc_config_t * *config*)

The default values are as follows.

```
*     config->srtccalenable = false;
*     config->srtccalvalue = 0U;
*
```

Parameters

<i>config</i>	Pointer to the user's SNVS configuration structure.
---------------	---

31.3.4.6 status_t SNVS_LP_SRTC_SetDatetime (SNVS_Type * *base*, const snvs_lp_srtc_datetime_t * *datetime*)

Parameters

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

Returns

kStatus_Success: Success in setting the time and starting the SNVS SRTC
kStatus_InvalidArgument: Error because the datetime format is incorrect

31.3.4.7 void SNVS_LP_SRTC_GetDatetime (SNVS_Type * *base*, snvs_lp_srtc_datetime_t * *datetime*)

Parameters

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

31.3.4.8 status_t SNVS_LP_SRTC_SetAlarm (SNVS_Type * *base*, const snvs_lp_srtc_datetime_t * *alarmTime*)

The function sets the SRTC alarm. It also checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error. Please note, that SRTC alarm has limited resolution because only 32 most significant bits of SRTC counter are compared to SRTC Alarm register. If the alarm time is beyond SRTC resolution, the function does not set the alarm and returns an error.

Parameters

<i>base</i>	SNVS peripheral base address
<i>alarmTime</i>	Pointer to the structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the SNVS SRTC alarm
kStatus_InvalidArgument: Error because the alarm datetime format is incorrect
kStatus_Fail: Error because the alarm time has already passed or is beyond resolution

31.3.4.9 void SNVS_LP_SRTC_GetAlarm (SNVS_Type * *base*, snvs_lp_srtc_datetime_t * *datetime*)

Parameters

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

31.3.4.10 static void SNVS_LP_SRTC_EnableInterrupts (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration :: _-snvs_lp_srtc_interrupts

31.3.4.11 static void SNVS_LP_SRTC_DisableInterrupts (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration :: _-snvs_lp_srtc_interrupts

31.3.4.12 uint32_t SNVS_LP_SRTC_GetEnabledInterrupts (SNVS_Type * *base*)

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration :: _snvs_lp_srtc_-interrupts

31.3.4.13 **uint32_t SNVS_LP_SRTC_GetStatusFlags (SNVS_Type * *base*)**

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration :: _snvs_lp_srtc_status_flags

31.3.4.14 **static void SNVS_LP_SRTC_ClearStatusFlags (SNVS_Type * *base*, uint32_t *mask*) [inline], [static]**

Parameters

<i>base</i>	SNVS peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration :: _-snvs_lp_srtc_status_flags

31.3.4.15 **static void SNVS_LP_SRTC_StartTimer (SNVS_Type * *base*) [inline], [static]**

Parameters

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

31.3.4.16 **static void SNVS_LP_SRTC_StopTimer (SNVS_Type * *base*) [inline], [static]**

Parameters

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

31.3.4.17 void SNVS_LP_PassiveTamperPin_GetDefaultConfig (**snvs_lp_passive_tamper_t** * *config*)

The default values are as follows. code config->polarity = 0U; config->filterenable = 0U; if available on SoC config->filter = 0U; if available on SoC endcode

Parameters

<i>config</i>	Pointer to the user's SNVS configuration structure.
---------------	---

31.3.4.18 static void SNVS_LP_EnableMonotonicCounter (**SNVS_Type** * *base*, **bool enable**) [inline], [static]

Parameters

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

31.3.4.19 uint64_t SNVS_LP_GetMonotonicCounter (**SNVS_Type** * *base*)

Parameters

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

Returns

Current Monotonic Counter value.

31.3.4.20 static void SNVS_LP_IncreaseMonotonicCounter (**SNVS_Type** * *base*) [inline], [static]

Increase the Monotonic Counter by 1.

Parameters

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

31.3.4.21 void SNVS_LP_WriteZeroizableMasterKey (SNVS_Type * *base*, uint32_t *ZMKey*[SNVS_ZMK_REG_COUNT])

Parameters

<i>base</i>	SNVS peripheral base address
<i>ZMKey</i>	The ZMK write to the SNVS register.

31.3.4.22 static void SNVS_LP_SetZeroizableMasterKeyValid (SNVS_Type * *base*, bool *valid*) [inline], [static]

This API could only be called when using software programming mode. After writing ZMK using [SNVS_LP_WriteZeroizableMasterKey](#), call this API to make the ZMK valid.

Parameters

<i>base</i>	SNVS peripheral base address
<i>valid</i>	Pass true to set valid, false to set invalid.

31.3.4.23 static bool SNVS_LP_GetZeroizableMasterKeyValid (SNVS_Type * *base*) [inline], [static]

In hardware programming mode, call this API to check whether the ZMK is valid.

Parameters

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

Returns

true if valid, false if invalid.

31.3.4.24 static void SNVS_LP_SetZeroizableMasterKeyProgramMode (SNVS_Type * *base*, snvs_lp_zmk_program_mode_t *mode*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>mode</i>	ZMK programming mode.

31.3.4.25 static void SNVS_LP_EnableZeroizableMasterKeyECC (SNVS_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

31.3.4.26 static void SNVS_LP_SetMasterKeyMode (SNVS_Type * *base*, snvs_lp_master_key_mode_t *mode*) [inline], [static]

Parameters

<i>base</i>	SNVS peripheral base address
<i>mode</i>	Master Key mode.

Note

When [kSNVS_ZMK](#) or [kSNVS_CMK](#) used, the SNVS_HP must be configured to enable the master key selection.

Chapter 32

TPM: Timer PWM Module

32.1 Overview

The MCUXpresso SDK provides a driver for the Timer PWM Module (TPM) of MCUXpresso SDK devices.

The TPM driver supports the generation of PWM signals, input capture, and output compare modes. On some SoCs, the driver supports the generation of combined PWM signals, dual-edge capture, and quadrature decoder modes. The driver also supports configuring each of the TPM fault inputs. The fault input is available only on some SoCs.

32.2 Introduction of TPM

32.2.1 Initialization and deinitialization

The function [TPM_Init\(\)](#) initializes the TPM with a specified configurations. The function [TPM_GetDefaultConfig\(\)](#) gets the default configurations. On some SoCs, the initialization function issues a software reset to reset the TPM internal logic. The initialization function configures the TPM's behavior when it receives a trigger input and its operation in doze and debug modes.

The function [TPM_Deinit\(\)](#) disables the TPM counter and turns off the module clock.

32.2.2 PWM Operations

The function [TPM_SetupPwm\(\)](#) sets up TPM channels for the PWM output. The function can set up the PWM signal properties for multiple channels. Each channel has its own [tpm_chnl_pwm_signal_param_t](#) structure that is used to specify the output signals duty cycle and level-mode. However, the same PWM period and PWM mode is applied to all channels requesting a PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 where 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle). When generating a combined PWM signal, the channel number passed refers to a channel pair number, for example 0 refers to channel 0 and 1, 1 refers to channels 2 and 3.

The function [TPM_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular TPM channel.

The function [TPM_UpdateChnlEdgeLevelSelect\(\)](#) updates the level select bits of a particular TPM channel. This can be used to disable the PWM output when making changes to the PWM signal.

32.2.3 Input capture operations

The function [TPM_SetupInputCapture\(\)](#) sets up a TPM channel for input capture. The user can specify the capture edge.

The function [TPM_SetupDualEdgeCapture\(\)](#) can be used to measure the pulse width of a signal. This is available only for certain SoCs. A channel pair is used during the capture with the input signal coming through a channel that can be configured. The user can specify the capture edge for each channel and any filter value to be used when processing the input signal.

32.2.4 Output compare operations

The function [TPM_SetupOutputCompare\(\)](#) sets up a TPM channel for output comparison. The user can specify the channel output on a successful comparison and a comparison value.

32.2.5 Quad decode

The function [TPM_SetupQuadDecode\(\)](#) sets up TPM channels 0 and 1 for quad decode, which is available only for certain SoCs. The user can specify the quad decode mode, polarity, and filter properties for each input signal.

32.2.6 Fault operation

The function [TPM_SetupFault\(\)](#) sets up the properties for each fault, which is available only for certain SoCs. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

32.2.7 Status

Provides functions to get and clear the TPM status.

32.2.8 Interrupt

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

32.3 Typical use case

32.3.1 PWM output

Output the PWM signal on 2 TPM 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/tpm

Data Structures

- struct [tpm_chnl_pwm_signal_param_t](#)
Options to configure a TPM channel's PWM signal. [More...](#)
- struct [tpm_config_t](#)
TPM config structure. [More...](#)

Macros

- #define [TPM_MAX_COUNTER_VALUE\(x\)](#) ((1U != (uint8_t)FSL_FEATURE_TPM_HAS_32BIT_COUNTERn(x)) ? 0xFFFFU : 0xFFFFFFFFU)
Help macro to get the max counter value.

Enumerations

- enum [tpm_chnl_t](#) {

kTPM_Chnl_0 = 0U,

kTPM_Chnl_1,

kTPM_Chnl_2,

kTPM_Chnl_3,

kTPM_Chnl_4,

kTPM_Chnl_5,

kTPM_Chnl_6,

kTPM_Chnl_7 }

List of TPM channels.
- enum [tpm_pwm_mode_t](#) {

kTPM_EdgeAlignedPwm = 0U,

kTPM_CenterAlignedPwm }

TPM PWM operation modes.
- enum [tpm_pwm_level_select_t](#) {

kTPM_NoPwmSignal = 0U,

kTPM_LowTrue,

kTPM_HighTrue }

TPM PWM output pulse mode: high-true, low-true or no output.
- enum [tpm_chnl_control_bit_mask_t](#) {

kTPM_ChnlELSnAMask = TPM_CnSC_ELSA_MASK,

kTPM_ChnlELSnBMask = TPM_CnSC_ELSB_MASK,

kTPM_ChnlMSAMask = TPM_CnSC_MSA_MASK,

kTPM_ChnlMSBMask = TPM_CnSC_MSB_MASK }

List of TPM channel modes and level control bit mask.

- enum `tpm_trigger_select_t`

Trigger sources available.
- enum `tpm_output_compare_mode_t` {

kTPM_NoOutputSignal = (1U << TPM_CnSC_MSA_SHIFT),

kTPM_ToggleOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_S-HIFT)),

kTPM_ClearOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_SHIFT)),

kTPM_SetOnMatch = ((1U << TPM_CnSC_MSA_SHIFT) | (3U << TPM_CnSC_ELSA_SHIFT)),

kTPM_HighPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (1U << TPM_CnSC_ELSA_SHIFT)),

kTPM_LowPulseOutput = ((3U << TPM_CnSC_MSA_SHIFT) | (2U << TPM_CnSC_ELSA_S-HIFT)) }

TPM output compare modes.

- enum `tpm_input_capture_edge_t` {

kTPM_RisingEdge = (1U << TPM_CnSC_ELSA_SHIFT),

kTPM_FallingEdge = (2U << TPM_CnSC_ELSA_SHIFT),

kTPM_RiseAndFallEdge = (3U << TPM_CnSC_ELSA_SHIFT) }

TPM input capture edge.

- enum `tpm_clock_source_t` {

kTPM_SystemClock = 1U,

kTPM_ExternalClock }

TPM clock source selection.

- enum `tpm_clock_prescale_t` {

kTPM_Prescale_Divide_1 = 0U,

kTPM_Prescale_Divide_2,

kTPM_Prescale_Divide_4,

kTPM_Prescale_Divide_8,

kTPM_Prescale_Divide_16,

kTPM_Prescale_Divide_32,

kTPM_Prescale_Divide_64,

kTPM_Prescale_Divide_128 }

TPM prescale value selection for the clock source.

- enum `tpm_interrupt_enable_t` {

kTPM_Chnl0InterruptEnable = (1U << 0),

kTPM_Chnl1InterruptEnable = (1U << 1),

kTPM_Chnl2InterruptEnable = (1U << 2),

kTPM_Chnl3InterruptEnable = (1U << 3),

kTPM_Chnl4InterruptEnable = (1U << 4),

kTPM_Chnl5InterruptEnable = (1U << 5),

kTPM_Chnl6InterruptEnable = (1U << 6),

kTPM_Chnl7InterruptEnable = (1U << 7),

kTPM_TimeOverflowInterruptEnable = (1U << 8) }

List of TPM interrupts.

- enum `tpm_status_flags_t` {

kTPM_Chnl0Flag = (1U << 0),

kTPM_Chnl1Flag = (1U << 1),

kTPM_Chnl2Flag = (1U << 2),

kTPM_Chnl3Flag = (1U << 3),

kTPM_Chnl4Flag = (1U << 4),

kTPM_Chnl5Flag = (1U << 5),

kTPM_Chnl6Flag = (1U << 6),

kTPM_Chnl7Flag = (1U << 7),

kTPM_TimeOverflowFlag = (1U << 8) }

List of TPM flags.

Driver version

- #define `FSL TPM_DRIVER_VERSION (MAKE_VERSION(2, 2, 2))`
TPM driver version 2.2.1.

Initialization and deinitialization

- void `TPM_Init` (TPM_Type *base, const `tpm_config_t` *config)
Ungates the TPM clock and configures the peripheral for basic operation.
- void `TPM_Deinit` (TPM_Type *base)
Stops the counter and gates the TPM clock.
- void `TPM_GetDefaultConfig` (`tpm_config_t` *config)
Fill in the TPM config struct with the default settings.
- `tpm_clock_prescale_t` `TPM_CalculateCounterClkDiv` (TPM_Type *base, uint32_t counterPeriod_Hz, uint32_t srcClock_Hz)
Calculates the counter clock prescaler.

Channel mode operations

- status_t `TPM_SetupPwm` (TPM_Type *base, const `tpm_chnl_pwm_signal_param_t` *chnlParams, uint8_t numOfChnls, `tpm_pwm_mode_t` mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
Configures the PWM signal parameters.
- status_t `TPM_UpdatePwmDutycycle` (TPM_Type *base, `tpm_chnl_t` chnlNumber, `tpm_pwm_mode_t` currentPwmMode, uint8_t dutyCyclePercent)
Update the duty cycle of an active PWM signal.
- void `TPM_UpdateChnlEdgeLevelSelect` (TPM_Type *base, `tpm_chnl_t` chnlNumber, uint8_t level)
Update the edge level selection for a channel.
- static uint8_t `TPM_GetChannelContorlBits` (TPM_Type *base, `tpm_chnl_t` chnlNumber)
Get the channel control bits value (mode, edge and level bit fileds).
- static void `TPM_DisableChannel` (TPM_Type *base, `tpm_chnl_t` chnlNumber)
Disable the channel.
- static void `TPM_EnableChannel` (TPM_Type *base, `tpm_chnl_t` chnlNumber, uint8_t control)
Enable the channel according to mode and level configs.
- void `TPM_SetupInputCapture` (TPM_Type *base, `tpm_chnl_t` chnlNumber, `tpm_input_capture_edge_t` captureMode)
Enables capturing an input signal on the channel using the function parameters.

- void [TPM_SetupOutputCompare](#) (TPM_Type *base, [tpm_chnl_t](#) chnlNumber, [tpm_output_compare_mode_t](#) compareMode, uint32_t compareValue)
Configures the TPM to generate timed pulses.

Interrupt Interface

- void [TPM_EnableInterrupts](#) (TPM_Type *base, uint32_t mask)
Enables the selected TPM interrupts.
- void [TPM_DisableInterrupts](#) (TPM_Type *base, uint32_t mask)
Disables the selected TPM interrupts.
- uint32_t [TPM_GetEnabledInterrupts](#) (TPM_Type *base)
Gets the enabled TPM interrupts.

Status Interface

- static uint32_t [TPM_GetChannelValue](#) (TPM_Type *base, [tpm_chnl_t](#) chnlNumber)
Gets the TPM channel value.
- static uint32_t [TPM_GetStatusFlags](#) (TPM_Type *base)
Gets the TPM status flags.
- static void [TPM_ClearStatusFlags](#) (TPM_Type *base, uint32_t mask)
Clears the TPM status flags.

Read and write the timer period

- static void [TPM_SetTimerPeriod](#) (TPM_Type *base, uint32_t ticks)
Sets the timer period in units of ticks.
- static uint32_t [TPM_GetCurrentTimerCount](#) (TPM_Type *base)
Reads the current timer counting value.

Timer Start and Stop

- static void [TPM_StartTimer](#) (TPM_Type *base, [tpm_clock_source_t](#) clockSource)
Starts the TPM counter.
- static void [TPM_StopTimer](#) (TPM_Type *base)
Stops the TPM counter.

32.4 Data Structure Documentation

32.4.1 struct tpm_chnl_pwm_signal_param_t

Data Fields

- [tpm_chnl_t](#) chnlNumber
TPM channel to configure.
- [tpm_pwm_level_select_t](#) level
PWM output active level select.
- uint8_t dutyCyclePercent
PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)...

Field Documentation

(1) `tpm_chnl_t tpm_chnl_pwm_signal_param_t::chnlNumber`

In combined mode (available in some SoC's), this represents the channel pair number

(2) `uint8_t tpm_chnl_pwm_signal_param_t::dutyCyclePercent`

100=always active signal (100% duty cycle)

32.4.2 struct tpm_config_t

This structure holds the configuration settings for the TPM peripheral. To initialize this structure to reasonable defaults, call the `TPM_GetDefaultConfig()` function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Data Fields

- `tpm_clock_prescale_t prescale`
Select TPM clock prescale value.
- `bool useGlobalTimeBase`
true: The TPM channels use an external global time base (the local counter still use for generate overflow interrupt and DMA request); false: All TPM channels use the local counter as their timebase
- `bool syncGlobalTimeBase`
true: The TPM counter is synchronized to the global time base; false: disabled
- `tpm_trigger_select_t triggerSelect`
Input trigger to use for controlling the counter operation.
- `bool enableDoze`
true: TPM counter is paused in doze mode; false: TPM counter continues in doze mode
- `bool enableDebugMode`
true: TPM counter continues in debug mode; false: TPM counter is paused in debug mode
- `bool enableReloadOnTrigger`
true: TPM counter is reloaded on trigger; false: TPM counter not reloaded
- `bool enableStopOnOverflow`
true: TPM counter stops after overflow; false: TPM counter continues running after overflow
- `bool enableStartOnTrigger`
true: TPM counter only starts when a trigger is detected; false: TPM counter starts immediately

32.5 Macro Definition Documentation**32.5.1 #define FSL TPM DRIVER VERSION (MAKE_VERSION(2, 2, 2))****32.6 Enumeration Type Documentation****32.6.1 enum tpm_chnl_t**

Note

Actual number of available channels is SoC dependent

Enumerator

- kTPM_Chnl_0* TPM channel number 0.
- kTPM_Chnl_1* TPM channel number 1.
- kTPM_Chnl_2* TPM channel number 2.
- kTPM_Chnl_3* TPM channel number 3.
- kTPM_Chnl_4* TPM channel number 4.
- kTPM_Chnl_5* TPM channel number 5.
- kTPM_Chnl_6* TPM channel number 6.
- kTPM_Chnl_7* TPM channel number 7.

32.6.2 enum tpm_pwm_mode_t

Enumerator

- kTPM_EdgeAlignedPwm* Edge aligned PWM.
- kTPM_CenterAlignedPwm* Center aligned PWM.

32.6.3 enum tpm_pwm_level_select_t

Note

When the TPM has PWM pause level select feature, the PWM output cannot be turned off by selecting the output level. In this case, the channel must be closed to close the PWM output.

Enumerator

- kTPM_NoPwmSignal* No PWM output on pin.
- kTPM_LowTrue* Low true pulses.
- kTPM_HighTrue* High true pulses.

32.6.4 enum tpm_chnl_control_bit_mask_t

Enumerator

- kTPM_ChnlELSnAMask* Channel ELSA bit mask.
- kTPM_ChnlELSnBMask* Channel ELSB bit mask.
- kTPM_ChnlMSAMask* Channel MSA bit mask.
- kTPM_ChnlMSBMask* Channel MSB bit mask.

32.6.5 enum tpm_trigger_select_t

This is used for both internal & external trigger sources (external trigger sources available in certain SoC's)

Note

The actual trigger sources available is SoC-specific.

32.6.6 enum tpm_output_compare_mode_t

Enumerator

kTPM_NoOutputSignal No channel output when counter reaches CnV.

kTPM_ToggleOnMatch Toggle output.

kTPM_ClearOnMatch Clear output.

kTPM_SetOnMatch Set output.

kTPM_HighPulseOutput Pulse output high.

kTPM_LowPulseOutput Pulse output low.

32.6.7 enum tpm_input_capture_edge_t

Enumerator

kTPM_RisingEdge Capture on rising edge only.

kTPM_FallingEdge Capture on falling edge only.

kTPM_RiseAndFallEdge Capture on rising or falling edge.

32.6.8 enum tpm_clock_source_t

Enumerator

kTPM_SystemClock System clock.

kTPM_ExternalClock External TPM_EXTCLK pin clock.

32.6.9 enum tpm_clock_prescale_t

Enumerator

kTPM_Prescale_Divide_1 Divide by 1.

kTPM_Prescale_Divide_2 Divide by 2.

kTPM_Prescale_Divide_4 Divide by 4.
kTPM_Prescale_Divide_8 Divide by 8.
kTPM_Prescale_Divide_16 Divide by 16.
kTPM_Prescale_Divide_32 Divide by 32.
kTPM_Prescale_Divide_64 Divide by 64.
kTPM_Prescale_Divide_128 Divide by 128.

32.6.10 enum tpm_interrupt_enable_t

Enumerator

kTPM_Chnl0InterruptEnable Channel 0 interrupt.
kTPM_Chnl1InterruptEnable Channel 1 interrupt.
kTPM_Chnl2InterruptEnable Channel 2 interrupt.
kTPM_Chnl3InterruptEnable Channel 3 interrupt.
kTPM_Chnl4InterruptEnable Channel 4 interrupt.
kTPM_Chnl5InterruptEnable Channel 5 interrupt.
kTPM_Chnl6InterruptEnable Channel 6 interrupt.
kTPM_Chnl7InterruptEnable Channel 7 interrupt.
kTPM_TimeOverflowInterruptEnable Time overflow interrupt.

32.6.11 enum tpm_status_flags_t

Enumerator

kTPM_Chnl0Flag Channel 0 flag.
kTPM_Chnl1Flag Channel 1 flag.
kTPM_Chnl2Flag Channel 2 flag.
kTPM_Chnl3Flag Channel 3 flag.
kTPM_Chnl4Flag Channel 4 flag.
kTPM_Chnl5Flag Channel 5 flag.
kTPM_Chnl6Flag Channel 6 flag.
kTPM_Chnl7Flag Channel 7 flag.
kTPM_TimeOverflowFlag Time overflow flag.

32.7 Function Documentation

32.7.1 void TPM_Init (TPM_Type * *base*, const tpm_config_t * *config*)

Note

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

Parameters

<i>base</i>	TPM peripheral base address
<i>config</i>	Pointer to user's TPM config structure.

32.7.2 void TPM_Deinit (TPM_Type * *base*)

Parameters

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

32.7.3 void TPM_GetDefaultConfig (tpm_config_t * *config*)

The default values are:

```
*     config->prescale = kTPM_Prescale_Divide_1;
*     config->useGlobalTimeBase = false;
*     config->syncGlobalTimeBase = true;
*     config->dozeEnable = false;
*     config->dbgMode = false;
*     config->enableReloadOnTrigger = false;
*     config->enableStopOnOverflow = false;
*     config->enableStartOnTrigger = false;
*#if FSL_FEATURE TPM HAS_PAUSE_COUNTER_ON_TRIGGER
*     config->enablePauseOnTrigger = false;
#endif
*     config->triggerSelect = kTPM_Trigger_Select_0;
*#if FSL_FEATURE TPM HAS_EXTERNAL_TRIGGER_SELECTION
*     config->triggerSource = kTPM_TriggerSource_External;
*     config->extTriggerPolarity = kTPM_ExtTrigger_Active_High;
#endif
*#if defined(FSL_FEATURE TPM HAS_POL) && FSL_FEATURE TPM HAS_POL
*     config->chnlPolarity = 0U;
#endif
*
```

Parameters

<i>config</i>	Pointer to user's TPM config structure.
---------------	---

32.7.4 tpm_clock_prescale_t TPM_CalculateCounterClkDiv (TPM_Type * *base*, uint32_t *counterPeriod_Hz*, uint32_t *srcClock_Hz*)

This function calculates the values for SC[PS].

Parameters

<i>base</i>	TPM peripheral base address
<i>counterPeriod-Hz</i>	The desired frequency in Hz which corresponds to the time when the counter reaches the mod value
<i>srcClock_Hz</i>	TPM counter clock in Hz

return Calculated clock prescaler value.

32.7.5 status_t TPM_SetupPwm (**TPM_Type * base**, **const tpm_chnl_pwm_signal_param_t * chnlParams**, **uint8_t numOfChnls**, **tpm_pwm_mode_t mode**, **uint32_t pwmFreq_Hz**, **uint32_t srcClock_Hz**)

User calls this function to configure the PWM signals period, mode, dutycycle and edge. Use this function to configure all the TPM channels that will be used to output a PWM signal

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlParams</i>	Array of PWM channel parameters to configure the channel(s)
<i>numOfChnls</i>	Number of channels to configure, this should be the size of the array passed in
<i>mode</i>	PWM operation mode, options available in enumeration tpm_pwm_mode_t
<i>pwmFreq_Hz</i>	PWM signal frequency in Hz
<i>srcClock_Hz</i>	TPM counter clock in Hz

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

32.7.6 status_t TPM_UpdatePwmDutycycle (**TPM_Type * base**, **tpm_chnl_t chnlNumber**, **tpm_pwm_mode_t currentPwmMode**, **uint8_t dutyCyclePercent**)

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number. In combined mode, this represents the channel pair number
<i>currentPwm-Mode</i>	The current PWM mode set during PWM setup
<i>dutyCycle-Percent</i>	New PWM pulse width, value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

Returns

kStatus_Success if the PWM setup was successful, kStatus_Error on failure

32.7.7 void TPM_UpdateChnlEdgeLevelSelect (TPM_Type * *base*, tpm_chnl_t *chnlNumber*, uint8_t *level*)

Note

When the TPM has PWM pause level select feature (FSL FEATURE TPM HAS PAUSE LEVEL SELECT = 1), the PWM output cannot be turned off by selecting the output level. In this case, must use TPM_DisableChannel API to close the PWM output.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number
<i>level</i>	The level to be set to the ELSnB:ELSnA field; valid values are 00, 01, 10, 11. See the appropriate SoC reference manual for details about this field.

32.7.8 static uint8_t TPM_GetChannelContorlBits (TPM_Type * *base*, tpm_chnl_t *chnlNumber*) [inline], [static]

This function disable the channel by clear all mode and level control bits.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number

Returns

The control bits value. This is the logical OR of members of the enumeration [tpm_chnl_control_bit_mask_t](#).

32.7.9 static void TPM_DisableChannel (**TPM_Type** * *base*, **tpm_chnl_t chnlNumber**) [inline], [static]

This function disable the channel by clear all mode and level control bits.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number

32.7.10 static void TPM_EnableChannel (**TPM_Type** * *base*, **tpm_chnl_t chnlNumber, uint8_t control**) [inline], [static]

This function enable the channel output according to input mode/level config parameters.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number
<i>control</i>	The control bits value. This is the logical OR of members of the enumeration tpm_chnl_control_bit_mask_t .

32.7.11 void TPM_SetupInputCapture (**TPM_Type** * *base*, **tpm_chnl_t chnlNumber, tpm_input_capture_edge_t captureMode**)

When the edge specified in the captureMode argument occurs on the channel, the TPM counter is captured into the CnV register. The user has to read the CnV register separately to get this value.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number
<i>captureMode</i>	Specifies which edge to capture

32.7.12 void TPM_SetupOutputCompare (**TPM_Type** * *base*, **tpm_chnl_t** *chnlNumber*, **tpm_output_compare_mode_t** *compareMode*, **uint32_t** *compareValue*)

When the TPM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number
<i>compareMode</i>	Action to take on the channel output when the compare condition is met
<i>compareValue</i>	Value to be programmed in the CnV register.

32.7.13 void TPM_EnableInterrupts (**TPM_Type** * *base*, **uint32_t** *mask*)

Parameters

<i>base</i>	TPM peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t

32.7.14 void TPM_DisableInterrupts (**TPM_Type** * *base*, **uint32_t** *mask*)

Parameters

<i>base</i>	TPM peripheral base address
<i>mask</i>	The interrupts to disable. This is a logical OR of members of the enumeration tpm_interrupt_enable_t

32.7.15 uint32_t TPM_GetEnabledInterrupts (TPM_Type * *base*)

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [tpm_interrupt_enable_t](#)

32.7.16 static uint32_t TPM_GetChannelValue (TPM_Type * *base*, tpm_chnl_t *chnlNumber*) [inline], [static]

Note

The TPM channel value contain the captured TPM counter value for the input modes or the match value for the output modes.

Parameters

<i>base</i>	TPM peripheral base address
<i>chnlNumber</i>	The channel number

Returns

The channle CnV regisyer value.

32.7.17 static uint32_t TPM_GetStatusFlags (TPM_Type * *base*) [inline], [static]

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration [tpm_status_flags_t](#)

32.7.18 static void TPM_ClearStatusFlags (**TPM_Type** * *base*, **uint32_t** *mask*) [inline], [static]

Parameters

<i>base</i>	TPM peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration tpm_status_flags_t

32.7.19 static void TPM_SetTimerPeriod (**TPM_Type** * *base*, **uint32_t** *ticks*) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note

1. This API allows the user to use the TPM module as a timer. Do not mix usage of this API with TPM's PWM setup API's.
2. Call the utility macros provided in the fsl_common.h to convert usec or msec to ticks.

Parameters

<i>base</i>	TPM peripheral base address
<i>ticks</i>	A timer period in units of ticks, which should be equal or greater than 1.

32.7.20 static **uint32_t** TPM_GetCurrentTimerCount (**TPM_Type** * *base*) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Note

Call the utility macros provided in the fsl_common.h to convert ticks to usec or msec.

Parameters

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

Returns

The current counter value in ticks

32.7.21 static void TPM_StartTimer (TPM_Type * *base*, tpm_clock_source_t *clockSource*) [inline], [static]

Parameters

<i>base</i>	TPM peripheral base address
<i>clockSource</i>	TPM clock source; once clock source is set the counter will start running

32.7.22 static void TPM_StopTimer (TPM_Type * *base*) [inline], [static]

Parameters

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

Chapter 33

TRGMUX: Trigger Mux Driver

33.1 Overview

The MCUXpresso SDK provides driver for the Trigger Mux (TRGMUX) module of MCUXpresso SDK devices.

33.2 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.

33.3 Macro Definition Documentation

33.3.1 #define FSL_TRGMUX_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

33.4 Enumeration Type Documentation

33.4.1 anonymous enum

Enumerator

kStatus(TRGMUX_Locked) Configure failed for register is locked.

33.4.2 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.

33.5 Function Documentation

33.5.1 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>.

33.5.2 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 34

TRNG: True Random Number Generator

The MCUXpresso SDK provides a peripheral driver for the True Random Number Generator (TRNG) module of MCUXpresso SDK devices.

The True Random Number Generator is a hardware accelerator module that generates a 512-bit entropy as needed by an entropy consuming module or by other post processing functions. A typical entropy consumer is a pseudo random number generator (PRNG) which can be implemented to achieve both true randomness and cryptographic strength random numbers using the TRNG output as its entropy seed. The entropy generated by a TRNG is intended for direct use by functions that generate secret keys, per-message secrets, random challenges, and other similar quantities used in cryptographic algorithms.

34.1 TRNG Initialization

1. Define the TRNG user configuration structure. Use `TRNG_InitUserConfigDefault()` function to set it to default TRNG configuration values.
2. Initialize the TRNG module, call the `TRNG_Init()` function, and pass the user configuration structure. This function automatically enables the TRNG module and its clock. After that, the TRNG is enabled and the entropy generation starts working.
3. To disable the TRNG module, call the `TRNG_Deinit()` function.

34.2 Get random data from TRNG

1. `TRNG_GetRandomData()` function gets random data from the TRNG module.

This example code shows how to initialize and get random data from the TRNG driver.

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

Chapter 35

TSTMR: Timestamp Timer Driver

35.1 Overview

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

Functions

- static uint64_t [TSTMR_ReadTimeStamp](#) (TSTMR_Type *base)
Reads the time stamp.
- static void [TSTMR_DelayUs](#) (TSTMR_Type *base, uint32_t delayInUs)
Delays for a specified number of microseconds.

Driver version

- #define [FSL_TSTMR_DRIVER_VERSION](#) ([MAKE_VERSION](#)(2, 0, 0))
Version 2.0.0.

35.2 Function Documentation

35.2.1 static uint64_t TSTMR_ReadTimeStamp (**TSTMR_Type * base**) [[inline](#)], [[static](#)]

This function reads the low and high registers and returns the 56-bit free running counter value. This can be read by software at any time to determine the software ticks.

Parameters

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

Returns

The 56-bit time stamp value.

35.2.2 static void TSTMR_DelayUs (**TSTMR_Type * base**, **uint32_t delayInUs**) [[inline](#)], [[static](#)]

This function repeatedly reads the timestamp register and waits for the user-specified delay value.

Parameters

<i>base</i>	TSTMR peripheral base address.
<i>delayInUs</i>	Delay value in microseconds.

Chapter 36

WDOG32: 32-bit Watchdog Timer

36.1 Overview

The MCUXpresso SDK provides a peripheral driver for the WDOG32 module of MCUXpresso SDK devices.

36.2 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.

36.3 Data Structure Documentation

36.3.1 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.

36.3.2 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.

36.4 Macro Definition Documentation

36.4.1 #define FSL_WDOG32_DRIVER_VERSION (MAKE_VERSION(2, 0, 4))

36.5 Enumeration Type Documentation

36.5.1 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.

36.5.2 enum wdog32_clock_prescaler_t

Enumerator

- kWDOG32_ClockPrescalerDivide1* Divided by 1.
- kWDOG32_ClockPrescalerDivide256* Divided by 256.

36.5.3 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.

36.5.4 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.

36.5.5 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.

36.6 Function Documentation

36.6.1 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](#)

36.6.2 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.

36.6.3 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.
-------------	---------------------------------

36.6.4 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
-------------	--------------------------------

36.6.5 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.
-------------	---------------------------------

36.6.6 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
-------------	--------------------------------

36.6.7 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

36.6.8 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

36.6.9 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.

36.6.10 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

36.6.11 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.

36.6.12 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.

36.6.13 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
-------------	--------------------------------

36.6.14 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 37

XRDC: Extended Resource Domain Controller

37.1 Overview

The MCUXpresso SDK provides a driver for the Extended Resource Domain Controller (XRDC) block of MCUXpresso SDK devices.

37.2 XRDC functions

The XRDC module includes four submodules, as follows:

- XRDC_MGR The Manager submodule coordinates all programming model reads and writes.
- XRDC_MDAC The Master Domain Assignment Controller handles resource assignments and generation of the domain identifiers (domain ID).
- XRDC_MRC The Memory Region Controller implements the access controls for slave memories based on the pre-programmed region descriptor registers.
- XRDC_PAC The Peripheral Access Controller implements the access controls for slave peripherals based on the preprogrammed domain access control registers.

Accordingly, the XRDC driver functions could be grouped as follows:

- XRDC_MGR functions.
- XRDC_MDAC functions.
- XRDC_MRC functions.
- XRDC_PAC functions.

37.3 Typical use case

37.3.1 Set up configurations during system initialization

The domain assignment and access policy can be configured during the system initialization.

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

37.3.2 XRDC error handle

When an access violation occurs, the hard fault is triggered. The function [XRDC_GetAndClearFirstDomainError\(\)](#) is used to get the error information. Although there might be more than one error, this function only gets the first error.

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

37.3.3 Access involve SEMA42

See the SoC reference manual to check which SEMA42 instance is used. For example, for KL28, the memory region defined by the MRC0 uses the SEMA42-0, while the memory region defined by MRC1 uses the SEMA42-1. The peripherals controlled by the PAC0 and PAC2 use the SEMA42-0, while the peripherals controlled by PAC1 use the SEMA42-1.

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

Data Structures

- struct `xrdc_hardware_config_t`
XRDC hardware configuration. [More...](#)
- struct `xrdc_processor_domain_assignment_t`
Domain assignment for the processor bus master. [More...](#)
- struct `xrdc_non_processor_domain_assignment_t`
Domain assignment for the non-processor bus master. [More...](#)
- struct `xrdc_pid_config_t`
XRDC process identifier (PID) configuration. [More...](#)
- struct `xrdc_periph_access_config_t`
XRDC peripheral domain access control configuration. [More...](#)
- struct `xrdc_mem_access_config_t`
XRDC memory region domain access control configuration. [More...](#)
- struct `xrdc_error_t`
XRDC domain error definition. [More...](#)

Enumerations

- enum { `kStatus_XRDC_NoError` = MAKE_STATUS(kStatusGroup_XRDC, 0) }
XRDC status _xrdc_status.
- enum `xrdc_pid_enable_t` {
`kXRDC_PidDisable`,
`kXRDC_PidDisable1`,
`kXRDC_PidExp0`,
`kXRDC_PidExp1` }
XRDC PID enable mode, the register bit XRDC_MDA_Wx[PE], used for domain hit evaluation.
- enum `xrdc_did_sel_t` {
`kXRDC_DidMda`,
`kXRDC_DidInput`,
`kXRDC_DidMdaAndInput`,
`kXRDC_DidReserved` }
XRDC domain ID select method, the register bit XRDC_MDA_Wx[DIDS], used for domain hit evaluation.
- enum `xrdc_secure_attr_t` {
`kXRDC_ForceSecure`,
`kXRDC_ForceNonSecure`,
`kXRDC_MasterSecure`,
`kXRDC_MasterSecure1` }
XRDC secure attribute, the register bit XRDC_MDA_Wx[SA], used for non-processor bus master domain assignment.

- enum `xrdc_privilege_attr_t` {
 `kXRDC_ForceUser`,
 `kXRDC_ForcePrivilege`,
 `kXRDC_MasterPrivilege`,
 `kXRDC_MasterPrivilege1` }
 XRDC privileged attribute, the register bit XRDC_MDA_Wx[PA], used for non-processor bus master domain assignment.
- enum `xrdc_pid_lock_t` {
 `kXRDC_PidLockSecurePrivilegeWritable` = 0U,
 `kXRDC_PidLockSecurePrivilegeWritable1` = 1U,
 `kXRDC_PidLockMasterXOnly` = 2U,
 `kXRDC_PidLockLocked` = 3U }
 XRDC PID LK2 definition XRDC_PIDn[LK2].
- enum `xrdc_access_policy_t`
 XRDC domain access control policy.
- enum `xrdc_access_config_lock_t` {
 `kXRDC_AccessConfigLockWritable` = 0U,
 `kXRDC_AccessConfigLockWritable1` = 1U,
 `kXRDC_AccessConfigLockDomainXOnly` = 2U,
 `kXRDC_AccessConfigLockLocked` = 3U }
 Access configuration lock mode, the register field PDAC and MRGD LK2.
- enum `xrdc_mem_size_t` {

```
kXRDC_MemSizeNone = 0U,  
kXRDC_MemSize32B = 4U,  
kXRDC_MemSize64B = 5U,  
kXRDC_MemSize128B = 6U,  
kXRDC_MemSize256B = 7U,  
kXRDC_MemSize512B = 8U,  
kXRDC_MemSize1K = 9U,  
kXRDC_MemSize2K = 10U,  
kXRDC_MemSize4K = 11U,  
kXRDC_MemSize8K = 12U,  
kXRDC_MemSize16K = 13U,  
kXRDC_MemSize32K = 14U,  
kXRDC_MemSize64K = 15U,  
kXRDC_MemSize128K = 16U,  
kXRDC_MemSize256K = 17U,  
kXRDC_MemSize512K = 18U,  
kXRDC_MemSize1M = 19U,  
kXRDC_MemSize2M = 20U,  
kXRDC_MemSize4M = 21U,  
kXRDC_MemSize8M = 22U,  
kXRDC_MemSize16M = 23U,  
kXRDC_MemSize32M = 24U,  
kXRDC_MemSize64M = 25U,  
kXRDC_MemSize128M = 26U,  
kXRDC_MemSize256M = 27U,  
kXRDC_MemSize512M = 28U,  
kXRDC_MemSize1G = 29U,  
kXRDC_MemSize2G = 30U,  
kXRDC_MemSize4G = 31U }
```

XRDC memory size definition.

- enum `xrdc_controller_t` {

```

kXRDC_MemController0 = 0U,
kXRDC_MemController1 = 1U,
kXRDC_MemController2 = 2U,
kXRDC_MemController3 = 3U,
kXRDC_MemController4 = 4U,
kXRDC_MemController5 = 5U,
kXRDC_MemController6 = 6U,
kXRDC_MemController7 = 7U,
kXRDC_MemController8 = 8U,
kXRDC_MemController9 = 9U,
kXRDC_MemController10 = 10U,
kXRDC_MemController11 = 11U,
kXRDC_MemController12 = 12U,
kXRDC_MemController13 = 13U,
kXRDC_MemController14 = 14U,
kXRDC_MemController15 = 15U,
kXRDC_PeriphController0 = 16U,
kXRDC_PeriphController1 = 17U,
kXRDC_PeriphController2 = 18U,
kXRDC_PeriphController3 = 19U }

```

XRDC controller definition for domain error check.

- enum `xrdc_error_state_t` {


```

kXRDC_ErrorStateNone = 0x00U,
kXRDC_ErrorStateNone1 = 0x01U,
kXRDC_ErrorStateSingle = 0x02U,
kXRDC_ErrorStateMulti = 0x03U }
```

XRDC domain error state definition XRDC_DERR_WI_n[EST].

- enum `xrdc_error_attr_t` {


```

kXRDC_ErrorSecureUserInst = 0x00U,
kXRDC_ErrorSecureUserData = 0x01U,
kXRDC_ErrorSecurePrivilegeInst = 0x02U,
kXRDC_ErrorSecurePrivilegeData = 0x03U,
kXRDC_ErrorNonSecureUserInst = 0x04U,
kXRDC_ErrorNonSecureUserData = 0x05U,
kXRDC_ErrorNonSecurePrivilegeInst = 0x06U,
kXRDC_ErrorNonSecurePrivilegeData = 0x07U }
```

XRDC domain error attribute definition XRDC_DERR_WI_n[EATR].

- enum `xrdc_error_type_t` {


```

kXRDC_ErrorTypeRead = 0x00U,
kXRDC_ErrorTypeWrite = 0x01U }
```

XRDC domain error access type definition XRDC_DERR_WI_n[ERW].

Functions

- void `XRDC_Init` (XRDC_Type *base)
Initializes the XRDC module.

- void [XRDC_Deinit](#) (XRDC_Type *base)
De-initializes the XRDC module.

XRDC manager (XRDC_MGR)

- void [XRDC_GetHardwareConfig](#) (XRDC_Type *base, [xrdc_hardware_config_t](#) *config)
Gets the XRDC hardware configuration.
- static void [XRDC_LockGlobalControl](#) (XRDC_Type *base)
Locks the XRDC global control register XRDC_CR.
- static void [XRDC_SetGlobalValid](#) (XRDC_Type *base, bool valid)
Sets the XRDC global valid.
- static uint8_t [XRDC_GetCurrentMasterDomainId](#) (XRDC_Type *base)
Gets the domain ID of the current bus master.
- status_t [XRDC_GetAndClearFirstDomainError](#) (XRDC_Type *base, [xrdc_error_t](#) *error)
Gets and clears the first domain error of the current domain.
- status_t [XRDC_GetAndClearFirstSpecificDomainError](#) (XRDC_Type *base, [xrdc_error_t](#) *error, uint8_t domainId)
Gets and clears the first domain error of the specific domain.

XRDC Master Domain Assignment Controller (XRDC_MDAC).

- void [XRDC_GetPidDefaultConfig](#) ([xrdc_pid_config_t](#) *config)
Gets the default PID configuration structure.
- void [XRDC_SetPidConfig](#) (XRDC_Type *base, [xrdc_master_t](#) master, const [xrdc_pid_config_t](#) *config)
Configures the PID for a specific bus master.
- static void [XRDC_SetPidLockMode](#) (XRDC_Type *base, [xrdc_master_t](#) master, [xrdc_pid_lock_t](#) lockMode)
Sets the PID configuration register lock mode.
- void [XRDC_GetDefaultNonProcessorDomainAssignment](#) ([xrdc_non_processor_domain_assignment_t](#) *domainAssignment)
Gets the default master domain assignment for non-processor bus master.
- void [XRDC_GetDefaultProcessorDomainAssignment](#) ([xrdc_processor_domain_assignment_t](#) *domainAssignment)
Gets the default master domain assignment for the processor bus master.
- void [XRDC_SetNonProcessorDomainAssignment](#) (XRDC_Type *base, [xrdc_master_t](#) master, uint8_t assignIndex, const [xrdc_non_processor_domain_assignment_t](#) *domainAssignment)
Sets the non-processor bus master domain assignment.
- void [XRDC_SetProcessorDomainAssignment](#) (XRDC_Type *base, [xrdc_master_t](#) master, uint8_t assignIndex, const [xrdc_processor_domain_assignment_t](#) *domainAssignment)
Sets the processor bus master domain assignment.
- static void [XRDC_LockMasterDomainAssignment](#) (XRDC_Type *base, [xrdc_master_t](#) master, uint8_t assignIndex)
Locks the bus master domain assignment register.
- static void [XRDC_SetMasterDomainAssignmentValid](#) (XRDC_Type *base, [xrdc_master_t](#) master, uint8_t assignIndex, bool valid)
Sets the master domain assignment as valid or invalid.

XRDC Memory Region Controller (XRDC_MRC)

- void [XRDC_GetMemAccessDefaultConfig](#) (`xrdc_mem_access_config_t` *config)
Gets the default memory region access policy.
- void [XRDC_SetMemAccessConfig](#) (`XRDC_Type` *base, const `xrdc_mem_access_config_t` *config)
Sets the memory region access policy.
- static void [XRDC_SetMemAccessLockMode](#) (`XRDC_Type` *base, `xrdc_mem_t` mem, `xrdc_access_config_lock_t` lockMode)
Sets the memory region descriptor register lock mode.
- static void [XRDC_SetMemAccessValid](#) (`XRDC_Type` *base, `xrdc_mem_t` mem, bool valid)
Sets the memory region descriptor as valid or invalid.

XRDC Peripheral Access Controller (XRDC_PAC)

- void [XRDC_GetPeriphAccessDefaultConfig](#) (`xrdc_periph_access_config_t` *config)
Gets the default peripheral access configuration.
- void [XRDC_SetPeriphAccessConfig](#) (`XRDC_Type` *base, const `xrdc_periph_access_config_t` *config)
Sets the peripheral access configuration.
- static void [XRDC_SetPeriphAccessLockMode](#) (`XRDC_Type` *base, `xrdc_periph_t` periph, `xrdc_access_config_lock_t` lockMode)
Sets the peripheral access configuration register lock mode.
- static void [XRDC_SetPeriphAccessValid](#) (`XRDC_Type` *base, `xrdc_periph_t` periph, bool valid)
Sets the peripheral access as valid or invalid.

37.4 Data Structure Documentation

37.4.1 struct xrdc_hardware_config_t

Data Fields

- `uint8_t masterNumber`
Number of bus masters.
- `uint8_t domainNumber`
Number of domains.
- `uint8_t pacNumber`
Number of PACs.
- `uint8_t mrcNumber`
Number of MRCs.

Field Documentation

- (1) `uint8_t xrdc_hardware_config_t::masterNumber`
- (2) `uint8_t xrdc_hardware_config_t::domainNumber`
- (3) `uint8_t xrdc_hardware_config_t::pacNumber`
- (4) `uint8_t xrdc_hardware_config_t::mrcNumber`

37.4.2 struct xrdc_processor_domain_assignment_t**Data Fields**

- `uint32_t domainId: 4U`
Domain ID.
- `uint32_t domainIdSelect: 2U`
Domain ID select method, see [xrdc_did_sel_t](#).
- `uint32_t pidEnable: 2U`
PId enable method, see [xrdc_pid_enable_t](#).
- `uint32_t pidMask: 6U`
PId mask.
- `uint32_t __pad0__: 2U`
Reserved.
- `uint32_t pid: 6U`
PId value.
- `uint32_t __pad1__: 2U`
Reserved.
- `uint32_t logicPartId: 4U`
Logical partition ID.
- `uint32_t enableLogicPartId: 1U`
Logical partition ID.
- `uint32_t __pad2__: 1U`
Reserved.
- `uint32_t lock: 1U`
Lock the register.
- `uint32_t __pad3__: 1U`
Reserved.

Field Documentation

- (1) `uint32_t xrdc_processor_domain_assignment_t::domainId`
- (2) `uint32_t xrdc_processor_domain_assignment_t::domainIdSelect`
- (3) `uint32_t xrdc_processor_domain_assignment_t::pidEnable`
- (4) `uint32_t xrdc_processor_domain_assignment_t::pidMask`
- (5) `uint32_t xrdc_processor_domain_assignment_t::__pad0__`
- (6) `uint32_t xrdc_processor_domain_assignment_t::pid`
- (7) `uint32_t xrdc_processor_domain_assignment_t::__pad1__`
- (8) `uint32_t xrdc_processor_domain_assignment_t::logicPartId`
- (9) `uint32_t xrdc_processor_domain_assignment_t::enableLogicPartId`
- (10) `uint32_t xrdc_processor_domain_assignment_t::__pad2__`
- (11) `uint32_t xrdc_processor_domain_assignment_t::lock`
- (12) `uint32_t xrdc_processor_domain_assignment_t::__pad3__`

37.4.3 struct xrdc_non_processor_domain_assignment_t

Data Fields

- `uint32_t domainId: 4U`
Domain ID.
- `uint32_t privilegeAttr: 2U`
Privileged attribute, see [xrdc_privilege_attr_t](#).
- `uint32_t secureAttr: 2U`
Secure attribute, see [xrdc_secure_attr_t](#).
- `uint32_t bypassDomainId: 1U`
Bypass domain ID.
- `uint32_t __pad0__: 15U`
Reserved.
- `uint32_t logicPartId: 4U`
Logical partition ID.
- `uint32_t enableLogicPartId: 1U`
Enable logical partition ID.
- `uint32_t __pad1__: 1U`
Reserved.
- `uint32_t lock: 1U`
Lock the register.
- `uint32_t __pad2__: 1U`
Reserved.

Field Documentation

- (1) `uint32_t xrdc_non_processor_domain_assignment_t::domainId`
- (2) `uint32_t xrdc_non_processor_domain_assignment_t::privilegeAttr`
- (3) `uint32_t xrdc_non_processor_domain_assignment_t::secureAttr`
- (4) `uint32_t xrdc_non_processor_domain_assignment_t::bypassDomainId`
- (5) `uint32_t xrdc_non_processor_domain_assignment_t::__pad0__`
- (6) `uint32_t xrdc_non_processor_domain_assignment_t::logicPartId`
- (7) `uint32_t xrdc_non_processor_domain_assignment_t::enableLogicPartId`
- (8) `uint32_t xrdc_non_processor_domain_assignment_t::__pad1__`
- (9) `uint32_t xrdc_non_processor_domain_assignment_t::lock`
- (10) `uint32_t xrdc_non_processor_domain_assignment_t::__pad2__`

37.4.4 struct xrdc_pid_config_t

Data Fields

- `uint32_t pid`: 6U
PID value, PIDn[PID].
- `uint32_t __pad0__`: 22U
Reserved.
- `uint32_t tsmEnable`: 1U
Enable three-state model.
- `uint32_t lockMode`: 2U
PIDn configuration lock mode, see [xrdc_pid_lock_t](#).
- `uint32_t __pad1__`: 1U
Reserved.

Field Documentation

- (1) `uint32_t xrdc_pid_config_t::pid`
- (2) `uint32_t xrdc_pid_config_t::__pad0__`
- (3) `uint32_t xrdc_pid_config_t::tsmEnable`
- (4) `uint32_t xrdc_pid_config_t::lockMode`
- (5) `uint32_t xrdc_pid_config_t::__pad1__`

37.4.5 struct xrdc_periph_access_config_t**Data Fields**

- `xrdc_periph_t periph`
Peripheral name.
- `xrdc_access_config_lock_t lockMode`
PDACn lock configuration.
- `bool enableSema`
Enable semaphore or not.
- `uint32_t semaNum`
Semaphore number.
- `xrdc_access_policy_t policy [FSL_FEATURE_XRDC_DOMAIN_COUNT]`
Access policy for each domain.

Field Documentation

- (1) `xrdc_periph_t xrdc_periph_access_config_t::periph`
- (2) `xrdc_access_config_lock_t xrdc_periph_access_config_t::lockMode`
- (3) `bool xrdc_periph_access_config_t::enableSema`
- (4) `uint32_t xrdc_periph_access_config_t::semaNum`
- (5) `xrdc_access_policy_t xrdc_periph_access_config_t::policy[FSL_FEATURE_XRDC_DOMAIN_COUNT]`

37.4.6 struct xrdc_mem_access_config_t**Data Fields**

- `xrdc_mem_t mem`
Memory region descriptor name.
- `bool enableSema`
Enable semaphore or not.
- `uint8_t semaNum`

- *Semaphore number.*
- **xrdc_mem_size_t size**
Memory region size.
- **uint8_t subRegionDisableMask**
Sub-region disable mask.
- **xrdc_access_config_lock_t lockMode**
MRGDn lock configuration.
- **xrdc_access_policy_t policy** [FSL_FEATURE_XRDC_DOMAIN_COUNT]
Access policy for each domain.
- **uint32_t baseAddress**
Memory region base/start address.

Field Documentation

- (1) **xrdc_mem_t xrdc_mem_access_config_t::mem**
- (2) **bool xrdc_mem_access_config_t::enableSema**
- (3) **uint8_t xrdc_mem_access_config_t::semaNum**
- (4) **xrdc_mem_size_t xrdc_mem_access_config_t::size**
- (5) **uint8_t xrdc_mem_access_config_t::subRegionDisableMask**
- (6) **xrdc_access_config_lock_t xrdc_mem_access_config_t::lockMode**
- (7) **xrdc_access_policy_t xrdc_mem_access_config_t::policy[FSL_FEATURE_XRDC_DOMAIN_COUNT]**
- (8) **uint32_t xrdc_mem_access_config_t::baseAddress**

37.4.7 struct xrdc_error_t

Data Fields

- **xrdc_controller_t controller**
Which controller captured access violation.
- **uint32_t address**
Access address that generated access violation.
- **xrdc_error_state_t errorState**
Error state.
- **xrdc_error_attr_t errorAttr**
Error attribute.
- **xrdc_error_type_t errorType**
Error type.
- **uint8_t errorPort**
Error port.
- **uint8_t domainId**
Domain ID.

Field Documentation

- (1) `xrdc_controller_t xrdc_error_t::controller`
- (2) `uint32_t xrdc_error_t::address`
- (3) `xrdc_error_state_t xrdc_error_t::errorState`
- (4) `xrdc_error_attr_t xrdc_error_t::errorAttr`
- (5) `xrdc_error_type_t xrdc_error_t::errorType`
- (6) `uint8_t xrdc_error_t::errorPort`
- (7) `uint8_t xrdc_error_t::domainId`

37.5 Enumeration Type Documentation**37.5.1 anonymous enum**

Enumerator

kStatus_XRDC_NoError No error captured.

37.5.2 enum xrdc_pid_enable_t

Enumerator

kXRDC_PidDisable PID is not used in domain hit evalution.

kXRDC_PidDisable1 PID is not used in domain hit evalution.

kXRDC_PidExp0 ((XRDC_MDA_W[PID] & ~XRDC_MDA_W[PIDM]) == (XRDC_PID[PID] & ~XRDC_MDA_W[PIDM])).

kXRDC_PidExp1 ~((XRDC_MDA_W[PID] & ~XRDC_MDA_W[PIDM]) == (XRDC_PID[PID] & ~XRDC_MDA_W[PIDM])).

37.5.3 enum xrdc_did_sel_t

Enumerator

kXRDC_DidMda Use MDAn[3:0] as DID.

kXRDC_DidInput Use the input DID (DID_in) as DID.

kXRDC_DidMdaAndInput Use MDAn[3:2] concatenated with DID_in[1:0] as DID.

kXRDC_DidReserved Reserved.

37.5.4 enum xrdc_secure_attr_t

Enumerator

kXRDC_ForceSecure Force the bus attribute for this master to secure.

kXRDC_ForceNonSecure Force the bus attribute for this master to non-secure.

kXRDC_MasterSecure Use the bus master's secure/nonsecure attribute directly.

kXRDC_MasterSecure1 Use the bus master's secure/nonsecure attribute directly.

37.5.5 enum xrdc_privilege_attr_t

Enumerator

kXRDC_ForceUser Force the bus attribute for this master to user.

kXRDC_ForcePrivilege Force the bus attribute for this master to privileged.

kXRDC_MasterPrivilege Use the bus master's attribute directly.

kXRDC_MasterPrivilege1 Use the bus master's attribute directly.

37.5.6 enum xrdc_pid_lock_t

Enumerator

kXRDC_PidLockSecurePrivilegeWritable Writable by any secure privileged write.

kXRDC_PidLockSecurePrivilegeWritable1 Writable by any secure privileged write.

kXRDC_PidLockMasterXOnly PIDx is only writable by master x.

kXRDC_PidLockLocked Read-only until the next reset.

37.5.7 enum xrdc_access_config_lock_t

Enumerator

kXRDC_AccessConfigLockWritable Entire PDACn/MRGDn can be written.

kXRDC_AccessConfigLockWritable1 Entire PDACn/MRGDn can be written.

kXRDC_AccessConfigLockDomainXOnly Domain x only write the DxACP field.

kXRDC_AccessConfigLockLocked PDACn is read-only until the next reset.

37.5.8 enum xrdc_mem_size_t

Enumerator

kXRDC_MemSizeNone None size.

kXRDC_MemSize32B $2^{(4+1)} = 32$
kXRDC_MemSize64B $2^{(5+1)} = 64$
kXRDC_MemSize128B $2^{(6+1)} = 128$
kXRDC_MemSize256B $2^{(7+1)} = 256$
kXRDC_MemSize512B $2^{(8+1)} = 512$
kXRDC_MemSize1K $2^{(9+1)} = 1\text{kB}$
kXRDC_MemSize2K $2^{(10+1)} = 2\text{kB}$
kXRDC_MemSize4K $2^{(11+1)} = 4\text{kB}$
kXRDC_MemSize8K $2^{(12+1)} = 8\text{kB}$
kXRDC_MemSize16K $2^{(13+1)} = 16\text{kB}$
kXRDC_MemSize32K $2^{(14+1)} = 32\text{kB}$
kXRDC_MemSize64K $2^{(15+1)} = 64\text{kB}$
kXRDC_MemSize128K $2^{(16+1)} = 128\text{kB}$
kXRDC_MemSize256K $2^{(17+1)} = 256\text{kB}$
kXRDC_MemSize512K $2^{(18+1)} = 512\text{kB}$
kXRDC_MemSize1M $2^{(19+1)} = 1\text{MB}$
kXRDC_MemSize2M $2^{(20+1)} = 2\text{MB}$
kXRDC_MemSize4M $2^{(21+1)} = 4\text{MB}$
kXRDC_MemSize8M $2^{(22+1)} = 8\text{MB}$
kXRDC_MemSize16M $2^{(23+1)} = 16\text{MB}$
kXRDC_MemSize32M $2^{(24+1)} = 32\text{MB}$
kXRDC_MemSize64M $2^{(25+1)} = 64\text{MB}$
kXRDC_MemSize128M $2^{(26+1)} = 128\text{MB}$
kXRDC_MemSize256M $2^{(27+1)} = 256\text{MB}$
kXRDC_MemSize512M $2^{(28+1)} = 512\text{MB}$
kXRDC_MemSize1G $2^{(29+1)} = 1\text{GB}$
kXRDC_MemSize2G $2^{(30+1)} = 2\text{GB}$
kXRDC_MemSize4G $2^{(31+1)} = 4\text{GB}$

37.5.9 enum xrdc_controller_t

Enumerator

kXRDC_MemController0 Memory region controller 0.
kXRDC_MemController1 Memory region controller 1.
kXRDC_MemController2 Memory region controller 2.
kXRDC_MemController3 Memory region controller 3.
kXRDC_MemController4 Memory region controller 4.
kXRDC_MemController5 Memory region controller 5.
kXRDC_MemController6 Memory region controller 6.
kXRDC_MemController7 Memory region controller 7.
kXRDC_MemController8 Memory region controller 8.
kXRDC_MemController9 Memory region controller 9.
kXRDC_MemController10 Memory region controller 10.

<i>kXRDC_MemController11</i>	Memory region controller 11.
<i>kXRDC_MemController12</i>	Memory region controller 12.
<i>kXRDC_MemController13</i>	Memory region controller 13.
<i>kXRDC_MemController14</i>	Memory region controller 14.
<i>kXRDC_MemController15</i>	Memory region controller 15.
<i>kXRDC_PeriphController0</i>	Peripheral access controller 0.
<i>kXRDC_PeriphController1</i>	Peripheral access controller 1.
<i>kXRDC_PeriphController2</i>	Peripheral access controller 2.
<i>kXRDC_PeriphController3</i>	Peripheral access controller 3.

37.5.10 enum xrdc_error_state_t

Enumerator

<i>kXRDC_ErrorStateNone</i>	No access violation detected.
<i>kXRDC_ErrorStateNone1</i>	No access violation detected.
<i>kXRDC_ErrorStateSingle</i>	Single access violation detected.
<i>kXRDC_ErrorStateMulti</i>	Multiple access violation detected.

37.5.11 enum xrdc_error_attr_t

Enumerator

<i>kXRDC_ErrorSecureUserInst</i>	Secure user mode, instruction fetch access.
<i>kXRDC_ErrorSecureUserData</i>	Secure user mode, data access.
<i>kXRDC_ErrorSecurePrivilegeInst</i>	Secure privileged mode, instruction fetch access.
<i>kXRDC_ErrorSecurePrivilegeData</i>	Secure privileged mode, data access.
<i>kXRDC_ErrorNonSecureUserInst</i>	NonSecure user mode, instruction fetch access.
<i>kXRDC_ErrorNonSecureUserData</i>	NonSecure user mode, data access.
<i>kXRDC_ErrorNonSecurePrivilegeInst</i>	NonSecure privileged mode, instruction fetch access.
<i>kXRDC_ErrorNonSecurePrivilegeData</i>	NonSecure privileged mode, data access.

37.5.12 enum xrdc_error_type_t

Enumerator

<i>kXRDC_ErrorTypeRead</i>	Error occurs on read reference.
<i>kXRDC_ErrorTypeWrite</i>	Error occurs on write reference.

37.6 Function Documentation

37.6.1 void XRDC_Init (**XRDC_Type** * *base*)

This function enables the XRDC clock.

Parameters

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

37.6.2 void XRDC_Deinit(XRDC_Type * *base*)

This function disables the XRDC clock.

Parameters

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

37.6.3 void XRDC_GetHardwareConfig(XRDC_Type * *base*, xrdc_hardware_config_t * *config*)

This function gets the XRDC hardware configurations, including number of bus masters, number of domains, number of MRCs and number of PACs.

Parameters

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

37.6.4 static void XRDC_LockGlobalControl(XRDC_Type * *base*) [inline], [static]

This function locks the XRDC_CR register. After it is locked, the register is read-only until the next reset.

Parameters

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

37.6.5 static void XRDC_SetGlobalValid(XRDC_Type * *base*, bool *valid*) [inline], [static]

This function sets the XRDC global valid or invalid. When the XRDC is global invalid, all accesses from all bus masters to all slaves are allowed.

Parameters

<i>base</i>	XRDC peripheral base address.
<i>valid</i>	True to valid XRDC.

37.6.6 static uint8_t XRDC_GetCurrentMasterDomainId (**XRDC_Type * base**) [inline], [static]

This function returns the domain ID of the current bus master.

Parameters

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

Returns

Domain ID of current bus master.

37.6.7 status_t XRDC_GetAndClearFirstDomainError (**XRDC_Type * base**, **xrdc_error_t * error**)

This function gets the first access violation information for the current domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

<i>base</i>	XRDC peripheral base address.
<i>error</i>	Pointer to the error information.

Returns

If the access violation is captured, this function returns the kStatus_Success. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the kStatus_XRDC_NoError.

37.6.8 status_t XRDC_GetAndClearFirstSpecificDomainError (**XRDC_Type * base**, **xrdc_error_t * error**, **uint8_t domainId**)

This function gets the first access violation information for the specific domain and clears the pending flag. There might be multiple access violations pending for the current domain. This function only processes the first error.

Parameters

<i>base</i>	XRDC peripheral base address.
<i>error</i>	Pointer to the error information.
<i>domainId</i>	The error of which domain to get and clear.

Returns

If the access violation is captured, this function returns the kStatus_Success. The error information can be obtained from the parameter error. If no access violation is captured, this function returns the kStatus_XRDC_NoError.

37.6.9 void XRDC_GetPidDefaultConfig (*xrdc_pid_config_t* * *config*)

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

```
* config->pid      = 0U;
* config->tsmEnable = 0U;
* config->sp4smEnable = 0U;
* config->lockMode  = kXRDC_PidLockSecurePrivilegeWritetable;
*
```

Parameters

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

37.6.10 void XRDC_SetPidConfig (*XRDC_Type* * *base*, *xrdc_master_t* *master*, *const xrdc_pid_config_t* * *config*)

This function configures the PID for a specific bus master. Do not use this function for non-processor bus masters.

Parameters

<i>base</i>	XRDC peripheral base address.
<i>master</i>	Which bus master to configure.

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

37.6.11 static void XRDC_SetPidLockMode (*XRDC_Type* * *base*, *xrdc_master_t master*, *xrdc_pid_lock_t lockMode*) [inline], [static]

This function sets the PID configuration register lock XRDC_PIDn[LK2].

Parameters

<i>base</i>	XRDC peripheral base address.
<i>master</i>	Which master's PID to lock.
<i>lockMode</i>	Lock mode to set.

37.6.12 void XRDC_GetDefaultNonProcessorDomainAssignment (*xrdc_non_processor_domain_assignment_t* * *domainAssignment*)

This function gets the default master domain assignment for non-processor bus master. It should only be used for the non-processor bus masters, such as DMA. This function sets the assignment as follows:

```
* assignment->domainId      = 0U;
* assignment->privilegeAttr = kXRDC_ForceUser;
* assignment->privilegeAttr = kXRDC_ForceSecure;
* assignment->bypassDomainId = 0U;
* assignment->blogicPartId  = 0U;
* assignment->benableLogicPartId = 0U;
* assignment->lock          = 0U;
*
```

Parameters

<i>domainAssignment</i>	Pointer to the assignment structure.
-------------------------	--------------------------------------

37.6.13 void XRDC_GetDefaultProcessorDomainAssignment (*xrdc_processor_domain_assignment_t* * *domainAssignment*)

This function gets the default master domain assignment for the processor bus master. It should only be used for the processor bus masters, such as CORE0. This function sets the assignment as follows:

```
* assignment->domainId      = 0U;
* assignment->domainIdSelect = kXRDC_DidMda;
```

```

* assignment->dpidEnable      = kXRDC_PidDisable;
* assignment->pidMask        = OU;
* assignment->pid            = OU;
* assignment->logicPartId   = OU;
* assignment->enableLogicPartId = OU;
* assignment->lock           = OU;
*

```

Parameters

<i>domainAssignment</i>	Pointer to the assignment structure.
-------------------------	--------------------------------------

37.6.14 void XRDC_SetNonProcessorDomainAssignment (XRDC_Type * *base*, xrdc_master_t *master*, uint8_t *assignIndex*, const xrdc_non_processor_domain_assignment_t * *domainAssignment*)

This function sets the non-processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter *assignIndex* specifies which assignment register to set.

Example: Set domain assignment for DMA0.

```

* xrdc_non_processor_domain_assignment_t nonProcessorAssignment;
*
* XRDC_GetDefaultNonProcessorDomainAssignment (&
*     nonProcessorAssignment);
* nonProcessorAssignment.domainId = 1;
* nonProcessorAssignment.xxxx     = xxxx;
*
* XRDC_SetMasterDomainAssignment (XRDC, kXrdcMasterDma0, OU, &nonProcessorAssignment);
*

```

Parameters

<i>base</i>	XRDC peripheral base address.
<i>master</i>	Which master to configure.
<i>assignIndex</i>	Which assignment register to set.
<i>domainAssignment</i>	Pointer to the assignment structure.

37.6.15 void XRDC_SetProcessorDomainAssignment (XRDC_Type * *base*, xrdc_master_t *master*, uint8_t *assignIndex*, const xrdc_processor_domain_assignment_t * *domainAssignment*)

This function sets the processor master domain assignment as valid. One bus master might have multiple domain assignment registers. The parameter *assignIndex* specifies which assignment register to set.

Example: Set domain assignment for core 0. In this example, there are 3 assignment registers for core 0.

```
* xrdc_processor_domain_assignment_t processorAssignment;
*
* XRDC_GetDefaultProcessorDomainAssignment (&processorAssignment);
*
* processorAssignment.domainId = 1;
* processorAssignment.xxx      = xxx;
* XRDC_SetMasterDomainAssignment (XRDC, kXrdcMasterCpu0, 0U, &processorAssignment);
*
* processorAssignment.domainId = 2;
* processorAssignment.xxx      = xxx;
* XRDC_SetMasterDomainAssignment (XRDC, kXrdcMasterCpu0, 1U, &processorAssignment);
*
* processorAssignment.domainId = 0;
* processorAssignment.xxx      = xxx;
* XRDC_SetMasterDomainAssignment (XRDC, kXrdcMasterCpu0, 2U, &processorAssignment);
*
```

Parameters

<i>base</i>	XRDC peripheral base address.
<i>master</i>	Which master to configure.
<i>assignIndex</i>	Which assignment register to set.
<i>domainAssignment</i>	Pointer to the assignment structure.

37.6.16 static void XRDC_LockMasterDomainAssignment (XRDC_Type * *base*, xrdc_master_t *master*, uint8_t *assignIndex*) [inline], [static]

This function locks the master domain assignment. One bus master might have multiple domain assignment registers. The parameter *assignIndex* specifies which assignment register to lock. After it is locked, the register can't be changed until next reset.

Parameters

<i>base</i>	XRDC peripheral base address.
<i>master</i>	Which master to configure.
<i>assignIndex</i>	Which assignment register to lock.

37.6.17 static void XRDC_SetMasterDomainAssignmentValid (XRDC_Type * *base*, xrdc_master_t *master*, uint8_t *assignIndex*, bool *valid*) [inline], [static]

This function sets the master domain assignment as valid or invalid. One bus master might have multiple domain assignment registers. The parameter *assignIndex* specifies which assignment register to

configure.

Parameters

<i>base</i>	XRDC peripheral base address.
<i>master</i>	Which master to configure.
<i>assignIndex</i>	Index for the domain assignment register.
<i>valid</i>	True to set valid, false to set invalid.

37.6.18 void XRDC_GetMemAccessDefaultConfig (*xrdc_mem_access_config_t* * *config*)

This function gets the default memory region access policy. It sets the policy as follows:

```
* config->enableSema      = false;
* config->semaNum        = 0U;
* config->subRegionDisableMask = 0U;
* config->size           = kXrdcMemSizeNone;
* config->lockMode       = kXRDC_AccessConfigLockWritable;
* config->baseAddress    = 0U;
* config->policy[0]       = kXRDC_AccessPolicyNone;
* config->policy[1]       = kXRDC_AccessPolicyNone;
* ...
* config->policy[15]      = kXRDC_AccessPolicyNone;
*
```

Parameters

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

37.6.19 void XRDC_SetMemAccessConfig (*XRDC_Type* * *base*, *const xrdc_mem_access_config_t* * *config*)

This function sets the memory region access configuration as valid. There are two methods to use it:

Example 1: Set one configuration run time. Set memory region 0x20000000 ~ 0x20000400 accessible by domain 0, use MRC0_1.

```
* xrdc_mem_access_config_t config =
* {
*     .mem          = kXRDC_MemMrc0_1,
*     .baseAddress  = 0x20000000U,
*     .size         = kXRDC_MemSize1K,
*     .policy[0]    = kXRDC_AccessPolicyAll
* };
* XRDC_SetMemAccessConfig(XRDC, &config);
*
```

Example 2: Set multiple configurations during startup. Set memory region 0x20000000 ~ 0x20000400 accessible by domain 0, use MRC0_1. Set memory region 0x1FFF0000 ~ 0x1FFF0800 accessible by domain 0, use MRC0_2.

```

* xrdc_mem_access_config_t configs[] =
* {
*     {
*         .mem      = kXRDC_MemMrc0_1,
*         .baseAddress = 0x20000000U,
*         .size      = kXRDC_MemSize1K,
*         .policy[0]  = kXRDC_AccessPolicyAll
*     },
*     {
*         .mem      = kXRDC_MemMrc0_2,
*         .baseAddress = 0x1FFF0000U,
*         .size      = kXRDC_MemSize2K,
*         .policy[0]  = kXRDC_AccessPolicyAll
*     }
* };
*
* for (i=0U; i<((sizeof(configs)/sizeof(configs[0]))); i++)
* {
*     XRDC_SetMemAccessConfig(XRDC, &configs[i]);
* }
*

```

Parameters

<i>base</i>	XRDC peripheral base address.
<i>config</i>	Pointer to the access policy configuration structure.

37.6.20 static void XRDC_SetMemAccessLockMode (XRDC_Type * *base*, xrdc_mem_t *mem*, xrdc_access_config_lock_t *lockMode*) [inline], [static]

Parameters

<i>base</i>	XRDC peripheral base address.
<i>mem</i>	Which memory region descriptor to lock.
<i>lockMode</i>	The lock mode to set.

37.6.21 static void XRDC_SetMemAccessValid (XRDC_Type * *base*, xrdc_mem_t *mem*, bool *valid*) [inline], [static]

This function sets the memory region access configuration dynamically. For example:

```

* xrdc_mem_access_config_t config =
* {
*     .mem      = kXRDC_MemMrc0_1,
*     .baseAddress = 0x20000000U,
*     .size      = kXRDC_MemSize1K,
*     .policy[0]  = kXRDC_AccessPolicyAll
* };

```

```
* XRDC_SetMemAccessConfig(XRDC, &config);
*
* XRDC_SetMemAccessValid(kXRDC_MemMrc0_1, false);
*
* XRDC_SetMemAccessValid(kXRDC_MemMrc0_1, true);
*
```

Parameters

<i>base</i>	XRDC peripheral base address.
<i>mem</i>	Which memory region descriptor to set.
<i>valid</i>	True to set valid, false to set invalid.

37.6.22 void XRDC_GetPeriphAccessDefaultConfig (*xrdc_periph_access_config_t * config*)

The default configuration is set as follows:

```
* config->enableSema      = false;
* config->semaNum        = 0U;
* config->lockMode       = kXRDC_AccessConfigLockWritable;
* config->policy[0]       = kXRDC_AccessPolicyNone;
* config->policy[1]       = kXRDC_AccessPolicyNone;
* ...
* config->policy[15]      = kXRDC_AccessPolicyNone;
*
```

Parameters

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

37.6.23 void XRDC_SetPeriphAccessConfig (*XRDC_Type * base, const xrdc_periph_access_config_t * config*)

This function sets the peripheral access configuration as valid. Two methods to use it: Method 1: Set for one peripheral, which is used for runtime settings. Example: set LPTMR0 accessible by domain 0

```
* xrdc_periph_access_config_t config;
*
* config.periph    = kXRDC_PeriphLptmr0;
* config.policy[0] = kXRDC_AccessPolicyAll;
* XRDC_SetPeriphAccessConfig(XRDC, &config);
*
```

Method 2: Set for multiple peripherals, which is used for initialization settings.

```

* xrdc_periph_access_config_t configs[] =
* {
*     {
*         .periph      = kXRDC_PeriphLptmr0,
*         .policy[0]   = kXRDC_AccessPolicyAll,
*         .policy[1]   = kXRDC_AccessPolicyAll
*     },
*     {
*         .periph      = kXRDC_PeriphLpuart0,
*         .policy[0]   = kXRDC_AccessPolicyAll,
*         .policy[1]   = kXRDC_AccessPolicyAll
*     }
* };
*
* for (i=0U; i<(sizeof(configs)/sizeof(configs[0])), i++)
* {
*     XRDC_SetPeriphAccessConfig(XRDC, &config[i]);
* }
*

```

Parameters

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

37.6.24 static void XRDC_SetPeriphAccessLockMode (XRDC_Type * *base*, xrdc_periph_t *periph*, xrdc_access_config_lock_t *lockMode*) [inline], [static]

Parameters

<i>base</i>	XRDC peripheral base address.
<i>periph</i>	Which peripheral access configuration register to lock.
<i>lockMode</i>	The lock mode to set.

37.6.25 static void XRDC_SetPeriphAccessValid (XRDC_Type * *base*, xrdc_periph_t *periph*, bool *valid*) [inline], [static]

This function sets the peripheral access configuration dynamically. For example:

```

* xrdc_periph_access_config_t config =
* {
*     .periph      = kXRDC_PeriphLptmr0;
*     .policy[0]   = kXRDC_AccessPolicyAll;
* };
* XRDC_SetPeriphAccessConfig(XRDC, &config);
*
* XRDC_SetPeriphAccessValid(kXrdcPeriLptmr0, false);
*

```

```
* XRDC_SetPeriphAccessValid(kXrdcPeriLptmr0, true);  
*
```

Parameters

<i>base</i>	XRDC peripheral base address.
<i>periph</i>	Which peripheral access configuration to set.
<i>valid</i>	True to set valid, false to set invalid.

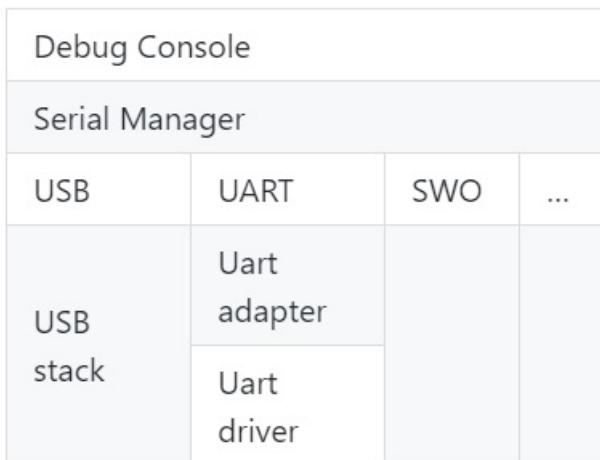
Chapter 38

Debug Console

38.1 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

38.2 Function groups

38.2.1 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);
```

38.2.2 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

specifier	Description
-----------	-------------

- 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 */

```

38.2.3 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

SDK_DEBUGCONSOLE	SDK_DEBUGCONSOLE_UART	PRINTF	printf
-------------------------	------------------------------	---------------	---------------

* the **low level peripheral** could be USB CDC, UART, or SWO, and so on.

38.3 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 GNUMC Base, add file '**fsl_sbrk.c**' in path: ..\{package}\devices\{subset}\utilities\fsl_sbrk.c to your project.

Modules

- **SWO**
- **Semihosting**
- **debug console configuration**

The configuration is used for debug console only.

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.

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.
- status_t [DbgConsole_TryGetchar](#) (char *ch)
Debug console try to get char This function provides a API which will not block current task, if character is available return it, otherwise return fail.

38.4 Macro Definition Documentation

38.4.1 #define DEBUGCONSOLE_REDIRECT_TO_TOOLCHAIN 0U

Select toolchain printf and scanf.

38.4.2 #define DEBUGCONSOLE_REDIRECT_TO_SDK 1U

38.4.3 #define DEBUGCONSOLE_DISABLE 2U

38.4.4 #define SDK_DEBUGCONSOLE DEBUGCONSOLE_REDIRECT_TO_SDK

The macro only support to be redefined in project setting.

38.4.5 #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.

38.5 Function Documentation

38.5.1 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.

Returns

<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
------------------------	------------------------

38.5.2 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.

38.5.3 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.

38.5.4 status_t DbgConsole_ExitLowpower (void)

This function is used to restore from low power consumption.

Returns

Indicates whether de-initialization was successful or not.

38.5.5 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.

38.5.6 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.

38.5.7 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.

38.5.8 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.

38.5.9 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.

38.5.10 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.

38.5.11 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.

38.5.12 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.

38.5.13 **status_t DbgConsole_TryGetchar (char * *ch*)**

Parameters

<i>ch</i>	the address of char to receive
-----------	--------------------------------

Returns

Indicates get char was successful or not.

38.6 debug console configuration

The configuration is used for debug console only.

38.6.1 Overview

Please note, it is not sued for debug console lite.

Macros

- `#define DEBUG_CONSOLE_TRANSMIT_BUFFER_LEN (512U)`
If Non-blocking mode is needed, please define it at project setting, otherwise blocking mode is the default transfer mode.
- `#define DEBUG_CONSOLE_RECEIVE_BUFFER_LEN (1024U)`
define the receive buffer length which is used to store the user input, buffer is enabled automatically when non-blocking transfer is using, This value will affect the RAM's ultilization, should be set per paltform's capability and software requirement.
- `#define DEBUG_CONSOLE_TX_RELIABLE_ENABLE (1U)`
Whether enable the reliable TX function If the macro is zero, the reliable TX function of the debug console is disabled.
- `#define DEBUG_CONSOLE_RX_ENABLE (1U)`
Whether enable the RX function If the macro is zero, the receive function of the debug console is disabled.
- `#define DEBUG_CONSOLE_PRINTF_MAX_LOG_LEN (128U)`
define the MAX log length debug console support , that is when you call printf("log", x);, the log length can not bigger than this value.
- `#define DEBUG_CONSOLE_SCANF_MAX_LOG_LEN (20U)`
define the buffer support buffer scanf log length, that is when you call scanf("log", &x);, the log length can not bigger than this value.
- `#define DEBUG_CONSOLE_SYNCHRONIZATION_BM 0`
Debug console synchronization User should not change these macro for synchronization mode, but add the corresponding synchronization mechanism per different software environment.
- `#define DEBUG_CONSOLE_SYNCHRONIZATION_FREERTOS 1`
synchronization for freertos software
- `#define DEBUG_CONSOLE_SYNCHRONIZATION_MODE DEBUG_CONSOLE_SYNCHRONIZATION_BM`
RTOS synchronization mechanism disable If not defined, default is enable, to avoid multitask log print mess.
- `#define DEBUG_CONSOLE_ENABLE_ECHO_FUNCTION 0`
echo function support If you want to use the echo function,please define DEBUG_CONSOLE_ENABLE_ECHO at your project setting.
- `#define BOARD_USE_VIRTUALCOM 0U`
Definition to select virtual com(USB CDC) as the debug console.

38.6.2 Macro Definition Documentation

38.6.2.1 #define DEBUG_CONSOLE_TRANSMIT_BUFFER_LEN (512U)

Warning: If you want to use non-blocking transfer, please make sure the corresponding IO interrupt is enable, otherwise there is no output. And non-blocking is combine with buffer, no matter bare-metal or rtos. Below shows how to configure in your project if you want to use non-blocking mode. For IAR, right click project and select "Options", define it in "C/C++ Compiler->Preprocessor->Defined symbols". For KEIL, click "Options for Target...", define it in "C/C++->Preprocessor Symbols->Define". For ARM-GCC, open CmakeLists.txt and add the following lines, "SET(CMAKE_C_FLAGS_DEBUG "\${CMAKE_C_FLAGS_DEBUG} -DDEBUG_CONSOLE_TRANSFER_NON_BLOCKING")" for debug target. "SET(CMAKE_C_FLAGS_RELEASE "\${CMAKE_C_FLAGS_RELEASE} -DDEBUG_CONSOLE_TRANSFER_NON_BLOCKING")" for release target. For MCUXpresso, right click project and select "Properties", define it in "C/C++ Build->Settings->MCU C Complier->Preprocessor".

define the transmit buffer length which is used to store the multi task log, buffer is enabled automatically when non-blocking transfer is using, This value will affect the RAM's utilization, should be set per platform's capability and software requirement. If it is configured too small, log maybe missed , because the log will not be buffered if the buffer is full, and the print will return immediately with -1. And this value should be multiple of 4 to meet memory alignment.

38.6.2.2 #define DEBUG_CONSOLE_RECEIVE_BUFFER_LEN (1024U)

If it is configured too small, log maybe missed, because buffer will be overwritten if buffer is too small. And this value should be multiple of 4 to meet memory alignment.

38.6.2.3 #define DEBUG_CONSOLE_TX_RELIABLE_ENABLE (1U)

When the macro is zero, the string of PRINTF will be thrown away after the transmit buffer is full.

38.6.2.4 #define DEBUG_CONSOLE_PRINTF_MAX_LOG_LEN (128U)

This macro decide the local log buffer length, the buffer locate at stack, the stack maybe overflow if the buffer is too big and current task stack size not big enough.

38.6.2.5 #define DEBUG_CONSOLE_SCANF_MAX_LOG_LEN (20U)

As same as the DEBUG_CONSOLE_BUFFER_PRINTF_MAX_LOG_LEN.

38.6.2.6 #define DEBUG_CONSOLE_SYNCHRONIZATION_BM 0

Such as, if another RTOS is used, add: #define DEBUG_CONSOLE_SYNCHRONIZATION_XXXX 3 in this configuration file and implement the synchronization in fsl.log.c.

synchronization for baremetal software

38.6.2.7 #define DEBUG_CONSOLE_SYNCHRONIZATION_MODE DEBUG_CONSOLE_SYNCHRONIZATION_BM

If other RTOS is used, you can implement the RTOS's specific synchronization mechanism in fsl.log.c If synchronization is disabled, log maybe messed on terminal.

38.6.2.8 #define BOARD_USE_VIRTUALCOM 0U

38.7 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.

38.7.1 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.

38.7.2 Guide Semihosting for Keil µVision

NOTE: Semihosting is not support by MDK-ARM, use the retargeting functionality of MDK-ARM instead.

38.7.3 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.

38.7.4 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.

38.8 SWO

Serial wire output is a mechanism for ARM targets to output signal from core through a single pin. Some IDEs also support SWO, such IAR and KEIL, both input and output are supported, see below for details.

38.8.1 Guide SWO for SDK

NOTE: After the setting both "printf" and "PRINTF" are available for debugging, JlinkSWOViewer can be used to capture the output log.

Step 1: Setting up the environment

1. Define SERIAL_PORT_TYPE_SWO in your project settings.
2. Prepare code, the port and baudrate can be decided by application, clkSrcFreq should be mcu core clock frequency:

```
DbgConsole_Init(instance, baudRate, kSerialPort_Swo, clkSrcFreq);
```

3. Use PRINTF or printf to print some thing in application.

Step 2: Building the project

Step 3: Download and run project

38.8.1.1 Guide SWO for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging.

Step 1: Setting up the environment

1. Choose 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 SWO.
4. To configure the hardware's generation of trace data, click the SWO Configuration button available in the SWO Configuration dialog box. The value of the CPU clock option must reflect the frequency of the CPU clock speed at which the application executes. Note also that the settings you make are preserved between debug sessions. To decrease the amount of transmissions on the communication channel, you can disable the Timestamp option. Alternatively, set a lower rate for PC Sampling or use a higher SWO clock frequency.
5. Open the SWO Trace window from J-LINK, and click the Activate button to enable trace data collection.
6. There are three cases for this SDK_DEBUGCONSOLE_UART whether or not defined. a: if use uppercase PRINTF to output log, The SDK_DEBUGCONSOLE_UART defined or not defined will not effect debug function. b: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to zero, then debug function ok. c: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to one, then debug function ok.

NOTE: Case a or c only apply at example which enable swo function, the SDK_DEBUGCONSOLE_UART definition in fsl_debug_console.h. For case a and c, Do and not do the above third step will be not affect function.

1. Start the project by choosing Project>Download and Debug.

Step 2: Building the project

Step 3: Starting swo

1. Download and debug application.
2. Choose View -> Terminal I/O to display the output from the I/O operations.
3. Run application.

38.8.2 Guide SWO for Keil µVision

NOTE: After the setting both "printf" and "scanf" are available for debugging.

Step 1: Setting up the environment

1. There are three cases for this SDK_DEBUGCONSOLE_UART whether or not defined. a: if use uppercase PRINTF to output log, the SDK_DEBUGCONSOLE_UART definition does not affect the functionality and skip the second step directly. b: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to zero, then start the second step. c: if use lowercase printf to output log and defined SDK_DEBUGCONSOLE_UART to one, then skip the second step directly.

NOTE: Case a or c only apply at example which enable swo function, the SDK_DEBUGCONSOLE_UART definition in fsl_debug_console.h.

1. In menu bar, click Management Run-Time Environment icon, select Compiler, unfold I/O, enable STDERR/STDIN/STDOUT and set the variant to ITM.
2. Open Project>Options for target or using Alt+F7 or click.
3. Select “Debug” tab, select “J-Link/J-Trace Cortex” and click “Setting button”.
4. Select “Debug” tab and choose Port:SW, then select “Trace” tab, choose “Enable” and click OK, please make sure the Core clock is set correctly, enable autodetect max SWO clk, enable ITM Stimulus Ports 0.

Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

Step 4: Run the project

1. Choose “Debug” on menu bar or Ctrl F5.
2. In menu bar, choose “Serial Window” and click to “Debug (printf) Viewer”.
3. Run line by line to see result in Console Window.

38.8.3 Guide SWO for MCUXpresso IDE

NOTE: MCUX support SWO for LPC-Link2 debug probe only.

38.8.4 Guide SWO for ARMGCC

NOTE: ARMGCC has no library support SWO.

Chapter 39

Notification Framework

39.1 Overview

This section describes the programming interface of the Notifier driver.

39.2 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.

39.3 Data Structure Documentation

39.3.1 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**

39.3.2 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**

39.3.3 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`

39.4 Typedef Documentation

39.4.1 `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.

39.4.2 `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.

39.4.3 **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.

39.5 Enumeration Type Documentation

39.5.1 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.

39.5.2 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.

39.5.3 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.

39.5.4 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.

39.6 Function Documentation

39.6.1 `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.

39.6.2 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.

39.6.3 `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 40

Shell

40.1 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.

40.2 Function groups

40.2.1 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>");
```

40.2.2 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.

40.2.3 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),

`kStatus_SHELL_RetUsage` = MAKE_STATUS(kStatusGroup_SHELL, 4) }

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.

40.3 Data Structure Documentation

40.3.1 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`**

40.4 Macro Definition Documentation

40.4.1 `#define SHELL_NON_BLOCKING_MODE SERIAL_MANAGER_NON_BLOCKING_MODE`

40.4.2 `#define SHELL_AUTO_COMPLETE (1U)`

40.4.3 `#define SHELL_BUFFER_SIZE (64U)`

40.4.4 `#define SHELL_MAX_ARGS (8U)`

40.4.5 `#define SHELL_HISTORY_COUNT (3U)`

40.4.6 `#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

40.4.7 #define SHELL_USE_COMMON_TASK (0U)**40.4.8 #define SHELL_TASK_PRIORITY (2U)****40.4.9 #define SHELL_TASK_STACK_SIZE (1000U)****40.4.10 #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.
-------------	--------------------------------------

**40.4.11 #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.

40.4.12 #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".
----------------	--

40.5 Typedef Documentation

40.5.1 **typedef shell_status_t(* cmd_function_t)(shell_handle_t shellHandle, int32_t argc, char **argv)**

40.6 Enumeration Type Documentation

40.6.1 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.

kStatus_SHELL_RetUsage RetUsage for print cmd usage.

40.7 Function Documentation

40.7.1 **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_Open-WriteHandleFailed</i>	Open the write handle failed.
<i>kStatus_SHELL_Open-ReadHandleFailed</i>	Open the read handle failed.

40.7.2 **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.

40.7.3 **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.
------------------------------	--------------------------------------

40.7.4 **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.
----------------------------	--------------------

40.7.5 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.

40.7.6 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.

40.7.7 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.

40.7.8 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.

40.7.9 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.

40.7.10 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.
--------------------	----------------------------------

40.7.11 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 41

CODEC Driver

41.1 Overview

The MCUXpresso SDK provides a codec abstraction driver interface to access codec register.

Modules

- [CODEC Common Driver](#)
- [CODEC I2C Driver](#)
- [DA7212 Driver](#)
- [SGTL5000 Driver](#)
- [WM8960 Driver](#)

41.2 CODEC Common Driver

41.2.1 Overview

The codec common driver provides a codec control abstraction interface.

Modules

- CODEC Adapter
- DA7212 Adapter
- SGTL5000 Adapter
- WM8960 Adapter

Data Structures

- struct `codec_config_t`
Initialize structure of the codec. [More...](#)
- struct `codec_capability_t`
codec capability [More...](#)
- struct `codec_handle_t`
Codec handle definition. [More...](#)

Macros

- #define `CODEC_VOLUME_MAX_VALUE` (100U)
codec maximum volume range

Enumerations

- enum {

 `kStatus_CODEC_NotSupport` = MAKE_STATUS(kStatusGroup_CODEC, 0U),

 `kStatus_CODEC_DeviceNotRegistered` = MAKE_STATUS(kStatusGroup_CODEC, 1U),

 `kStatus_CODEC_I2CBusInitialFailed`,

 `kStatus_CODEC_I2CCommandTransferFailed` }

CODEC status.
- enum `codec_audio_protocol_t` {

 `kCODEC_BusI2S` = 0U,

 `kCODEC_BusLeftJustified` = 1U,

 `kCODEC_BusRightJustified` = 2U,

 `kCODEC_BusPCMA` = 3U,

 `kCODEC_BusPCMB` = 4U,

 `kCODEC_BusTDM` = 5U }

AUDIO format definition.

- enum {

kCODEC_AudioSampleRate8KHz = 8000U,

kCODEC_AudioSampleRate11025Hz = 11025U,

kCODEC_AudioSampleRate12KHz = 12000U,

kCODEC_AudioSampleRate16KHz = 16000U,

kCODEC_AudioSampleRate22050Hz = 22050U,

kCODEC_AudioSampleRate24KHz = 24000U,

kCODEC_AudioSampleRate32KHz = 32000U,

kCODEC_AudioSampleRate44100Hz = 44100U,

kCODEC_AudioSampleRate48KHz = 48000U,

kCODEC_AudioSampleRate96KHz = 96000U,

kCODEC_AudioSampleRate192KHz = 192000U,

kCODEC_AudioSampleRate384KHz = 384000U }

audio sample rate definition
- enum {

kCODEC_AudioBitWidth16bit = 16U,

kCODEC_AudioBitWidth20bit = 20U,

kCODEC_AudioBitWidth24bit = 24U,

kCODEC_AudioBitWidth32bit = 32U }

audio bit width
- enum `codec_module_t` {

kCODEC_ModuleADC = 0U,

kCODEC_ModuleDAC = 1U,

kCODEC_ModulePGA = 2U,

kCODEC_ModuleHeadphone = 3U,

kCODEC_ModuleSpeaker = 4U,

kCODEC_ModuleLinein = 5U,

kCODEC_ModuleLineout = 6U,

kCODEC_ModuleVref = 7U,

kCODEC_ModuleMicbias = 8U,

kCODEC_ModuleMic = 9U,

kCODEC_ModuleI2SIn = 10U,

kCODEC_ModuleI2SOut = 11U,

kCODEC_ModuleMixer = 12U }

audio codec module
- enum `codec_module_ctrl_cmd_t` { kCODEC_ModuleSwitchI2SInInterface = 0U }

audio codec module control cmd
- enum {

kCODEC_ModuleI2SInInterfacePCM = 0U,

kCODEC_ModuleI2SInInterfaceDSD = 1U }

audio codec module digital interface
- enum {

kCODEC_RecordSourceDifferentialLine = 1U,

kCODEC_RecordSourceLineInput = 2U,

kCODEC_RecordSourceDifferentialMic = 4U,

kCODEC_RecordSourceDigitalMic = 8U,

```

kCODEC_RecordSourceSingleEndMic = 16U }

    audio codec module record source value

• enum {
    kCODEC_RecordChannelLeft1 = 1U,
    kCODEC_RecordChannelLeft2 = 2U,
    kCODEC_RecordChannelLeft3 = 4U,
    kCODEC_RecordChannelRight1 = 1U,
    kCODEC_RecordChannelRight2 = 2U,
    kCODEC_RecordChannelRight3 = 4U,
    kCODEC_RecordChannelDifferentialPositive1 = 1U,
    kCODEC_RecordChannelDifferentialPositive2 = 2U,
    kCODEC_RecordChannelDifferentialPositive3 = 4U,
    kCODEC_RecordChannelDifferentialNegative1 = 8U,
    kCODEC_RecordChannelDifferentialNegative2 = 16U,
    kCODEC_RecordChannelDifferentialNegative3 = 32U }

    audio codec record channel

• enum {
    kCODEC_PlaySourcePGA = 1U,
    kCODEC_PlaySourceInput = 2U,
    kCODEC_PlaySourceDAC = 4U,
    kCODEC_PlaySourceMixerIn = 1U,
    kCODEC_PlaySourceMixerInLeft = 2U,
    kCODEC_PlaySourceMixerInRight = 4U,
    kCODEC_PlaySourceAux = 8U }

    audio codec module play source value

• enum {
    kCODEC_PlayChannelHeadphoneLeft = 1U,
    kCODEC_PlayChannelHeadphoneRight = 2U,
    kCODEC_PlayChannelSpeakerLeft = 4U,
    kCODEC_PlayChannelSpeakerRight = 8U,
    kCODEC_PlayChannelLineOutLeft = 16U,
    kCODEC_PlayChannelLineOutRight = 32U,
    kCODEC_PlayChannelLeft0 = 1U,
    kCODEC_PlayChannelRight0 = 2U,
    kCODEC_PlayChannelLeft1 = 4U,
    kCODEC_PlayChannelRight1 = 8U,
    kCODEC_PlayChannelLeft2 = 16U,
    kCODEC_PlayChannelRight2 = 32U,
    kCODEC_PlayChannelLeft3 = 64U,
    kCODEC_PlayChannelRight3 = 128U }

    codec play channel

• enum {

```

```
kCODEC_VolumeHeadphoneLeft = 1U,  
kCODEC_VolumeHeadphoneRight = 2U,  
kCODEC_VolumeSpeakerLeft = 4U,  
kCODEC_VolumeSpeakerRight = 8U,  
kCODEC_VolumeLineOutLeft = 16U,  
kCODEC_VolumeLineOutRight = 32U,  
kCODEC_VolumeLeft0 = 1UL << 0U,  
kCODEC_VolumeRight0 = 1UL << 1U,  
kCODEC_VolumeLeft1 = 1UL << 2U,  
kCODEC_VolumeRight1 = 1UL << 3U,  
kCODEC_VolumeLeft2 = 1UL << 4U,  
kCODEC_VolumeRight2 = 1UL << 5U,  
kCODEC_VolumeLeft3 = 1UL << 6U,  
kCODEC_VolumeRight3 = 1UL << 7U,  
kCODEC_VolumeDAC = 1UL << 8U }
```

codec volume setting

- enum {

```
kCODEC_SupportModuleADC = 1U << 0U,  
kCODEC_SupportModuleDAC = 1U << 1U,  
kCODEC_SupportModulePGA = 1U << 2U,  
kCODEC_SupportModuleHeadphone = 1U << 3U,  
kCODEC_SupportModuleSpeaker = 1U << 4U,  
kCODEC_SupportModuleLinein = 1U << 5U,  
kCODEC_SupportModuleLineout = 1U << 6U,  
kCODEC_SupportModuleVref = 1U << 7U,  
kCODEC_SupportModuleMicbias = 1U << 8U,  
kCODEC_SupportModuleMic = 1U << 9U,  
kCODEC_SupportModuleI2SIn = 1U << 10U,  
kCODEC_SupportModuleI2SOut = 1U << 11U,  
kCODEC_SupportModuleMixer = 1U << 12U,  
kCODEC_SupportModuleI2SInSwitchInterface = 1U << 13U,  
kCODEC_SupportPlayChannelLeft0 = 1U << 0U,  
kCODEC_SupportPlayChannelRight0 = 1U << 1U,  
kCODEC_SupportPlayChannelLeft1 = 1U << 2U,  
kCODEC_SupportPlayChannelRight1 = 1U << 3U,  
kCODEC_SupportPlayChannelLeft2 = 1U << 4U,  
kCODEC_SupportPlayChannelRight2 = 1U << 5U,  
kCODEC_SupportPlayChannelLeft3 = 1U << 6U,  
kCODEC_SupportPlayChannelRight3 = 1U << 7U,  
kCODEC_SupportPlaySourcePGA = 1U << 8U,  
kCODEC_SupportPlaySourceInput = 1U << 9U,  
kCODEC_SupportPlaySourceDAC = 1U << 10U,  
kCODEC_SupportPlaySourceMixerIn = 1U << 11U,  
kCODEC_SupportPlaySourceMixerInLeft = 1U << 12U,  
kCODEC_SupportPlaySourceMixerInRight = 1U << 13U,  
kCODEC_SupportPlaySourceAux = 1U << 14U,  
kCODEC_SupportRecordSourceDifferentialLine = 1U << 0U,  
kCODEC_SupportRecordSourceLineInput = 1U << 1U,  
kCODEC_SupportRecordSourceDifferentialMic = 1U << 2U,  
kCODEC_SupportRecordSourceDigitalMic = 1U << 3U,  
kCODEC_SupportRecordSourceSingleEndMic = 1U << 4U,  
kCODEC_SupportRecordChannelLeft1 = 1U << 6U,  
kCODEC_SupportRecordChannelLeft2 = 1U << 7U,  
kCODEC_SupportRecordChannelLeft3 = 1U << 8U,  
kCODEC_SupportRecordChannelRight1 = 1U << 9U,  
kCODEC_SupportRecordChannelRight2 = 1U << 10U,  
kCODEC_SupportRecordChannelRight3 = 1U << 11U }
```

audio codec capability

Functions

- `status_t CODEC_Init (codec_handle_t *handle, codec_config_t *config)`
Codec initialization.
- `status_t CODEC_Deinit (codec_handle_t *handle)`
Codec de-initilization.
- `status_t CODEC_SetFormat (codec_handle_t *handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`
set audio data format.
- `status_t CODEC_ModuleControl (codec_handle_t *handle, codec_module_ctrl_cmd_t cmd, uint32_t data)`
codec module control.
- `status_t CODEC_SetVolume (codec_handle_t *handle, uint32_t channel, uint32_t volume)`
set audio codec pl volume.
- `status_t CODEC_SetMute (codec_handle_t *handle, uint32_t channel, bool mute)`
set audio codec module mute.
- `status_t CODEC_SetPower (codec_handle_t *handle, codec_module_t module, bool powerOn)`
set audio codec power.
- `status_t CODEC_SetRecord (codec_handle_t *handle, uint32_t recordSource)`
codec set record source.
- `status_t CODEC_SetRecordChannel (codec_handle_t *handle, uint32_t leftRecordChannel, uint32_t rightRecordChannel)`
codec set record channel.
- `status_t CODEC_SetPlay (codec_handle_t *handle, uint32_t playSource)`
codec set play source.

Driver version

- `#define FSL_CODEC_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))`
CLOCK driver version 2.3.1.

41.2.2 Data Structure Documentation

41.2.2.1 struct codec_config_t

Data Fields

- `uint32_t codecDevType`
codec type
- `void *codecDevConfig`
Codec device specific configuration.

41.2.2.2 struct codec_capability_t

Data Fields

- `uint32_t codecModuleCapability`
codec module capability
- `uint32_t codecPlayCapability`
codec play capability
- `uint32_t codecRecordCapability`
codec record capability
- `uint32_t codecVolumeCapability`
codec volume capability

41.2.2.3 struct _codec_handle

codec handle declaration

- Application should allocate a buffer with CODEC_HANDLE_SIZE for handle definition, such as `uint8_t codecHandleBuffer[CODEC_HANDLE_SIZE]; codec_handle_t *codecHandle = codecHandleBuffer;`

Data Fields

- `codec_config_t * codecConfig`
codec configuration function pointer
- `const codec_capability_t * codecCapability`
codec capability
- `uint8_t codecDevHandle [HAL_CODEC_HANDLER_SIZE]`
codec device handle

41.2.3 Macro Definition Documentation

41.2.3.1 #define FSL_CODEC_DRIVER_VERSION (MAKE_VERSION(2, 3, 1))

41.2.4 Enumeration Type Documentation

41.2.4.1 anonymous enum

Enumerator

`kStatus_CODEC_NotSupport` CODEC not support status.

`kStatus_CODEC_DeviceNotRegistered` CODEC device register failed status.

`kStatus_CODEC_I2CBusInitialFailed` CODEC i2c bus initialization failed status.

`kStatus_CODEC_I2CCommandTransferFailed` CODEC i2c bus command transfer failed status.

41.2.4.2 enum codec_audio_protocol_t

Enumerator

- kCODEC_BusI2S* I2S type.
- kCODEC_BusLeftJustified* Left justified mode.
- kCODEC_BusRightJustified* Right justified mode.
- kCODEC_BusPCMA* DSP/PCM A mode.
- kCODEC_BusPCMB* DSP/PCM B mode.
- kCODEC_BusTDM* TDM mode.

41.2.4.3 anonymous enum

Enumerator

- kCODEC_AudioSampleRate8KHz* Sample rate 8000 Hz.
- kCODEC_AudioSampleRate11025Hz* Sample rate 11025 Hz.
- kCODEC_AudioSampleRate12KHz* Sample rate 12000 Hz.
- kCODEC_AudioSampleRate16KHz* Sample rate 16000 Hz.
- kCODEC_AudioSampleRate22050Hz* Sample rate 22050 Hz.
- kCODEC_AudioSampleRate24KHz* Sample rate 24000 Hz.
- kCODEC_AudioSampleRate32KHz* Sample rate 32000 Hz.
- kCODEC_AudioSampleRate44100Hz* Sample rate 44100 Hz.
- kCODEC_AudioSampleRate48KHz* Sample rate 48000 Hz.
- kCODEC_AudioSampleRate96KHz* Sample rate 96000 Hz.
- kCODEC_AudioSampleRate192KHz* Sample rate 192000 Hz.
- kCODEC_AudioSampleRate384KHz* Sample rate 384000 Hz.

41.2.4.4 anonymous enum

Enumerator

- kCODEC_AudioBitWidth16bit* audio bit width 16
- kCODEC_AudioBitWidth20bit* audio bit width 20
- kCODEC_AudioBitWidth24bit* audio bit width 24
- kCODEC_AudioBitWidth32bit* audio bit width 32

41.2.4.5 enum codec_module_t

Enumerator

- kCODEC_ModuleADC* codec module ADC
- kCODEC_ModuleDAC* codec module DAC
- kCODEC_ModulePGA* codec module PGA
- kCODEC_ModuleHeadphone* codec module headphone

kCODEC_ModuleSpeaker codec module speaker
kCODEC_ModuleLinein codec module linein
kCODEC_ModuleLineout codec module lineout
kCODEC_ModuleVref codec module VREF
kCODEC_ModuleMicbias codec module MIC BIAS
kCODEC_ModuleMic codec module MIC
kCODEC_ModuleI2SIn codec module I2S in
kCODEC_ModuleI2SOut codec module I2S out
kCODEC_ModuleMixer codec module mixer

41.2.4.6 enum codec_module_ctrl_cmd_t

Enumerator

kCODEC_ModuleSwitchI2SInInterface module digital interface siwtch.

41.2.4.7 anonymous enum

Enumerator

kCODEC_ModuleI2SInInterfacePCM Pcm interface.
kCODEC_ModuleI2SInInterfaceDSD DSD interface.

41.2.4.8 anonymous enum

Enumerator

kCODEC_RecordSourceDifferentialLine record source from differential line
kCODEC_RecordSourceLineInput record source from line input
kCODEC_RecordSourceDifferentialMic record source from differential mic
kCODEC_RecordSourceDigitalMic record source from digital microphone
kCODEC_RecordSourceSingleEndMic record source from single microphone

41.2.4.9 anonymous enum

Enumerator

kCODEC_RecordChannelLeft1 left record channel 1
kCODEC_RecordChannelLeft2 left record channel 2
kCODEC_RecordChannelLeft3 left record channel 3
kCODEC_RecordChannelRight1 right record channel 1
kCODEC_RecordChannelRight2 right record channel 2
kCODEC_RecordChannelRight3 right record channel 3
kCODEC_RecordChannelDifferentialPositive1 differential positive record channel 1

kCODEC_RecordChannelDifferentialPositive2 differential positive record channel 2
kCODEC_RecordChannelDifferentialPositive3 differential positive record channel 3
kCODEC_RecordChannelDifferentialNegative1 differential negative record channel 1
kCODEC_RecordChannelDifferentialNegative2 differential negative record channel 2
kCODEC_RecordChannelDifferentialNegative3 differential negative record channel 3

41.2.4.10 anonymous enum

Enumerator

kCODEC_PlaySourcePGA play source PGA, bypass ADC
kCODEC_PlaySourceInput play source Input3
kCODEC_PlaySourceDAC play source DAC
kCODEC_PlaySourceMixerIn play source mixer in
kCODEC_PlaySourceMixerInLeft play source mixer in left
kCODEC_PlaySourceMixerInRight play source mixer in right
kCODEC_PlaySourceAux play source mixer in AUX

41.2.4.11 anonymous enum

Enumerator

kCODEC_PlayChannelHeadphoneLeft play channel headphone left
kCODEC_PlayChannelHeadphoneRight play channel headphone right
kCODEC_PlayChannelSpeakerLeft play channel speaker left
kCODEC_PlayChannelSpeakerRight play channel speaker right
kCODEC_PlayChannelLineOutLeft play channel lineout left
kCODEC_PlayChannelLineOutRight play channel lineout right
kCODEC_PlayChannelLeft0 play channel left0
kCODEC_PlayChannelRight0 play channel right0
kCODEC_PlayChannelLeft1 play channel left1
kCODEC_PlayChannelRight1 play channel right1
kCODEC_PlayChannelLeft2 play channel left2
kCODEC_PlayChannelRight2 play channel right2
kCODEC_PlayChannelLeft3 play channel left3
kCODEC_PlayChannelRight3 play channel right3

41.2.4.12 anonymous enum

Enumerator

kCODEC_VolumeHeadphoneLeft headphone left volume
kCODEC_VolumeHeadphoneRight headphone right volume
kCODEC_VolumeSpeakerLeft speaker left volume
kCODEC_VolumeSpeakerRight speaker right volume

kCODEC_VolumeLineOutLeft lineout left volume
kCODEC_VolumeLineOutRight lineout right volume
kCODEC_VolumeLeft0 left0 volume
kCODEC_VolumeRight0 right0 volume
kCODEC_VolumeLeft1 left1 volume
kCODEC_VolumeRight1 right1 volume
kCODEC_VolumeLeft2 left2 volume
kCODEC_VolumeRight2 right2 volume
kCODEC_VolumeLeft3 left3 volume
kCODEC_VolumeRight3 right3 volume
kCODEC_VolumeDAC dac volume

41.2.4.13 anonymous enum

Enumerator

kCODEC_SupportModuleADC codec capability of module ADC
kCODEC_SupportModuleDAC codec capability of module DAC
kCODEC_SupportModulePGA codec capability of module PGA
kCODEC_SupportModuleHeadphone codec capability of module headphone
kCODEC_SupportModuleSpeaker codec capability of module speaker
kCODEC_SupportModuleLinein codec capability of module linein
kCODEC_SupportModuleLineout codec capability of module lineout
kCODEC_SupportModuleVref codec capability of module vref
kCODEC_SupportModuleMicbias codec capability of module mic bias
kCODEC_SupportModuleMic codec capability of module mic bias
kCODEC_SupportModuleI2SIn codec capability of module I2S in
kCODEC_SupportModuleI2SOut codec capability of module I2S out
kCODEC_SupportModuleMixer codec capability of module mixer
kCODEC_SupportModuleI2SInSwitchInterface codec capability of module I2S in switch interface

kCODEC_SupportPlayChannelLeft0 codec capability of play channel left 0
kCODEC_SupportPlayChannelRight0 codec capability of play channel right 0
kCODEC_SupportPlayChannelLeft1 codec capability of play channel left 1
kCODEC_SupportPlayChannelRight1 codec capability of play channel right 1
kCODEC_SupportPlayChannelLeft2 codec capability of play channel left 2
kCODEC_SupportPlayChannelRight2 codec capability of play channel right 2
kCODEC_SupportPlayChannelLeft3 codec capability of play channel left 3
kCODEC_SupportPlayChannelRight3 codec capability of play channel right 3
kCODEC_SupportPlaySourcePGA codec capability of set playback source PGA
kCODEC_SupportPlaySourceInput codec capability of set playback source INPUT
kCODEC_SupportPlaySourceDAC codec capability of set playback source DAC
kCODEC_SupportPlaySourceMixerIn codec capability of set play source Mixer in
kCODEC_SupportPlaySourceMixerInLeft codec capability of set play source Mixer in left
kCODEC_SupportPlaySourceMixerInRight codec capability of set play source Mixer in right

kCODEC_SupportPlaySourceAux codec capability of set play source aux

kCODEC_SupportRecordSourceDifferentialLine codec capability of record source differential line

kCODEC_SupportRecordSourceLineInput codec capability of record source line input

kCODEC_SupportRecordSourceDifferentialMic codec capability of record source differential mic

kCODEC_SupportRecordSourceDigitalMic codec capability of record digital mic

kCODEC_SupportRecordSourceSingleEndMic codec capability of single end mic

kCODEC_SupportRecordChannelLeft1 left record channel 1

kCODEC_SupportRecordChannelLeft2 left record channel 2

kCODEC_SupportRecordChannelLeft3 left record channel 3

kCODEC_SupportRecordChannelRight1 right record channel 1

kCODEC_SupportRecordChannelRight2 right record channel 2

kCODEC_SupportRecordChannelRight3 right record channel 3

41.2.5 Function Documentation

41.2.5.1 status_t CODEC_Init (***codec_handle_t * handle***, ***codec_config_t * config***)

Parameters

<i>handle</i>	codec handle.
<i>config</i>	codec configurations.

Returns

kStatus_Success is success, else de-initial failed.

41.2.5.2 status_t CODEC_Deinit (***codec_handle_t * handle***)

Parameters

<i>handle</i>	codec handle.
---------------	---------------

Returns

kStatus_Success is success, else de-initial failed.

41.2.5.3 status_t CODEC_SetFormat (***codec_handle_t * handle***, ***uint32_t mclk***, ***uint32_t sampleRate***, ***uint32_t bitWidth***)

Parameters

<i>handle</i>	codec handle.
<i>mclk</i>	master clock frequency in HZ.
<i>sampleRate</i>	sample rate in HZ.
<i>bitWidth</i>	bit width.

Returns

kStatus_Success is success, else configure failed.

41.2.5.4 status_t CODEC_ModuleControl (*codec_handle_t * handle*, *codec_module_ctrl_cmd_t cmd*, *uint32_t data*)

This function is used for codec module control, support switch digital interface cmd, can be expand to support codec module specific feature.

Parameters

<i>handle</i>	codec handle.
<i>cmd</i>	module control cmd, reference _codec_module_ctrl_cmd.
<i>data</i>	value to write, when cmd is kCODEC_ModuleRecordSourceChannel, the data should be a value combine of channel and source, please reference macro CODEC_MODULE_RECORD_SOURCE_CHANNEL(source, LP, LN, RP, RN), reference codec specific driver for detail configurations.

Returns

kStatus_Success is success, else configure failed.

41.2.5.5 status_t CODEC_SetVolume (*codec_handle_t * handle*, *uint32_t channel*, *uint32_t volume*)

Parameters

<i>handle</i>	codec handle.
<i>channel</i>	audio codec volume channel, can be a value or combine value of _codec_volume_-capability or _codec_play_channel.
<i>volume</i>	volume value, support 0 ~ 100, 0 is mute, 100 is the maximum volume value.

Returns

kStatus_Success is success, else configure failed.

41.2.5.6 status_t CODEC_SetMute (*codec_handle_t * handle*, *uint32_t channel*, *bool mute*)

Parameters

<i>handle</i>	codec handle.
<i>channel</i>	audio codec volume channel, can be a value or combine value of _codec_volume_-capability or _codec_play_channel.
<i>mute</i>	true is mute, false is unmute.

Returns

kStatus_Success is success, else configure failed.

41.2.5.7 status_t CODEC_SetPower (*codec_handle_t * handle*, *codec_module_t module*, *bool powerOn*)

Parameters

<i>handle</i>	codec handle.
<i>module</i>	audio codec module.
<i>powerOn</i>	true is power on, false is power down.

Returns

kStatus_Success is success, else configure failed.

41.2.5.8 status_t CODEC_SetRecord (*codec_handle_t * handle*, *uint32_t recordSource*)

Parameters

<i>handle</i>	codec handle.
<i>recordSource</i>	audio codec record source, can be a value or combine value of _codec_record_source.

Returns

kStatus_Success is success, else configure failed.

41.2.5.9 status_t CODEC_SetRecordChannel (*codec_handle_t * handle*, *uint32_t leftRecordChannel*, *uint32_t rightRecordChannel*)

Parameters

<i>handle</i>	codec handle.
<i>leftRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.
<i>rightRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.

Returns

kStatus_Success is success, else configure failed.

41.2.5.10 status_t CODEC_SetPlay (*codec_handle_t * handle*, *uint32_t playSource*)

Parameters

<i>handle</i>	codec handle.
<i>playSource</i>	audio codec play source, can be a value or combine value of _codec_play_source.

Returns

kStatus_Success is success, else configure failed.

41.3 CODEC I2C Driver

41.3.1 Overview

The codec common driver provides a codec control abstraction interface.

Data Structures

- struct `codec_i2c_config_t`
CODEC I2C configurations structure. [More...](#)

Macros

- #define `CODEC_I2C_MASTER_HANDLER_SIZE` HAL_I2C_MASTER_HANDLE_SIZE
codec i2c handler

Enumerations

- enum `codec_reg_addr_t` {

`kCODEC_RegAddr8Bit` = 1U,
`kCODEC_RegAddr16Bit` = 2U }
CODEC device register address type.
- enum `codec_reg_width_t` {

`kCODEC_RegWidth8Bit` = 1U,
`kCODEC_RegWidth16Bit` = 2U,
`kCODEC_RegWidth32Bit` = 4U }
CODEC device register width.

Functions

- `status_t CODEC_I2C_Init` (void *handle, uint32_t i2cInstance, uint32_t i2cBaudrate, uint32_t i2cSourceClockHz)
Codec i2c bus initialization.
- `status_t CODEC_I2C_Deinit` (void *handle)
Codec i2c de-initilization.
- `status_t CODEC_I2C_Send` (void *handle, uint8_t deviceAddress, uint32_t subAddress, uint8_t subaddressSize, uint8_t *txBuff, uint8_t txBuffSize)
codec i2c send function.
- `status_t CODEC_I2C_Receive` (void *handle, uint8_t deviceAddress, uint32_t subAddress, uint8_t subaddressSize, uint8_t *rxBuff, uint8_t rxBuffSize)
codec i2c receive function.

41.3.2 Data Structure Documentation

41.3.2.1 struct codec_i2c_config_t

Data Fields

- `uint32_t codecI2CInstance`
i2c bus instance
- `uint32_t codecI2CSourceClock`
i2c bus source clock frequency

41.3.3 Enumeration Type Documentation

41.3.3.1 enum codec_reg_addr_t

Enumerator

`kCODEC_RegAddr8Bit` 8-bit register address.
`kCODEC_RegAddr16Bit` 16-bit register address.

41.3.3.2 enum codec_reg_width_t

Enumerator

`kCODEC_RegWidth8Bit` 8-bit register width.
`kCODEC_RegWidth16Bit` 16-bit register width.
`kCODEC_RegWidth32Bit` 32-bit register width.

41.3.4 Function Documentation

41.3.4.1 status_t CODEC_I2C_Init (void * handle, uint32_t i2cInstance, uint32_t i2cBaudrate, uint32_t i2cSourceClockHz)

Parameters

<code>handle</code>	i2c master handle.
<code>i2cInstance</code>	instance number of the i2c bus, such as 0 is corresponding to I2C0.

<i>i2cBaudrate</i>	i2c baudrate.
<i>i2cSource-ClockHz</i>	i2c source clock frequency.

Returns

kStatus_HAL_I2cSuccess is success, else initial failed.

41.3.4.2 status_t CODEC_I2C_Deinit (void * *handle*)

Parameters

<i>handle</i>	i2c master handle.
---------------	--------------------

Returns

kStatus_HAL_I2cSuccess is success, else deinitial failed.

41.3.4.3 status_t CODEC_I2C_Send (void * *handle*, uint8_t *deviceAddress*, uint32_t *subAddress*, uint8_t *subaddressSize*, uint8_t * *txBuff*, uint8_t *txBuffSize*)

Parameters

<i>handle</i>	i2c master handle.
<i>deviceAddress</i>	codec device address.
<i>subAddress</i>	register address.
<i>subaddressSize</i>	register address width.
<i>txBuff</i>	tx buffer pointer.
<i>txBuffSize</i>	tx buffer size.

Returns

kStatus_HAL_I2cSuccess is success, else send failed.

41.3.4.4 status_t CODEC_I2C_Receive (void * *handle*, uint8_t *deviceAddress*, uint32_t *subAddress*, uint8_t *subaddressSize*, uint8_t * *rxBuff*, uint8_t *rxBuffSize*)

Parameters

<i>handle</i>	i2c master handle.
<i>deviceAddress</i>	codec device address.
<i>subAddress</i>	register address.
<i>subaddressSize</i>	register address width.
<i>rxBuff</i>	rx buffer pointer.
<i>rxBuffSize</i>	rx buffer size.

Returns

kStatus_HAL_I2cSuccess is success, else receive failed.

41.4 DA7212 Driver

41.4.1 Overview

The da7212 driver provides a codec control interface.

Data Structures

- struct `da7212_pll_config_t`
da7212 pll configuration [More...](#)
- struct `da7212_audio_format_t`
da7212 audio format [More...](#)
- struct `da7212_config_t`
DA7212 configure structure. [More...](#)
- struct `da7212_handle_t`
da7212 codec handler [More...](#)

Macros

- #define `DA7212_I2C_HANDLER_SIZE` CODEC_I2C_MASTER_HANDLER_SIZE
da7212 handle size
- #define `DA7212_ADDRESS` (0x1A)
DA7212 I2C address.
- #define `DA7212_HEADPHONE_MAX_VOLUME_VALUE` 0x3FU
da7212 volume setting range

Enumerations

- enum `da7212_Input_t` {

kDA7212_Input_AUX = 0x0,
kDA7212_Input_MIC1_Dig,
kDA7212_Input_MIC1_An,
kDA7212_Input_MIC2 }
DA7212 input source select.
- enum `_da7212_play_channel` {

kDA7212_HeadphoneLeft = 1U,
kDA7212_HeadphoneRight = 2U,
kDA7212_Speaker = 4U }
da7212 play channel
- enum `da7212_Output_t` {

kDA7212_Output_HP = 0x0,
kDA7212_Output_SP }
DA7212 output device select.

- enum _da7212_module {
 kDA7212_ModuleADC,
 kDA7212_ModuleDAC,
 kDA7212_ModuleHeadphone,
 kDA7212_ModuleSpeaker }

DA7212 module.
- enum da7212_dac_source_t {
 kDA7212_DACSourceADC = 0x0U,
 kDA7212_DACSourceInputStream = 0x3U }

DA7212 functionality.
- enum da7212_volume_t {
 kDA7212_DACGainMute = 0x7,
 kDA7212_DACGainM72DB = 0x17,
 kDA7212_DACGainM60DB = 0x1F,
 kDA7212_DACGainM54DB = 0x27,
 kDA7212_DACGainM48DB = 0x2F,
 kDA7212_DACGainM42DB = 0x37,
 kDA7212_DACGainM36DB = 0x3F,
 kDA7212_DACGainM30DB = 0x47,
 kDA7212_DACGainM24DB = 0x4F,
 kDA7212_DACGainM18DB = 0x57,
 kDA7212_DACGainM12DB = 0x5F,
 kDA7212_DACGainM6DB = 0x67,
 kDA7212_DACGain0DB = 0x6F,
 kDA7212_DACGain6DB = 0x77,
 kDA7212_DACGain12DB = 0x7F }

DA7212 volume.
- enum da7212_protocol_t {
 kDA7212_BusI2S = 0x0,
 kDA7212_BusLeftJustified,
 kDA7212_BusRightJustified,
 kDA7212_BusDSPMode }

The audio data transfer protocol choice.
- enum da7212_sys_clk_source_t {
 kDA7212_SysClkSourceMCLK = 0U,
 kDA7212_SysClkSourcePLL = 1U << 14 }

da7212 system clock source
- enum da7212_pll_clk_source_t { kDA7212_PLLClkSourceMCLK = 0U }

DA7212 pll clock source.
- enum da7212_pll_out_clk_t {
 kDA7212_PLLOutputClk11289600 = 11289600U,
 kDA7212_PLLOutputClk12288000 = 12288000U }

DA7212 output clock frequency.
- enum da7212_master_bits_t {

```

kDA7212_MasterBits32PerFrame = 0U,
kDA7212_MasterBits64PerFrame = 1U,
kDA7212_MasterBits128PerFrame = 2U,
kDA7212_MasterBits256PerFrame = 3U }
    master mode bits per frame

```

Functions

- `status_t DA7212_Init (da7212_handle_t *handle, da7212_config_t *codecConfig)`
DA7212 initialize function.
- `status_t DA7212_ConfigAudioFormat (da7212_handle_t *handle, uint32_t masterClock_Hz, uint32_t sampleRate_Hz, uint32_t dataBits)`
Set DA7212 audio format.
- `status_t DA7212_SetPLLConfig (da7212_handle_t *handle, da7212_pll_config_t *config)`
DA7212 set PLL configuration This function will enable the GPIO1 FLL clock output function, so user can see the generated fll output clock frequency from WM8904 GPIO1.
- `void DA7212_ChangeHPVolume (da7212_handle_t *handle, da7212_volume_t volume)`
Set DA7212 playback volume.
- `void DA7212_Mute (da7212_handle_t *handle, bool isMuted)`
Mute or unmute DA7212.
- `void DA7212_ChangeInput (da7212_handle_t *handle, da7212_Input_t DA7212_Input)`
Set the input data source of DA7212.
- `void DA7212_ChangeOutput (da7212_handle_t *handle, da7212_Output_t DA7212_Output)`
Set the output device of DA7212.
- `status_t DA7212_SetChannelVolume (da7212_handle_t *handle, uint32_t channel, uint32_t volume)`
Set module volume.
- `status_t DA7212_SetChannelMute (da7212_handle_t *handle, uint32_t channel, bool isMute)`
Set module mute.
- `status_t DA7212_SetProtocol (da7212_handle_t *handle, da7212_protocol_t protocol)`
Set protocol for DA7212.
- `status_t DA7212_SetMasterModeBits (da7212_handle_t *handle, uint32_t bitWidth)`
Set master mode bits per frame for DA7212.
- `status_t DA7212_WriteRegister (da7212_handle_t *handle, uint8_t u8Register, uint8_t u8RegisterData)`
Write a register for DA7212.
- `status_t DA7212_ReadRegister (da7212_handle_t *handle, uint8_t u8Register, uint8_t *pu8RegisterData)`
Get a register value of DA7212.
- `status_t DA7212_Deinit (da7212_handle_t *handle)`
Deinit DA7212.

Driver version

- `#define FSL_DA7212_DRIVER_VERSION (MAKE_VERSION(2, 2, 3))`
CLOCK driver version 2.2.3.

41.4.2 Data Structure Documentation

41.4.2.1 struct da7212_pll_config_t

Data Fields

- `da7212_pll_clk_source_t source`
pll reference clock source
- `uint32_t refClock_HZ`
pll reference clock frequency
- `da7212_pll_out_clk_t outputClock_HZ`
pll output clock frequency

41.4.2.2 struct da7212_audio_format_t

Data Fields

- `uint32_t mclk_HZ`
master clock frequency
- `uint32_t sampleRate`
sample rate
- `uint32_t bitWidth`
bit width
- `bool isBclkInvert`
bit clock invertet

41.4.2.3 struct da7212_config_t

Data Fields

- `bool isMaster`
If DA7212 is master, true means master, false means slave.
- `da7212_protocol_t protocol`
Audio bus format, can be I2S, LJ, RJ or DSP mode.
- `da7212_dac_source_t dacSource`
DA7212 data source.
- `da7212_audio_format_t format`
audio format
- `uint8_t slaveAddress`
device address
- `codec_i2c_config_t i2cConfig`
i2c configuration
- `da7212_sys_clk_source_t sysClkSource`
system clock source
- `da7212_pll_config_t * pll`
pll configuration

Field Documentation

- (1) `bool da7212_config_t::isMaster`
- (2) `da7212_protocol_t da7212_config_t::protocol`
- (3) `da7212_dac_source_t da7212_config_t::dacSource`

41.4.2.4 struct da7212_handle_t

Data Fields

- `da7212_config_t * config`
da7212 config pointer
- `uint8_t i2cHandle [DA7212_I2C_HANDLER_SIZE]`
i2c handle

41.4.3 Macro Definition Documentation

41.4.3.1 #define FSL_DA7212_DRIVER_VERSION (MAKE_VERSION(2, 2, 3))

41.4.4 Enumeration Type Documentation

41.4.4.1 enum da7212_Input_t

Enumerator

- kDA7212_Input_AUX* Input from AUX.
- kDA7212_Input_MIC1_Dig* Input from MIC1 Digital.
- kDA7212_Input_MIC1_An* Input from Mic1 Analog.
- kDA7212_Input_MIC2* Input from MIC2.

41.4.4.2 enum _da7212_play_channel

Enumerator

- kDA7212_HeadphoneLeft* headphone left
- kDA7212_HeadphoneRight* headphone right
- kDA7212_Speaker* speaker channel

41.4.4.3 enum da7212_Output_t

Enumerator

- kDA7212_Output_HP* Output to headphone.

kDA7212_Output_SP Output to speaker.

41.4.4.4 enum _da7212_module

Enumerator

kDA7212_ModuleADC module ADC
kDA7212_ModuleDAC module DAC
kDA7212_ModuleHeadphone module headphone
kDA7212_ModuleSpeaker module speaker

41.4.4.5 enum da7212_dac_source_t

Enumerator

kDA7212_DACSourceADC DAC source from ADC.
kDA7212_DACSourceInputStream DAC source from.

41.4.4.6 enum da7212_volume_t

Enumerator

kDA7212_DACGainMute Mute DAC.
kDA7212_DACGainM72DB DAC volume -72db.
kDA7212_DACGainM60DB DAC volume -60db.
kDA7212_DACGainM54DB DAC volume -54db.
kDA7212_DACGainM48DB DAC volume -48db.
kDA7212_DACGainM42DB DAC volume -42db.
kDA7212_DACGainM36DB DAC volume -36db.
kDA7212_DACGainM30DB DAC volume -30db.
kDA7212_DACGainM24DB DAC volume -24db.
kDA7212_DACGainM18DB DAC volume -18db.
kDA7212_DACGainM12DB DAC volume -12db.
kDA7212_DACGainM6DB DAC volume -6db.
kDA7212_DACGain0DB DAC volume +0db.
kDA7212_DACGain6DB DAC volume +6db.
kDA7212_DACGain12DB DAC volume +12db.

41.4.4.7 enum da7212_protocol_t

Enumerator

kDA7212_BusI2S I2S Type.

kDA7212_BusLeftJustified Left justified.
kDA7212_BusRightJustified Right Justified.
kDA7212_BusDSPMode DSP mode.

41.4.4.8 enum da7212_sys_clk_source_t

Enumerator

kDA7212_SysClkSourceMCLK da7212 system clock soure from MCLK
kDA7212_SysClkSourcePLL da7212 system clock soure from pLL

41.4.4.9 enum da7212_pll_clk_source_t

Enumerator

kDA7212_PLLClkSourceMCLK DA7212 PLL clock source from MCLK.

41.4.4.10 enum da7212_pll_out_clk_t

Enumerator

kDA7212_PLLOutputClk11289600 output 112896000U
kDA7212_PLLOutputClk12288000 output 12288000U

41.4.4.11 enum da7212_master_bits_t

Enumerator

kDA7212_MasterBits32PerFrame master mode bits32 per frame
kDA7212_MasterBits64PerFrame master mode bits64 per frame
kDA7212_MasterBits128PerFrame master mode bits128 per frame
kDA7212_MasterBits256PerFrame master mode bits256 per frame

41.4.5 Function Documentation

41.4.5.1 status_t DA7212_Init (da7212_handle_t * handle, da7212_config_t * codecConfig)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>codecConfig</i>	<p>Codec configure structure. This parameter can be NULL, if NULL, set as default settings. The default setting:</p> <pre>* sgtl_init_t codec_config * codec_config.route = kDA7212_RoutePlayback * codec_config.bus = kDA7212_BusI2S * codec_config.isMaster = false *</pre>

41.4.5.2 status_t DA7212_ConfigAudioFormat (*da7212_handle_t * handle, uint32_t masterClock_Hz, uint32_t sampleRate_Hz, uint32_t dataBits*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>masterClock_Hz</i>	Master clock frequency in Hz. If DA7212 is slave, use the frequency of master, if DA7212 as master, it should be 1228000 while sample rate frequency is 8k/12K/16-K/24K/32K/48K/96K, 11289600 whie sample rate is 11.025K/22.05K/44.1K
<i>sampleRate_Hz</i>	Sample rate frequency in Hz.
<i>dataBits</i>	How many bits in a word of a audio frame, DA7212 only supports 16/20/24/32 bits.

41.4.5.3 status_t DA7212_SetPLLConfig (*da7212_handle_t * handle, da7212_pll_config_t * config*)

Parameters

<i>handle</i>	DA7212 handler pointer.
<i>config</i>	PLL configuration pointer.

41.4.5.4 void DA7212_ChangeHPVolume (*da7212_handle_t * handle, da7212_volume_t volume*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>volume</i>	The volume of playback.

41.4.5.5 void DA7212_Mute (*da7212_handle_t * handle, bool isMuted*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>isMuted</i>	True means mute, false means unmute.

41.4.5.6 void DA7212_ChangeInput (*da7212_handle_t * handle, da7212_Input_t DA7212_Input*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>DA7212_Input</i>	Input data source.

41.4.5.7 void DA7212_ChangeOutput (*da7212_handle_t * handle, da7212_Output_t DA7212_Output*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>DA7212_Output</i>	Output device of DA7212.

41.4.5.8 status_t DA7212_SetChannelVolume (*da7212_handle_t * handle, uint32_t channel, uint32_t volume*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>channel</i>	shoule be a value of _da7212_channel.
<i>volume</i>	volume range 0 - 0x3F mapped to range -57dB - 6dB.

41.4.5.9 status_t DA7212_SetChannelMute (*da7212_handle_t * handle*, *uint32_t channel*, *bool isMute*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>channel</i>	shoule be a value of _da7212_channel.
<i>isMute</i>	true is mute, false is unmute.

41.4.5.10 status_t DA7212_SetProtocol (*da7212_handle_t * handle*, *da7212_protocol_t protocol*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>protocol</i>	da7212_protocol_t.

41.4.5.11 status_t DA7212_SetMasterModeBits (*da7212_handle_t * handle*, *uint32_t bitWidth*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>bitWidth</i>	audio data bitwidth.

41.4.5.12 status_t DA7212_WriteRegister (*da7212_handle_t * handle*, *uint8_t u8Register*, *uint8_t u8RegisterData*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>u8Register</i>	DA7212 register address to be written.
<i>u8RegisterData</i>	Data to be written into register

41.4.5.13 status_t DA7212_ReadRegister (da7212_handle_t * *handle*, uint8_t *u8Register*, uint8_t * *pu8RegisterData*)

Parameters

<i>handle</i>	DA7212 handle pointer.
<i>u8Register</i>	DA7212 register address to be read.
<i>pu8RegisterData</i>	Pointer where the read out value to be stored.

41.4.5.14 status_t DA7212_Deinit (da7212_handle_t * *handle*)

Parameters

<i>handle</i>	DA7212 handle pointer.
---------------	------------------------

41.4.6 DA7212 Adapter

41.4.6.1 Overview

The da7212 adapter provides a codec unify control interface.

Macros

- `#define HAL_CODEC_DA7212_HANDLER_SIZE (DA7212_I2C_HANDLER_SIZE + 4)`
codec handler size

Functions

- `status_t HAL_CODEC_DA7212_Init (void *handle, void *config)`
Codec initialization.
- `status_t HAL_CODEC_DA7212_Deinit (void *handle)`
Codec de-initilization.
- `status_t HAL_CODEC_DA7212_SetFormat (void *handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`
set audio data format.
- `status_t HAL_CODEC_DA7212_SetVolume (void *handle, uint32_t playChannel, uint32_t volume)`
set audio codec module volume.
- `status_t HAL_CODEC_DA7212_SetMute (void *handle, uint32_t playChannel, bool isMute)`
set audio codec module mute.
- `status_t HAL_CODEC_DA7212_SetPower (void *handle, uint32_t module, bool powerOn)`
set audio codec module power.
- `status_t HAL_CODEC_DA7212_SetRecord (void *handle, uint32_t recordSource)`
codec set record source.
- `status_t HAL_CODEC_DA7212_SetRecordChannel (void *handle, uint32_t leftRecordChannel, uint32_t rightRecordChannel)`
codec set record channel.
- `status_t HAL_CODEC_DA7212_SetPlay (void *handle, uint32_t playSource)`
codec set play source.
- `status_t HAL_CODEC_DA7212_ModuleControl (void *handle, uint32_t cmd, uint32_t data)`
codec module control.
- `static status_t HAL_CODEC_Init (void *handle, void *config)`
Codec initilization.
- `static status_t HAL_CODEC_Deinit (void *handle)`
Codec de-initilization.
- `static status_t HAL_CODEC_SetFormat (void *handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`
set audio data format.
- `static status_t HAL_CODEC_SetVolume (void *handle, uint32_t playChannel, uint32_t volume)`
set audio codec module volume.
- `static status_t HAL_CODEC_SetMute (void *handle, uint32_t playChannel, bool isMute)`
set audio codec module mute.
- `static status_t HAL_CODEC_SetPower (void *handle, uint32_t module, bool powerOn)`

- static `status_t HAL_CODEC_SetRecord` (void *handle, uint32_t recordSource)
codec set record source.
- static `status_t HAL_CODEC_SetRecordChannel` (void *handle, uint32_t leftRecordChannel, uint32_t rightRecordChannel)
codec set record channel.
- static `status_t HAL_CODEC_SetPlay` (void *handle, uint32_t playSource)
codec set play source.
- static `status_t HAL_CODEC_ModuleControl` (void *handle, uint32_t cmd, uint32_t data)
codec module control.

41.4.6.2 Function Documentation

41.4.6.2.1 `status_t HAL_CODEC_DA7212_Init(void * handle, void * config)`

Parameters

<i>handle</i>	codec handle.
<i>config</i>	codec configuration.

Returns

kStatus_Success is success, else initial failed.

41.4.6.2.2 `status_t HAL_CODEC_DA7212_Deinit(void * handle)`

Parameters

<i>handle</i>	codec handle.
---------------	---------------

Returns

kStatus_Success is success, else de-initial failed.

41.4.6.2.3 `status_t HAL_CODEC_DA7212_SetFormat(void * handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`

Parameters

<i>handle</i>	codec handle.
<i>mclk</i>	master clock frequency in HZ.
<i>sampleRate</i>	sample rate in HZ.
<i>bitWidth</i>	bit width.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.4 status_t HAL_CODEC_DA7212_SetVolume (void * *handle*, uint32_t *playChannel*, uint32_t *volume*)

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>volume</i>	volume value, support 0 ~ 100, 0 is mute, 100 is the maximum volume value.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.5 status_t HAL_CODEC_DA7212_SetMute (void * *handle*, uint32_t *playChannel*, bool *isMute*)

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>isMute</i>	true is mute, false is unmute.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.6 status_t HAL_CODEC_DA7212_SetPower (void * *handle*, uint32_t *module*, bool *powerOn*)

Parameters

<i>handle</i>	codec handle.
<i>module</i>	audio codec module.
<i>powerOn</i>	true is power on, false is power down.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.7 status_t HAL_CODEC_DA7212_SetRecord (void * *handle*, uint32_t *recordSource*)

Parameters

<i>handle</i>	codec handle.
<i>recordSource</i>	audio codec record source, can be a value or combine value of _codec_record_source.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.8 status_t HAL_CODEC_DA7212_SetRecordChannel (void * *handle*, uint32_t *leftRecordChannel*, uint32_t *rightRecordChannel*)

Parameters

<i>handle</i>	codec handle.
<i>leftRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value or combine value of member in _codec_record_channel.
<i>rightRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.9 status_t HAL_CODEC_DA7212_SetPlay (void * *handle*, uint32_t *playSource*)

Parameters

<i>handle</i>	codec handle.
<i>playSource</i>	audio codec play source, can be a value or combine value of _codec_play_source.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.10 status_t HAL_CODEC_DA7212_ModuleControl (void * *handle*, uint32_t *cmd*, uint32_t *data*)

This function is used for codec module control, support switch digital interface cmd, can be expand to support codec module specific feature

Parameters

<i>handle</i>	codec handle.
<i>cmd</i>	module control cmd, reference _codec_module_ctrl_cmd.
<i>data</i>	value to write, when cmd is kCODEC_ModuleRecordSourceChannel, the data should be a value combine of channel and source, please reference macro CODEC_MODULE_RECORD_SOURCE_CHANNEL(source, LP, LN, RP, RN), reference codec specific driver for detail configurations.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.11 static status_t HAL_CODEC_Init (void * *handle*, void * *config*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>config</i>	codec configuration.

Returns

kStatus_Success is success, else initial failed.

41.4.6.2.12 static status_t HAL_CODEC_Deinit (void * *handle*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
---------------	---------------

Returns

kStatus_Success is success, else de-initial failed.

41.4.6.2.13 static status_t HAL_CODEC_SetFormat(void * *handle*, uint32_t *mclk*, uint32_t *sampleRate*, uint32_t *bitWidth*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>mclk</i>	master clock frequency in HZ.
<i>sampleRate</i>	sample rate in HZ.
<i>bitWidth</i>	bit width.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.14 static status_t HAL_CODEC_SetVolume(void * *handle*, uint32_t *playChannel*, uint32_t *volume*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>volume</i>	volume value, support 0 ~ 100, 0 is mute, 100 is the maximum volume value.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.15 static status_t HAL_CODEC_SetMute(void * *handle*, uint32_t *playChannel*, bool *isMute*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>isMute</i>	true is mute, false is unmute.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.16 static status_t HAL_CODEC_SetPower (void * *handle*, uint32_t *module*, bool *powerOn*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>module</i>	audio codec module.
<i>powerOn</i>	true is power on, false is power down.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.17 static status_t HAL_CODEC_SetRecord (void * *handle*, uint32_t *recordSource*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>recordSource</i>	audio codec record source, can be a value or combine value of _codec_record_source.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.18 static status_t HAL_CODEC_SetRecordChannel (void * *handle*, uint32_t *leftRecordChannel*, uint32_t *rightRecordChannel*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>leftRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value or combine value of member in _codec_record_channel.
<i>rightRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.19 static status_t HAL_CODEC_SetPlay (void * *handle*, uint32_t *playSource*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playSource</i>	audio codec play source, can be a value or combine value of _codec_play_source.

Returns

kStatus_Success is success, else configure failed.

41.4.6.2.20 static status_t HAL_CODEC_ModuleControl (void * *handle*, uint32_t *cmd*, uint32_t *data*) [inline], [static]

This function is used for codec module control, support switch digital interface cmd, can be expand to support codec module specific feature

Parameters

<i>handle</i>	codec handle.
<i>cmd</i>	module control cmd, reference _codec_module_ctrl_cmd.
<i>data</i>	value to write, when cmd is kCODEC_ModuleRecordSourceChannel, the data should be a value combine of channel and source, please reference macro CODEC_MODULE_RECORD_SOURCE_CHANNEL(source, LP, LN, RP, RN), reference codec specific driver for detail configurations.

Returns

kStatus_Success is success, else configure failed.

41.5 SGTL5000 Driver

41.5.1 Overview

The sgtl5000 driver provides a codec control interface.

Data Structures

- struct `sgtl_audio_format_t`
Audio format configuration. [More...](#)
- struct `sgtl_config_t`
Initialize structure of sgtl5000. [More...](#)
- struct `sgtl_handle_t`
SGTL codec handler. [More...](#)

Macros

- #define `CHIP_ID` 0x0000U
Define the register address of sgtl5000.
- #define `SGTL5000_HEADPHONE_MAX_VOLUME_VALUE` 0x7FU
SGTL5000 volume setting range.
- #define `SGTL5000_I2C_ADDR` 0x0A
SGTL5000 I2C address.
- #define `SGTL_I2C_HANDLER_SIZE` CODEC_I2C_MASTER_HANDLER_SIZE
sgtl handle size
- #define `SGTL_I2C_BITRATE` 100000U
sgtl i2c baudrate

Enumerations

- enum `sgtl_module_t` {

`kSGTL_ModuleADC` = 0x0,
`kSGTL_ModuleDAC`,
`kSGTL_ModuleDAP`,
`kSGTL_ModuleHP`,
`kSGTL_ModuleI2SIN`,
`kSGTL_ModuleI2SOUT`,
`kSGTL_ModuleLineIn`,
`kSGTL_ModuleLineOut`,
`kSGTL_ModuleMicin` }
- Modules in Sglt5000 board.*
- enum `sgtl_route_t` {

```

kSGTL_RouteBypass = 0x0,
kSGTL_RoutePlayback,
kSGTL_RoutePlaybackandRecord,
kSGTL_RoutePlaybackwithDAP,
kSGTL_RoutePlaybackwithDAPandRecord,
kSGTL_RouteRecord }

Sgtl5000 data route.
• enum sgtl_protocol_t {
    kSGTL_BusI2S = 0x0,
    kSGTL_BusLeftJustified,
    kSGTL_BusRightJustified,
    kSGTL_BusPCMA,
    kSGTL_BusPCMB }

The audio data transfer protocol choice.
• enum {
    kSGTL_HeadphoneLeft = 0,
    kSGTL_HeadphoneRight = 1,
    kSGTL_LineoutLeft = 2,
    kSGTL_LineoutRight = 3 }

sgtl play channel
• enum {
    kSGTL_RecordSourceLineIn = 0U,
    kSGTL_RecordSourceMic = 1U }

sgtl record source _sgtl_record_source
• enum {
    kSGTL_PlaySourceLineIn = 0U,
    kSGTL_PlaySourceDAC = 1U }

sgtl play source _stgl_play_source
• enum sgtl_sclk_edge_t {
    kSGTL_SclkValidEdgeRising = 0U,
    kSGTL_SclkValidEdgeFailling = 1U }

SGTL SCLK valid edge.

```

Functions

- `status_t SGTL_Init (sgtl_handle_t *handle, sgtl_config_t *config)`
sgtl5000 initialize function.
- `status_t SGTL_SetDataRoute (sgtl_handle_t *handle, sgtl_route_t route)`
Set audio data route in sgtl5000.
- `status_t SGTL_SetProtocol (sgtl_handle_t *handle, sgtl_protocol_t protocol)`
Set the audio transfer protocol.
- `void SGTL_SetMasterSlave (sgtl_handle_t *handle, bool master)`
Set sgtl5000 as master or slave.
- `status_t SGTL_SetVolume (sgtl_handle_t *handle, sgtl_module_t module, uint32_t volume)`
Set the volume of different modules in sgtl5000.
- `uint32_t SGTL_GetVolume (sgtl_handle_t *handle, sgtl_module_t module)`
Get the volume of different modules in sgtl5000.

- `status_t SGTL_SetMute (sgtl_handle_t *handle, sgtl_module_t module, bool mute)`
Mute/unmute modules in sgtl5000.
- `status_t SGTL_EnableModule (sgtl_handle_t *handle, sgtl_module_t module)`
Enable expected devices.
- `status_t SGTL_DisableModule (sgtl_handle_t *handle, sgtl_module_t module)`
Disable expected devices.
- `status_t SGTL_Deinit (sgtl_handle_t *handle)`
Deinit the sglt5000 codec.
- `status_t SGTL_ConfigDataFormat (sgtl_handle_t *handle, uint32_t mclk, uint32_t sample_rate, uint32_t bits)`
Configure the data format of audio data.
- `status_t SGTL_SetPlay (sgtl_handle_t *handle, uint32_t playSource)`
select SGTL codec play source.
- `status_t SGTL_SetRecord (sgtl_handle_t *handle, uint32_t recordSource)`
select SGTL codec record source.
- `status_t SGTL_WriteReg (sgtl_handle_t *handle, uint16_t reg, uint16_t val)`
Write register to sglt using I2C.
- `status_t SGTL_ReadReg (sgtl_handle_t *handle, uint16_t reg, uint16_t *val)`
Read register from sglt using I2C.
- `status_t SGTL_ModifyReg (sgtl_handle_t *handle, uint16_t reg, uint16_t clr_mask, uint16_t val)`
Modify some bits in the register using I2C.

Driver version

- `#define FSL_SGTL5000_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))`
CLOCK driver version 2.1.1.

41.5.2 Data Structure Documentation

41.5.2.1 struct sgtl_audio_format_t

Data Fields

- `uint32_t mclk_HZ`
master clock
- `uint32_t sampleRate`
Sample rate.
- `uint32_t bitWidth`
Bit width.
- `sgtl_sclk_edge_t sclkEdge`
sclk valid edge

41.5.2.2 struct sgtl_config_t

Data Fields

- `sgtl_route_t route`

- *Audio data route.*
- `sgtl_protocol_t bus`
Audio transfer protocol.
- `bool master_slave`
Master or slave.
- `sgtl_audio_format_t format`
audio format
- `uint8_t slaveAddress`
code device slave address
- `codec_i2c_config_t i2cConfig`
i2c bus configuration

Field Documentation

- (1) `sgtl_route_t sgtl_config_t::route`
- (2) `bool sgtl_config_t::master_slave`

True means master, false means slave.

41.5.2.3 struct `sgtl_handle_t`

Data Fields

- `sgtl_config_t * config`
sgtl config pointer
- `uint8_t i2cHandle [SGTL_I2C_HANDLER_SIZE]`
i2c handle

41.5.3 Macro Definition Documentation

41.5.3.1 #define `FSL_SGTL5000_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 1)`)

41.5.3.2 #define `CHIP_ID` `0x0000U`

41.5.3.3 #define `SGTL5000_I2C_ADDR` `0x0A`

41.5.4 Enumeration Type Documentation

41.5.4.1 enum `sgtl_module_t`

Enumerator

- kSGTL_ModuleADC* ADC module in SGTL5000.
- kSGTL_ModuleDAC* DAC module in SGTL5000.
- kSGTL_ModuleDAP* DAP module in SGTL5000.
- kSGTL_ModuleHP* Headphone module in SGTL5000.

kSGTL_ModuleI2SIN I2S-IN module in SGTL5000.
kSGTL_ModuleI2SOUT I2S-OUT module in SGTL5000.
kSGTL_ModuleLineIn Line-in moudle in SGTL5000.
kSGTL_ModuleLineOut Line-out module in SGTL5000.
kSGTL_ModuleMicin Micphone module in SGTL5000.

41.5.4.2 enum sgtl_route_t

Note

Only provide some typical data route, not all route listed. Users cannot combine any routes, once a new route is set, the previous one would be replaced.

Enumerator

kSGTL_RouteBypass LINEIN->Headphone.
kSGTL_RoutePlayback I2SIN->DAC->Headphone.
kSGTL_RoutePlaybackandRecord I2SIN->DAC->Headphone, LINEIN->ADC->I2SOUT.
kSGTL_RoutePlaybackwithDAP I2SIN->DAP->DAC->Headphone.
kSGTL_RoutePlaybackwithDAPandRecord I2SIN->DAP->DAC->HP, LINEIN->ADC->I2SOUT.
kSGTL_RouteRecord LINEIN->ADC->I2SOUT.

41.5.4.3 enum sgtl_protocol_t

Sgtl5000 only supports I2S format and PCM format.

Enumerator

kSGTL_BusI2S I2S Type.
kSGTL_BusLeftJustified Left justified.
kSGTL_BusRightJustified Right Justified.
kSGTL_BusPCMA PCMA.
kSGTL_BusPCMB PCMB.

41.5.4.4 anonymous enum

Enumerator

kSGTL_HeadphoneLeft headphone left channel
kSGTL_HeadphoneRight headphone right channel
kSGTL_LineoutLeft lineout left channel
kSGTL_LineoutRight lineout right channel

41.5.4.5 anonymous enum

Enumerator

kSGTL_RecordSourceLineIn record source line in
kSGTL_RecordSourceMic record source single end

41.5.4.6 anonymous enum

Enumerator

kSGTL_PlaySourceLineIn play source line in
kSGTL_PlaySourceDAC play source line in

41.5.4.7 enum sgtl_sclk_edge_t

Enumerator

kSGTL_SclkValidEdgeRising SCLK valid edge.
kSGTL_SclkValidEdgeFailling SCLK failling edge.

41.5.5 Function Documentation

41.5.5.1 status_t SGTL_Init(sgtl_handle_t * handle, sgtl_config_t * config)

This function calls SGTL_I2CInit(), and in this function, some configurations are fixed. The second parameter can be NULL. If users want to change the SGTL5000 settings, a configure structure should be prepared.

Note

If the codec_config is NULL, it would initialize sgtl5000 using default settings. The default setting:

```
* sgtl_init_t codec_config
* codec_config.route = kSGTL_RoutePlaybackandRecord
* codec_config.bus = kSGTL_BusI2S
* codec_config.master = slave
*
```

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>config</i>	sgtl5000 configuration structure. If this pointer equals to NULL, it means using the default configuration.

Returns

Initialization status

41.5.5.2 status_t SGTL_SetDataRoute (sgtl_handle_t * *handle*, sgtl_route_t *route*)

This function would set the data route according to route. The route cannot be combined, as all route would enable different modules.

Note

If a new route is set, the previous route would not work.

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>route</i>	Audio data route in sgtl5000.

41.5.5.3 status_t SGTL_SetProtocol (sgtl_handle_t * *handle*, sgtl_protocol_t *protocol*)

Sgtl5000 only supports I2S, I2S left, I2S right, PCM A, PCM B format.

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>protocol</i>	Audio data transfer protocol.

41.5.5.4 void SGTL_SetMasterSlave (sgtl_handle_t * *handle*, bool *master*)

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>master</i>	1 represent master, 0 represent slave.

41.5.5.5 status_t SGTL_SetVolume (*sgtl_handle_t * handle*, *sgtl_module_t module*, *uint32_t volume*)

This function would set the volume of sgtl5000 modules. This interface set module volume. The function assume that left channel and right channel has the same volume.

kSGTL_ModuleADC volume range: 0 - 0xF, 0dB - 22.5dB
kSGTL_ModuleDAC volume range: 0x3C - 0xF0, 0dB - -90dB
kSGTL_ModuleHP volume range: 0 - 0x7F, 12dB - -51.5dB
kSGTL_ModuleLineOut volume range: 0 - 0x1F, 0.5dB steps

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>module</i>	Sgtl5000 module, such as DAC, ADC and etc.
<i>volume</i>	Volume value need to be set. The value is the exact value in register.

41.5.5.6 uint32_t SGTL_GetVolume (*sgtl_handle_t * handle*, *sgtl_module_t module*)

This function gets the volume of sgtl5000 modules. This interface get DAC module volume. The function assume that left channel and right channel has the same volume.

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>module</i>	Sgtl5000 module, such as DAC, ADC and etc.

Returns

Module value, the value is exact value in register.

41.5.5.7 status_t SGTL_SetMute (*sgtl_handle_t * handle*, *sgtl_module_t module*, *bool mute*)

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>module</i>	Sgtl5000 module, such as DAC, ADC and etc.
<i>mute</i>	True means mute, and false means unmute.

41.5.5.8 status_t SGTL_EnableModule (*sgtl_handle_t * handle*, *sgtl_module_t module*)

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>module</i>	Module expected to enable.

41.5.5.9 status_t SGTL_DisableModule (*sgtl_handle_t * handle*, *sgtl_module_t module*)

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>module</i>	Module expected to enable.

41.5.5.10 status_t SGTL_Deinit (*sgtl_handle_t * handle*)

Shut down Sgtl5000 modules.

Parameters

<i>handle</i>	Sgtl5000 handle structure pointer.
---------------	------------------------------------

41.5.5.11 status_t SGTL_ConfigDataFormat (*sgtl_handle_t * handle*, *uint32_t mclk*, *uint32_t sample_rate*, *uint32_t bits*)

This function would configure the registers about the sample rate, bit depths.

Parameters

<i>handle</i>	Sgtl5000 handle structure pointer.
<i>mclk</i>	Master clock frequency of I2S.
<i>sample_rate</i>	Sample rate of audio file running in sgtl5000. Sgtl5000 now supports 8k, 11.025k, 12k, 16k, 22.05k, 24k, 32k, 44.1k, 48k and 96k sample rate.
<i>bits</i>	Bit depth of audio file (Sgtl5000 only supports 16bit, 20bit, 24bit and 32 bit in HW).

41.5.5.12 status_t SGTL_SetPlay (*sgtl_handle_t * handle, uint32_t playSource*)

Parameters

<i>handle</i>	Sgtl5000 handle structure pointer.
<i>playSource</i>	play source value, reference _sgtl_play_source.

Returns

kStatus_Success, else failed.

41.5.5.13 status_t SGTL_SetRecord (*sgtl_handle_t * handle, uint32_t recordSource*)

Parameters

<i>handle</i>	Sgtl5000 handle structure pointer.
<i>recordSource</i>	record source value, reference _sgtl_record_source.

Returns

kStatus_Success, else failed.

41.5.5.14 status_t SGTL_WriteReg (*sgtl_handle_t * handle, uint16_t reg, uint16_t val*)

Parameters

<i>handle</i>	Sgtl5000 handle structure.
---------------	----------------------------

<i>reg</i>	The register address in sgtl.
<i>val</i>	Value needs to write into the register.

41.5.5.15 status_t SGTL_ReadReg (*sgtl_handle_t * handle, uint16_t reg, uint16_t * val*)

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>reg</i>	The register address in sgtl.
<i>val</i>	Value written to.

41.5.5.16 status_t SGTL_ModifyReg (*sgtl_handle_t * handle, uint16_t reg, uint16_t clr_mask, uint16_t val*)

Parameters

<i>handle</i>	Sgtl5000 handle structure.
<i>reg</i>	The register address in sgtl.
<i>clr_mask</i>	The mask code for the bits want to write. The bit you want to write should be 0.
<i>val</i>	Value needs to write into the register.

41.5.6 SGTL5000 Adapter

41.5.6.1 Overview

The sgtl5000 adapter provides a codec unify control interface.

Macros

- `#define HAL_CODEC_SGTL_HANDLER_SIZE (SGTL_I2C_HANDLER_SIZE + 4)`
codec handler size

Functions

- `status_t HAL_CODEC_SGTL5000_Init (void *handle, void *config)`
Codec initialization.
- `status_t HAL_CODEC_SGTL5000_Deinit (void *handle)`
Codec de-initilization.
- `status_t HAL_CODEC_SGTL5000_SetFormat (void *handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`
set audio data format.
- `status_t HAL_CODEC_SGTL5000_SetVolume (void *handle, uint32_t playChannel, uint32_t volume)`
set audio codec module volume.
- `status_t HAL_CODEC_SGTL5000_SetMute (void *handle, uint32_t playChannel, bool isMute)`
set audio codec module mute.
- `status_t HAL_CODEC_SGTL5000_SetPower (void *handle, uint32_t module, bool powerOn)`
set audio codec module power.
- `status_t HAL_CODEC_SGTL5000_SetRecord (void *handle, uint32_t recordSource)`
codec set record source.
- `status_t HAL_CODEC_SGTL5000_SetRecordChannel (void *handle, uint32_t leftRecordChannel, uint32_t rightRecordChannel)`
codec set record channel.
- `status_t HAL_CODEC_SGTL5000_SetPlay (void *handle, uint32_t playSource)`
codec set play source.
- `status_t HAL_CODEC_SGTL5000_ModuleControl (void *handle, uint32_t cmd, uint32_t data)`
codec module control.
- `static status_t HAL_CODEC_Init (void *handle, void *config)`
Codec initilization.
- `static status_t HAL_CODEC_Deinit (void *handle)`
Codec de-initilization.
- `static status_t HAL_CODEC_SetFormat (void *handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`
set audio data format.
- `static status_t HAL_CODEC_SetVolume (void *handle, uint32_t playChannel, uint32_t volume)`
set audio codec module volume.
- `static status_t HAL_CODEC_SetMute (void *handle, uint32_t playChannel, bool isMute)`
set audio codec module mute.
- `static status_t HAL_CODEC_SetPower (void *handle, uint32_t module, bool powerOn)`

- static `status_t HAL_CODEC_SetRecord` (void *handle, uint32_t recordSource)
codec set record source.
- static `status_t HAL_CODEC_SetRecordChannel` (void *handle, uint32_t leftRecordChannel, uint32_t rightRecordChannel)
codec set record channel.
- static `status_t HAL_CODEC_SetPlay` (void *handle, uint32_t playSource)
codec set play source.
- static `status_t HAL_CODEC_ModuleControl` (void *handle, uint32_t cmd, uint32_t data)
codec module control.

41.5.6.2 Function Documentation

41.5.6.2.1 `status_t HAL_CODEC_SGTL5000_Init(void * handle, void * config)`

Parameters

<i>handle</i>	codec handle.
<i>config</i>	codec configuration.

Returns

kStatus_Success is success, else initial failed.

41.5.6.2.2 `status_t HAL_CODEC_SGTL5000_Deinit(void * handle)`

Parameters

<i>handle</i>	codec handle.
---------------	---------------

Returns

kStatus_Success is success, else de-initial failed.

41.5.6.2.3 `status_t HAL_CODEC_SGTL5000_SetFormat(void * handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`

Parameters

<i>handle</i>	codec handle.
<i>mclk</i>	master clock frequency in HZ.
<i>sampleRate</i>	sample rate in HZ.
<i>bitWidth</i>	bit width.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.4 status_t HAL_CODEC_SGTL5000_SetVolume (void * *handle*, uint32_t *playChannel*, uint32_t *volume*)

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>volume</i>	volume value, support 0 ~ 100, 0 is mute, 100 is the maximum volume value.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.5 status_t HAL_CODEC_SGTL5000_SetMute (void * *handle*, uint32_t *playChannel*, bool *isMute*)

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>isMute</i>	true is mute, false is unmute.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.6 status_t HAL_CODEC_SGTL5000_SetPower (void * *handle*, uint32_t *module*, bool *powerOn*)

Parameters

<i>handle</i>	codec handle.
<i>module</i>	audio codec module.
<i>powerOn</i>	true is power on, false is power down.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.7 status_t HAL_CODEC_SGTL5000_SetRecord (void * *handle*, uint32_t *recordSource*)

Parameters

<i>handle</i>	codec handle.
<i>recordSource</i>	audio codec record source, can be a value or combine value of _codec_record_source.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.8 status_t HAL_CODEC_SGTL5000_SetRecordChannel (void * *handle*, uint32_t *leftRecordChannel*, uint32_t *rightRecordChannel*)

Parameters

<i>handle</i>	codec handle.
<i>leftRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value or combine value of member in _codec_record_channel.
<i>rightRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.9 status_t HAL_CODEC_SGTL5000_SetPlay (void * *handle*, uint32_t *playSource*)

Parameters

<i>handle</i>	codec handle.
<i>playSource</i>	audio codec play source, can be a value or combine value of _codec_play_source.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.10 **status_t HAL_CODEC_SGTL5000_ModuleControl (void * *handle*, uint32_t *cmd*, uint32_t *data*)**

This function is used for codec module control, support switch digital interface cmd, can be expand to support codec module specific feature

Parameters

<i>handle</i>	codec handle.
<i>cmd</i>	module control cmd, reference _codec_module_ctrl_cmd.
<i>data</i>	value to write, when cmd is kCODEC_ModuleRecordSourceChannel, the data should be a value combine of channel and source, please reference macro CODEC_MODULE_RECORD_SOURCE_CHANNEL(source, LP, LN, RP, RN), reference codec specific driver for detail configurations.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.11 **static status_t HAL_CODEC_Init (void * *handle*, void * *config*) [inline], [static]**

Parameters

<i>handle</i>	codec handle.
<i>config</i>	codec configuration.

Returns

kStatus_Success is success, else initial failed.

41.5.6.2.12 **static status_t HAL_CODEC_Deinit (void * *handle*) [inline], [static]**

Parameters

<i>handle</i>	codec handle.
---------------	---------------

Returns

kStatus_Success is success, else de-initial failed.

41.5.6.2.13 static status_t HAL_CODEC_SetFormat(void * *handle*, uint32_t *mclk*, uint32_t *sampleRate*, uint32_t *bitWidth*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>mclk</i>	master clock frequency in HZ.
<i>sampleRate</i>	sample rate in HZ.
<i>bitWidth</i>	bit width.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.14 static status_t HAL_CODEC_SetVolume(void * *handle*, uint32_t *playChannel*, uint32_t *volume*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>volume</i>	volume value, support 0 ~ 100, 0 is mute, 100 is the maximum volume value.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.15 static status_t HAL_CODEC_SetMute(void * *handle*, uint32_t *playChannel*, bool *isMute*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>isMute</i>	true is mute, false is unmute.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.16 static status_t HAL_CODEC_SetPower (void * *handle*, uint32_t *module*, bool *powerOn*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>module</i>	audio codec module.
<i>powerOn</i>	true is power on, false is power down.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.17 static status_t HAL_CODEC_SetRecord (void * *handle*, uint32_t *recordSource*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>recordSource</i>	audio codec record source, can be a value or combine value of _codec_record_source.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.18 static status_t HAL_CODEC_SetRecordChannel (void * *handle*, uint32_t *leftRecordChannel*, uint32_t *rightRecordChannel*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>leftRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value or combine value of member in _codec_record_channel.
<i>rightRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.19 static status_t HAL_CODEC_SetPlay (void * *handle*, uint32_t *playSource*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playSource</i>	audio codec play source, can be a value or combine value of _codec_play_source.

Returns

kStatus_Success is success, else configure failed.

41.5.6.2.20 static status_t HAL_CODEC_ModuleControl (void * *handle*, uint32_t *cmd*, uint32_t *data*) [inline], [static]

This function is used for codec module control, support switch digital interface cmd, can be expand to support codec module specific feature

Parameters

<i>handle</i>	codec handle.
<i>cmd</i>	module control cmd, reference _codec_module_ctrl_cmd.
<i>data</i>	value to write, when cmd is kCODEC_ModuleRecordSourceChannel, the data should be a value combine of channel and source, please reference macro CODEC_MODULE_RECORD_SOURCE_CHANNEL(source, LP, LN, RP, RN), reference codec specific driver for detail configurations.

Returns

kStatus_Success is success, else configure failed.

41.6 WM8960 Driver

41.6.1 Overview

The wm8960 driver provides a codec control interface.

Data Structures

- struct `wm8960_audio_format_t`
wm8960 audio format More...
- struct `wm8960_master_sysclk_config_t`
wm8960 master system clock configuration More...
- struct `wm8960_config_t`
Initialize structure of WM8960. More...
- struct `wm8960_handle_t`
wm8960 codec handler More...

Macros

- #define `WM8960_I2C_HANDLER_SIZE` CODEC_I2C_MASTER_HANDLER_SIZE
wm8960 handle size
- #define `WM8960_LINVOL` 0x0U
Define the register address of WM8960.
- #define `WM8960_CACHEREGNUM` 56U
Cache register number.
- #define `WM8960_CLOCK2_BCLK_DIV_MASK` 0xFU
WM8960 CLOCK2 bits.
- #define `WM8960_IFACE1_FORMAT_MASK` 0x03U
WM8960_IFACE1 FORMAT bits.
- #define `WM8960_IFACE1_WL_MASK` 0x0CU
WM8960_IFACE1 WL bits.
- #define `WM8960_IFACE1_LRP_MASK` 0x10U
WM8960_IFACE1 LRP bit.
- #define `WM8960_IFACE1_DLRSWAP_MASK` 0x20U
WM8960_IFACE1 DLRSWAP bit.
- #define `WM8960_IFACE1_MS_MASK` 0x40U
WM8960_IFACE1 MS bit.
- #define `WM8960_IFACE1_BCLKINV_MASK` 0x80U
WM8960_IFACE1 BCLKINV bit.
- #define `WM8960_IFACE1_ALRSWAP_MASK` 0x100U
WM8960_IFACE1 ALRSWAP bit.
- #define `WM8960_POWER1_VREF_MASK` 0x40U
WM8960_POWER1.
- #define `WM8960_POWER2_DACL_MASK` 0x100U
WM8960_POWER2.
- #define `WM8960_I2C_ADDR` 0x1A
WM8960 I2C address.
- #define `WM8960_I2C_BAUDRATE` (100000U)

- `#define WM8960_ADC_MAX_VOLUME_vVALUE 0xFFU`
WM8960 maximum volume value.

Enumerations

- enum `wm8960_module_t` {

`kWM8960_ModuleADC` = 0,
`kWM8960_ModuleDAC` = 1,
`kWM8960_ModuleVREF` = 2,
`kWM8960_ModuleHP` = 3,
`kWM8960_ModuleMICB` = 4,
`kWM8960_ModuleMIC` = 5,
`kWM8960_ModuleLineIn` = 6,
`kWM8960_ModuleLineOut` = 7,
`kWM8960_ModuleSpeaker` = 8,
`kWM8960_ModuleOMIX` = 9 }

Modules in WM8960 board.
- enum {

`kWM8960_HeadphoneLeft` = 1,
`kWM8960_HeadphoneRight` = 2,
`kWM8960_SpeakerLeft` = 4,
`kWM8960_SpeakerRight` = 8 }

wm8960 play channel
- enum `wm8960_play_source_t` {

`kWM8960_PlaySourcePGA` = 1,
`kWM8960_PlaySourceInput` = 2,
`kWM8960_PlaySourceDAC` = 4 }

wm8960 play source
- enum `wm8960_route_t` {

`kWM8960_RouteBypass` = 0,
`kWM8960_RoutePlayback` = 1,
`kWM8960_RoutePlaybackandRecord` = 2,
`kWM8960_RouteRecord` = 5 }

WM8960 data route.
- enum `wm8960_protocol_t` {

`kWM8960_BusI2S` = 2,
`kWM8960_BusLeftJustified` = 1,
`kWM8960_BusRightJustified` = 0,
`kWM8960_BusPCMA` = 3,
`kWM8960_BusPCMB` = 3 | (1 << 4) }

The audio data transfer protocol choice.
- enum `wm8960_input_t` {

```

kWM8960_InputClosed = 0,
kWM8960_InputSingleEndedMic = 1,
kWM8960_InputDifferentialMicInput2 = 2,
kWM8960_InputDifferentialMicInput3 = 3,
kWM8960_InputLineINPUT2 = 4,
kWM8960_InputLineINPUT3 = 5 }

    wm8960 input source
• enum {
    kWM8960_AudioSampleRate8KHz = 8000U,
    kWM8960_AudioSampleRate11025Hz = 11025U,
    kWM8960_AudioSampleRate12KHz = 12000U,
    kWM8960_AudioSampleRate16KHz = 16000U,
    kWM8960_AudioSampleRate22050Hz = 22050U,
    kWM8960_AudioSampleRate24KHz = 24000U,
    kWM8960_AudioSampleRate32KHz = 32000U,
    kWM8960_AudioSampleRate44100Hz = 44100U,
    kWM8960_AudioSampleRate48KHz = 48000U,
    kWM8960_AudioSampleRate96KHz = 96000U,
    kWM8960_AudioSampleRate192KHz = 192000U,
    kWM8960_AudioSampleRate384KHz = 384000U }

        audio sample rate definition
• enum {
    kWM8960_AudioBitWidth16bit = 16U,
    kWM8960_AudioBitWidth20bit = 20U,
    kWM8960_AudioBitWidth24bit = 24U,
    kWM8960_AudioBitWidth32bit = 32U }

        audio bit width
• enum wm8960_sysclk_source_t {
    kWM8960_SysClkSourceMclk = 0U,
    kWM8960_SysClkSourceInternalPLL = 1U }

    wm8960 sysclk source

```

Functions

- **status_t WM8960_Init (wm8960_handle_t *handle, const wm8960_config_t *config)**
WM8960 initialize function.
- **status_t WM8960_Deinit (wm8960_handle_t *handle)**
Deinit the WM8960 codec.
- **status_t WM8960_SetDataRoute (wm8960_handle_t *handle, wm8960_route_t route)**
Set audio data route in WM8960.
- **status_t WM8960_SetLeftInput (wm8960_handle_t *handle, wm8960_input_t input)**
Set left audio input source in WM8960.
- **status_t WM8960_SetRightInput (wm8960_handle_t *handle, wm8960_input_t input)**
Set right audio input source in WM8960.
- **status_t WM8960_SetProtocol (wm8960_handle_t *handle, **wm8960_protocol_t** protocol)**
Set the audio transfer protocol.

- void [WM8960_SetMasterSlave](#) (wm8960_handle_t *handle, bool master)
Set WM8960 as master or slave.
- status_t [WM8960_SetVolume](#) (wm8960_handle_t *handle, wm8960_module_t module, uint32_t volume)
Set the volume of different modules in WM8960.
- uint32_t [WM8960_GetVolume](#) (wm8960_handle_t *handle, wm8960_module_t module)
Get the volume of different modules in WM8960.
- status_t [WM8960_SetMute](#) (wm8960_handle_t *handle, wm8960_module_t module, bool isEnabled)
Mute modules in WM8960.
- status_t [WM8960_SetModule](#) (wm8960_handle_t *handle, wm8960_module_t module, bool isEnabled)
Enable/disable expected devices.
- status_t [WM8960_SetPlay](#) (wm8960_handle_t *handle, uint32_t playSource)
SET the WM8960 play source.
- status_t [WM8960_ConfigDataFormat](#) (wm8960_handle_t *handle, uint32_t sysclk, uint32_t sample_rate, uint32_t bits)
Configure the data format of audio data.
- status_t [WM8960_SetJackDetect](#) (wm8960_handle_t *handle, bool isEnabled)
Enable/disable jack detect feature.
- status_t [WM8960_WriteReg](#) (wm8960_handle_t *handle, uint8_t reg, uint16_t val)
Write register to WM8960 using I2C.
- status_t [WM8960_ReadReg](#) (uint8_t reg, uint16_t *val)
Read register from WM8960 using I2C.
- status_t [WM8960_ModifyReg](#) (wm8960_handle_t *handle, uint8_t reg, uint16_t mask, uint16_t val)
Modify some bits in the register using I2C.

Driver version

- #define [FSL_WM8960_DRIVER_VERSION](#) (MAKE_VERSION(2, 2, 2))
CLOCK driver version 2.2.2.

41.6.2 Data Structure Documentation

41.6.2.1 struct [wm8960_audio_format_t](#)

Data Fields

- uint32_t [mclk_HZ](#)
master clock frequency
- uint32_t [sampleRate](#)
sample rate
- uint32_t [bitWidth](#)
bit width

41.6.2.2 struct `wm8960_master_sysclk_config_t`

Data Fields

- `wm8960_sysclk_source_t sysclkSource`
sysclk source
- `uint32_t sysclkFreq`
PLL output frequency value.

41.6.2.3 struct `wm8960_config_t`

Data Fields

- `wm8960_route_t route`
Audio data route.
- `wm8960_protocol_t bus`
Audio transfer protocol.
- `wm8960_audio_format_t format`
Audio format.
- `bool master_slave`
Master or slave.
- `wm8960_master_sysclk_config_t masterClock`
master clock configurations
- `bool enableSpeaker`
True means enable class D speaker as output, false means no.
- `wm8960_input_t leftInputSource`
Left input source for WM8960.
- `wm8960_input_t rightInputSource`
Right input source for WM8960.
- `wm8960_play_source_t playSource`
play source
- `uint8_t slaveAddress`
wm8960 device address
- `codec_i2c_config_t i2cConfig`
i2c configuration

Field Documentation

(1) `wm8960_route_t wm8960_config_t::route`

(2) `bool wm8960_config_t::master_slave`

41.6.2.4 struct `wm8960_handle_t`

Data Fields

- `const wm8960_config_t * config`
wm8904 config pointer
- `uint8_t i2cHandle [WM8960_I2C_HANDLER_SIZE]`
i2c handle

41.6.3 Macro Definition Documentation

41.6.3.1 #define WM8960_LINVOL 0x0U

41.6.3.2 #define WM8960_I2C_ADDR 0x1A

41.6.4 Enumeration Type Documentation

41.6.4.1 enum wm8960_module_t

Enumerator

kWM8960_ModuleADC ADC module in WM8960.

kWM8960_ModuleDAC DAC module in WM8960.

kWM8960_ModuleVREF VREF module.

kWM8960_ModuleHP Headphone.

kWM8960_ModuleMICB Mic bias.

kWM8960_ModuleMIC Input Mic.

kWM8960_ModuleLineIn Analog in PGA.

kWM8960_ModuleLineOut Line out module.

kWM8960_ModuleSpeaker Speaker module.

kWM8960_ModuleOMIX Output mixer.

41.6.4.2 anonymous enum

Enumerator

kWM8960_HeadphoneLeft wm8960 headphone left channel

kWM8960_HeadphoneRight wm8960 headphone right channel

kWM8960_SpeakerLeft wm8960 speaker left channel

kWM8960_SpeakerRight wm8960 speaker right channel

41.6.4.3 enum wm8960_play_source_t

Enumerator

kWM8960_PlaySourcePGA wm8960 play source PGA

kWM8960_PlaySourceInput wm8960 play source Input

kWM8960_PlaySourceDAC wm8960 play source DAC

41.6.4.4 enum wm8960_route_t

Only provide some typical data route, not all route listed. Note: Users cannot combine any routes, once a new route is set, the previous one would be replaced.

Enumerator

kWM8960_RouteBypass LINEIN->Headphone.
kWM8960_RoutePlayback I2SIN->DAC->Headphone.
kWM8960_RoutePlaybackandRecord I2SIN->DAC->Headphone, LINEIN->ADC->I2SOUT.
kWM8960_RouteRecord LINEIN->ADC->I2SOUT.

41.6.4.5 enum `wm8960_protocol_t`

WM8960 only supports I2S format and PCM format.

Enumerator

kWM8960_BusI2S I2S type.
kWM8960_BusLeftJustified Left justified mode.
kWM8960_BusRightJustified Right justified mode.
kWM8960_BusPCMA PCM A mode.
kWM8960_BusPCMB PCM B mode.

41.6.4.6 enum `wm8960_input_t`

Enumerator

kWM8960_InputClosed Input device is closed.
kWM8960_InputSingleEndedMic Input as single ended mic, only use L/RINPUT1.
kWM8960_InputDifferentialMicInput2 Input as differential mic, use L/RINPUT1 and L/RINPUT2.
kWM8960_InputDifferentialMicInput3 Input as differential mic, use L/RINPUT1 and L/RINPUT3.
kWM8960_InputLineINPUT2 Input as line input, only use L/RINPUT2.
kWM8960_InputLineINPUT3 Input as line input, only use L/RINPUT3.

41.6.4.7 anonymous enum

Enumerator

kWM8960_AudioSampleRate8KHz Sample rate 8000 Hz.
kWM8960_AudioSampleRate11025Hz Sample rate 11025 Hz.
kWM8960_AudioSampleRate12KHz Sample rate 12000 Hz.
kWM8960_AudioSampleRate16KHz Sample rate 16000 Hz.
kWM8960_AudioSampleRate22050Hz Sample rate 22050 Hz.
kWM8960_AudioSampleRate24KHz Sample rate 24000 Hz.
kWM8960_AudioSampleRate32KHz Sample rate 32000 Hz.
kWM8960_AudioSampleRate44100Hz Sample rate 44100 Hz.
kWM8960_AudioSampleRate48KHz Sample rate 48000 Hz.

kWM8960_AudioSampleRate96KHz Sample rate 96000 Hz.
kWM8960_AudioSampleRate192KHz Sample rate 192000 Hz.
kWM8960_AudioSampleRate384KHz Sample rate 384000 Hz.

41.6.4.8 anonymous enum

Enumerator

kWM8960_AudioBitWidth16bit audio bit width 16
kWM8960_AudioBitWidth20bit audio bit width 20
kWM8960_AudioBitWidth24bit audio bit width 24
kWM8960_AudioBitWidth32bit audio bit width 32

41.6.4.9 enum **wm8960_sysclk_source_t**

Enumerator

kWM8960_SysClkSourceMclk sysclk source from external MCLK
kWM8960_SysClkSourceInternalPLL sysclk source from internal PLL

41.6.5 Function Documentation

41.6.5.1 status_t **WM8960_Init** (**wm8960_handle_t * handle**, **const wm8960_config_t * config**)

The second parameter is NULL to WM8960 in this version. If users want to change the settings, they have to use `wm8960_write_reg()` or `wm8960_modify_reg()` to set the register value of WM8960. Note: If the `codec_config` is NULL, it would initialize WM8960 using default settings. The default setting: `codec_config->route = kWM8960_RoutePlaybackandRecord` `codec_config->bus = kWM8960_BusI2S` `codec_config->master = slave`

Parameters

<i>handle</i>	WM8960 handle structure.
<i>config</i>	WM8960 configuration structure.

41.6.5.2 status_t **WM8960_Deinit** (**wm8960_handle_t * handle**)

This function close all modules in WM8960 to save power.

Parameters

<i>handle</i>	WM8960 handle structure pointer.
---------------	----------------------------------

41.6.5.3 status_t WM8960_SetDataRoute (*wm8960_handle_t * handle, wm8960_route_t route*)

This function would set the data route according to route. The route cannot be combined, as all route would enable different modules. Note: If a new route is set, the previous route would not work.

Parameters

<i>handle</i>	WM8960 handle structure.
<i>route</i>	Audio data route in WM8960.

41.6.5.4 status_t WM8960_SetLeftInput (*wm8960_handle_t * handle, wm8960_input_t input*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>input</i>	Audio input source.

41.6.5.5 status_t WM8960_SetRightInput (*wm8960_handle_t * handle, wm8960_input_t input*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>input</i>	Audio input source.

41.6.5.6 status_t WM8960_SetProtocol (*wm8960_handle_t * handle, wm8960_protocol_t protocol*)

WM8960 only supports I2S, left justified, right justified, PCM A, PCM B format.

Parameters

<i>handle</i>	WM8960 handle structure.
<i>protocol</i>	Audio data transfer protocol.

41.6.5.7 void WM8960_SetMasterSlave (**wm8960_handle_t * handle, bool master**)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>master</i>	1 represent master, 0 represent slave.

41.6.5.8 status_t WM8960_SetVolume (**wm8960_handle_t * handle, wm8960_module_t module, uint32_t volume**)

This function would set the volume of WM8960 modules. Uses need to appoint the module. The function assume that left channel and right channel has the same volume.

Module:kWM8960_ModuleADC, volume range value: 0 is mute, 1-255 is -97db to 30db
 Module:kWM8960_ModuleDAC, volume range value: 0 is mute, 1-255 is -127db to 0db
 Module:kWM8960_ModuleHP, volume range value: 0 - 2F is mute, 0x30 - 0x7F is -73db to 6db
 Module:kWM8960_ModuleLineIn, volume range value: 0 - 0x3F is -17.25db to 30db
 Module:kWM8960_ModuleSpeaker, volume range value: 0 - 2F is mute, 0x30 - 0x7F is -73db to 6db

Parameters

<i>handle</i>	WM8960 handle structure.
<i>module</i>	Module to set volume, it can be ADC, DAC, Headphone and so on.
<i>volume</i>	Volume value need to be set.

41.6.5.9 uint32_t WM8960_GetVolume (**wm8960_handle_t * handle, wm8960_module_t module**)

This function gets the volume of WM8960 modules. Uses need to appoint the module. The function assume that left channel and right channel has the same volume.

Parameters

<i>handle</i>	WM8960 handle structure.
<i>module</i>	Module to set volume, it can be ADC, DAC, Headphone and so on.

Returns

Volume value of the module.

41.6.5.10 status_t WM8960_SetMute (*wm8960_handle_t * handle, wm8960_module_t module, bool isEnabled*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>module</i>	Modules need to be mute.
<i>isEnabled</i>	Mute or unmute, 1 represent mute.

41.6.5.11 status_t WM8960_SetModule (*wm8960_handle_t * handle, wm8960_module_t module, bool isEnabled*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>module</i>	Module expected to enable.
<i>isEnabled</i>	Enable or disable moudles.

41.6.5.12 status_t WM8960_SetPlay (*wm8960_handle_t * handle, uint32_t playSource*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>playSource</i>	play source , can be a value combine of kWM8960_ModuleHeadphoneSourcePG-A, kWM8960_ModuleHeadphoneSourceDAC, kWM8960_ModulePlaySourceInput, kWM8960_ModulePlayMonoRight, kWM8960_ModulePlayMonoLeft.

Returns

kStatus_WM8904_Success if successful, different code otherwise..

41.6.5.13 status_t WM8960_ConfigDataFormat (*wm8960_handle_t * handle, uint32_t sysclk, uint32_t sample_rate, uint32_t bits*)

This function would configure the registers about the sample rate, bit depths.

Parameters

<i>handle</i>	WM8960 handle structure pointer.
<i>sysclk</i>	system clock of the codec which can be generated by MCLK or PLL output.
<i>sample_rate</i>	Sample rate of audio file running in WM8960. WM8960 now supports 8k, 11.025k, 12k, 16k, 22.05k, 24k, 32k, 44.1k, 48k and 96k sample rate.
<i>bits</i>	Bit depth of audio file (WM8960 only supports 16bit, 20bit, 24bit and 32 bit in HW).

41.6.5.14 status_t WM8960_SetJackDetect (*wm8960_handle_t * handle, bool isEnabled*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>isEnabled</i>	Enable or disable moudles.

41.6.5.15 status_t WM8960_WriteReg (*wm8960_handle_t * handle, uint8_t reg, uint16_t val*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>reg</i>	The register address in WM8960.
<i>val</i>	Value needs to write into the register.

41.6.5.16 status_t WM8960_ReadReg (*uint8_t reg, uint16_t * val*)

Parameters

<i>reg</i>	The register address in WM8960.
<i>val</i>	Value written to.

41.6.5.17 status_t WM8960_ModifyReg (*wm8960_handle_t * handle, uint8_t reg, uint16_t mask, uint16_t val*)

Parameters

<i>handle</i>	WM8960 handle structure.
<i>reg</i>	The register address in WM8960.
<i>mask</i>	The mask code for the bits want to write. The bit you want to write should be 0.
<i>val</i>	Value needs to write into the register.

41.6.6 WM8960 Adapter

41.6.6.1 Overview

The wm8960 adapter provides a codec unify control interface.

Macros

- #define `HAL_CODEC_WM8960_HANDLER_SIZE` (`WM8960_I2C_HANDLER_SIZE + 4`)
codec handler size

Functions

- `status_t HAL_CODEC_WM8960_Init` (void *handle, void *config)
Codec initialization.
- `status_t HAL_CODEC_WM8960_Deinit` (void *handle)
Codec de-initilization.
- `status_t HAL_CODEC_WM8960_SetFormat` (void *handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)
set audio data format.
- `status_t HAL_CODEC_WM8960_SetVolume` (void *handle, uint32_t playChannel, uint32_t volume)
set audio codec module volume.
- `status_t HAL_CODEC_WM8960_SetMute` (void *handle, uint32_t playChannel, bool isMute)
set audio codec module mute.
- `status_t HAL_CODEC_WM8960_SetPower` (void *handle, uint32_t module, bool powerOn)
set audio codec module power.
- `status_t HAL_CODEC_WM8960_SetRecord` (void *handle, uint32_t recordSource)
codec set record source.
- `status_t HAL_CODEC_WM8960_SetRecordChannel` (void *handle, uint32_t leftRecordChannel, uint32_t rightRecordChannel)
codec set record channel.
- `status_t HAL_CODEC_WM8960_SetPlay` (void *handle, uint32_t playSource)
codec set play source.
- `status_t HAL_CODEC_WM8960_ModuleControl` (void *handle, uint32_t cmd, uint32_t data)
codec module control.
- static `status_t HAL_CODEC_Init` (void *handle, void *config)
Codec initilization.
- static `status_t HAL_CODEC_Deinit` (void *handle)
Codec de-initilization.
- static `status_t HAL_CODEC_SetFormat` (void *handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)
set audio data format.
- static `status_t HAL_CODEC_SetVolume` (void *handle, uint32_t playChannel, uint32_t volume)
set audio codec module volume.
- static `status_t HAL_CODEC_SetMute` (void *handle, uint32_t playChannel, bool isMute)
set audio codec module mute.
- static `status_t HAL_CODEC_SetPower` (void *handle, uint32_t module, bool powerOn)

- static `status_t HAL_CODEC_SetRecord` (void *handle, uint32_t recordSource)
codec set record source.
- static `status_t HAL_CODEC_SetRecordChannel` (void *handle, uint32_t leftRecordChannel, uint32_t rightRecordChannel)
codec set record channel.
- static `status_t HAL_CODEC_SetPlay` (void *handle, uint32_t playSource)
codec set play source.
- static `status_t HAL_CODEC_ModuleControl` (void *handle, uint32_t cmd, uint32_t data)
codec module control.

41.6.6.2 Function Documentation

41.6.6.2.1 `status_t HAL_CODEC_WM8960_Init(void * handle, void * config)`

Parameters

<i>handle</i>	codec handle.
<i>config</i>	codec configuration.

Returns

kStatus_Success is success, else initial failed.

41.6.6.2.2 `status_t HAL_CODEC_WM8960_Deinit(void * handle)`

Parameters

<i>handle</i>	codec handle.
---------------	---------------

Returns

kStatus_Success is success, else de-initial failed.

41.6.6.2.3 `status_t HAL_CODEC_WM8960_SetFormat(void * handle, uint32_t mclk, uint32_t sampleRate, uint32_t bitWidth)`

Parameters

<i>handle</i>	codec handle.
<i>mclk</i>	master clock frequency in HZ.
<i>sampleRate</i>	sample rate in HZ.
<i>bitWidth</i>	bit width.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.4 status_t HAL_CODEC_WM8960_SetVolume (void * *handle*, uint32_t *playChannel*, uint32_t *volume*)

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>volume</i>	volume value, support 0 ~ 100, 0 is mute, 100 is the maximum volume value.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.5 status_t HAL_CODEC_WM8960_SetMute (void * *handle*, uint32_t *playChannel*, bool *isMute*)

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>isMute</i>	true is mute, false is unmute.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.6 status_t HAL_CODEC_WM8960_SetPower (void * *handle*, uint32_t *module*, bool *powerOn*)

Parameters

<i>handle</i>	codec handle.
<i>module</i>	audio codec module.
<i>powerOn</i>	true is power on, false is power down.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.7 status_t HAL_CODEC_WM8960_SetRecord (void * *handle*, uint32_t *recordSource*)

Parameters

<i>handle</i>	codec handle.
<i>recordSource</i>	audio codec record source, can be a value or combine value of _codec_record_source.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.8 status_t HAL_CODEC_WM8960_SetRecordChannel (void * *handle*, uint32_t *leftRecordChannel*, uint32_t *rightRecordChannel*)

Parameters

<i>handle</i>	codec handle.
<i>leftRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value or combine value of member in _codec_record_channel.
<i>rightRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.9 status_t HAL_CODEC_WM8960_SetPlay (void * *handle*, uint32_t *playSource*)

Parameters

<i>handle</i>	codec handle.
<i>playSource</i>	audio codec play source, can be a value or combine value of _codec_play_source.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.10 status_t HAL_CODEC_WM8960_ModuleControl (void * *handle*, uint32_t *cmd*, uint32_t *data*)

This function is used for codec module control, support switch digital interface cmd, can be expand to support codec module specific feature

Parameters

<i>handle</i>	codec handle.
<i>cmd</i>	module control cmd, reference _codec_module_ctrl_cmd.
<i>data</i>	value to write, when cmd is kCODEC_ModuleRecordSourceChannel, the data should be a value combine of channel and source, please reference macro CODEC_MODULE_RECORD_SOURCE_CHANNEL(source, LP, LN, RP, RN), reference codec specific driver for detail configurations.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.11 static status_t HAL_CODEC_Init (void * *handle*, void * *config*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>config</i>	codec configuration.

Returns

kStatus_Success is success, else initial failed.

41.6.6.2.12 static status_t HAL_CODEC_Deinit (void * *handle*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
---------------	---------------

Returns

kStatus_Success is success, else de-initial failed.

41.6.6.2.13 static status_t HAL_CODEC_SetFormat(void * *handle*, uint32_t *mclk*, uint32_t *sampleRate*, uint32_t *bitWidth*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>mclk</i>	master clock frequency in HZ.
<i>sampleRate</i>	sample rate in HZ.
<i>bitWidth</i>	bit width.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.14 static status_t HAL_CODEC_SetVolume(void * *handle*, uint32_t *playChannel*, uint32_t *volume*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>volume</i>	volume value, support 0 ~ 100, 0 is mute, 100 is the maximum volume value.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.15 static status_t HAL_CODEC_SetMute(void * *handle*, uint32_t *playChannel*, bool *isMute*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playChannel</i>	audio codec play channel, can be a value or combine value of _codec_play_channel.
<i>isMute</i>	true is mute, false is unmute.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.16 static status_t HAL_CODEC_SetPower (void * *handle*, uint32_t *module*, bool *powerOn*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>module</i>	audio codec module.
<i>powerOn</i>	true is power on, false is power down.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.17 static status_t HAL_CODEC_SetRecord (void * *handle*, uint32_t *recordSource*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>recordSource</i>	audio codec record source, can be a value or combine value of _codec_record_source.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.18 static status_t HAL_CODEC_SetRecordChannel (void * *handle*, uint32_t *leftRecordChannel*, uint32_t *rightRecordChannel*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>leftRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value or combine value of member in _codec_record_channel.
<i>rightRecord-Channel</i>	audio codec record channel, reference _codec_record_channel, can be a value combine of member in _codec_record_channel.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.19 static status_t HAL_CODEC_SetPlay (void * *handle*, uint32_t *playSource*) [inline], [static]

Parameters

<i>handle</i>	codec handle.
<i>playSource</i>	audio codec play source, can be a value or combine value of _codec_play_source.

Returns

kStatus_Success is success, else configure failed.

41.6.6.2.20 static status_t HAL_CODEC_ModuleControl (void * *handle*, uint32_t *cmd*, uint32_t *data*) [inline], [static]

This function is used for codec module control, support switch digital interface cmd, can be expand to support codec module specific feature

Parameters

<i>handle</i>	codec handle.
<i>cmd</i>	module control cmd, reference _codec_module_ctrl_cmd.
<i>data</i>	value to write, when cmd is kCODEC_ModuleRecordSourceChannel, the data should be a value combine of channel and source, please reference macro CODEC_MODULE_RECORD_SOURCE_CHANNEL(source, LP, LN, RP, RN), reference codec specific driver for detail configurations.

Returns

kStatus_Success is success, else configure failed.

Chapter 42

Serial Manager

42.1 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 SWO](#)
- [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](#) (1U)
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_PORT_TYPE_BLE_WU](#) (0U)
Enable or disable BLE WU port (1 - enable, 0 - disable)
- #define [SERIAL_MANAGER_TASK_HANDLE_TX](#) (0U)

- `#define SERIAL_MANAGER_WRITE_TIME_DELAY_DEFAULT_VALUE (1U)`
Enable or disable SerialManager_Task() handle TX to prevent recursive calling.
- `#define SERIAL_MANAGER_READ_TIME_DELAY_DEFAULT_VALUE (1U)`
Set the default delay time in ms used by SerialManager_WriteTimeDelay().
- `#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 (44U)`
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 + 124U)`
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 int32_t(* serial_manager_lowpower_critical_callback_t)(int32_t power_mode)`
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`,

 `kSerialPort_BleWu` }

 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_WriteNonBlocking (serial_write_handle_t writeHandle, uint8_t *buffer, uint32_t length)`
Transmits data with the non-blocking mode.
- `serial_manager_status_t SerialManager_ReadNonBlocking (serial_read_handle_t readHandle, uint8_t *buffer, uint32_t length)`
Reads data with the non-blocking mode.
- `serial_manager_status_t SerialManager_TryRead (serial_read_handle_t readHandle, uint8_t *buffer, uint32_t length, uint32_t *receivedLength)`
Tries to read data.
- `serial_manager_status_t SerialManager_CancelWriting (serial_write_handle_t writeHandle)`
Cancels unfinished send transmission.
- `serial_manager_status_t SerialManager_CancelReading (serial_read_handle_t readHandle)`
Cancels unfinished receive transmission.
- `serial_manager_status_t SerialManager_InstallTxCallback (serial_write_handle_t writeHandle, serial_manager_callback_t callback, void *callbackParam)`
Installs a TX callback and callback parameter.
- `serial_manager_status_t SerialManager_InstallRxCallback (serial_read_handle_t readHandle, serial_manager_callback_t callback, void *callbackParam)`
Installs a RX callback and callback parameter.
- `static bool SerialManager_needPollingIsr (void)`
Check if need polling ISR.
- `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 *pfnCallback)`
This function performs initialization of the callbacks structure used to disable lowpower when serial manager is active.

42.2 Data Structure Documentation

42.2.1 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.

42.2.2 `struct serial_manager_callback_message_t`

Data Fields

- `uint8_t * buffer`
Transferred buffer.
- `uint32_t length`
Transferred data length.

42.3 Macro Definition Documentation

42.3.1 `#define SERIAL_MANAGER_WRITE_TIME_DELAY_DEFAULT_VALUE (1U)`

42.3.2 `#define SERIAL_MANAGER_READ_TIME_DELAY_DEFAULT_VALUE (1U)`

42.3.3 `#define SERIAL_MANAGER_USE_COMMON_TASK (0U)`

Macro to determine whether use common task.

42.3.4 `#define SERIAL_MANAGER_HANDLE_SIZE (SERIAL_MANAGER_HANDLE_SIZE_TEMP + 124U)`

42.3.5 `#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.
-------------	---

**42.3.6 #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.
-------------	---

**42.3.7 #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.
-------------	--

42.3.8 #define SERIAL_MANAGER_TASK_PRIORITY (2U)

42.3.9 #define SERIAL_MANAGER_TASK_STACK_SIZE (1000U)

42.4 Enumeration Type Documentation

42.4.1 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.
- kSerialPort_BleWu* Serial port BLE WU.

42.4.2 enum serial_manager_type_t

Enumerator

- kSerialManager_NonBlocking* None blocking handle.
- kSerialManager_Blocking* Blocking handle.

42.4.3 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.

42.5 Function Documentation

42.5.1 `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 <code>SERIAL_MANAGER_HANDLE_SIZE</code> 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: <code>SERIAL_MANAGER_HANDLE_DEFINE(serialHandle)</code> ; 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

<code>kStatus_SerialManager_Error</code>	An error occurred.
<code>kStatus_SerialManager_Success</code>	The Serial Manager module initialization succeed.

42.5.2 `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

<code>kStatus_SerialManager_Success</code>	The serial manager de-initialization succeed.
<code>kStatus_SerialManager_Busy</code>	Opened reading or writing handle is not closed.

42.5.3 `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);
*
*
```

42.5.4 **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.
---------------------------------------	-------------------------------

42.5.5 **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 uint32_t readHandle[((SERIAL_MANAGER_READ_HANDLE_SIZE + sizeof(uint32_t) - 1U) / sizeof(uint32_t))];

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));
*
```

42.5.6 **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.
---------------------------------------	-------------------------------

42.5.7 **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.

42.5.8 `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

<i>readHandle</i>	The serial manager module handle pointer.
<i>buffer</i>	Start address of the data to store the received data.
<i>length</i>	The length of the data to be received.

Return values

<i>kStatus_SerialManager_-Success</i>	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.

42.5.9 `serial_manager_status_t SerialManager_WriteNonBlocking (` `serial_write_handle_t writeHandle, uint8_t * buffer, uint32_t length)`

This is a non-blocking function, which returns directly without waiting for all data to be sent. When all data is sent, the module notifies the upper layer through a TX callback function and passes the status parameter `kStatus_SerialManager_Success`. 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 TX callback is mandatory before the function could be used.

Parameters

<code>writeHandle</code>	The serial manager module handle pointer.
<code>buffer</code>	Start address of the data to write.
<code>length</code>	Length of the data to write.

Return values

<code>kStatus_SerialManager_- Success</code>	Successfully sent all data.
<code>kStatus_SerialManager_- Busy</code>	Previous transmission still not finished; data not all sent yet.
<code>kStatus_SerialManager_- Error</code>	An error occurred.

42.5.10 `serial_manager_status_t SerialManager_ReadNonBlocking (` `serial_read_handle_t readHandle, uint8_t * buffer, uint32_t length)`

This is a non-blocking function, which returns directly without waiting for all data to be received. When all data is received, the module driver notifies the upper layer through a RX callback function and passes the

status parameter `kStatus_SerialManager_Success`. 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 RX callback is mandatory before the function could be used.

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.

42.5.11 `serial_manager_status_t SerialManager_TryRead (serial_read_handle_t readHandle, uint8_t * buffer, uint32_t length, uint32_t * receivedLength)`

The function tries to read data from internal ring buffer. If the ring buffer is not empty, the data will be copied from ring buffer to up layer buffer. The copied length is the minimum of the ring buffer and up layer length. After the data is copied, the actual data length is passed by the parameter length. And There can only one buffer for receiving for the reading handle at the same time.

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.
<code>receivedLength</code>	Length received from the ring buffer directly.

Return values

<i>kStatus_SerialManager_-Success</i>	Successfully received all data.
<i>kStatus_SerialManager_-Busy</i>	Previous transmission still not finished; data not all received yet.
<i>kStatus_SerialManager_-Error</i>	An error occurred.

42.5.12 **serial_manager_status_t SerialManager_CancelWriting (serial_write_handle_t writeHandle)**

The function cancels unfinished send transmission. When the transfer is canceled, the module notifies the upper layer through a TX callback function and passes the status parameter [kStatus_SerialManager_-Canceled](#).

Note

The function [SerialManager_CancelWriting](#) cannot be used to abort the transmission of the function [SerialManager_WriteBlocking](#).

Parameters

<i>writeHandle</i>	The serial manager module handle pointer.
--------------------	---

Return values

<i>kStatus_SerialManager_-Success</i>	Get successfully abort the sending.
<i>kStatus_SerialManager_-Error</i>	An error occurred.

42.5.13 **serial_manager_status_t SerialManager_CancelReading (serial_read_handle_t readHandle)**

The function cancels unfinished receive transmission. When the transfer is canceled, the module notifies the upper layer through a RX callback function and passes the status parameter [kStatus_SerialManager_-Canceled](#).

Note

The function [SerialManager_CancelReading](#) cannot be used to abort the transmission of the function [SerialManager_ReadBlocking](#).

Parameters

<i>readHandle</i>	The serial manager module handle pointer.
-------------------	---

Return values

<i>kStatus_SerialManager_-Success</i>	Get successfully abort the receiving.
<i>kStatus_SerialManager_-Error</i>	An error occurred.

42.5.14 **serial_manager_status_t SerialManager_InstallTxCallback (** **serial_write_handle_t writeHandle, serial_manager_callback_t callback,** **void * callbackParam)**

This function is used to install the TX callback and callback parameter for the serial manager module. When any status of TX transmission changed, the driver will notify the upper layer by the installed callback function. And the status is also passed as status parameter when the callback is called.

Parameters

<i>writeHandle</i>	The serial manager module handle pointer.
<i>callback</i>	The callback function.
<i>callbackParam</i>	The parameter of the callback function.

Return values

<i>kStatus_SerialManager_-Success</i>	Successfully install the callback.
---------------------------------------	------------------------------------

42.5.15 **serial_manager_status_t SerialManager_InstallRxCallback (** **serial_read_handle_t readHandle, serial_manager_callback_t callback,** **void * callbackParam)**

This function is used to install the RX callback and callback parameter for the serial manager module. When any status of RX transmission changed, the driver will notify the upper layer by the installed callback

function. And the status is also passed as status parameter when the callback is called.

Parameters

<i>readHandle</i>	The serial manager module handle pointer.
<i>callback</i>	The callback function.
<i>callbackParam</i>	The parameter of the callback function.

Return values

<i>kStatus_SerialManager_-Success</i>	Successfully install the callback.
---------------------------------------	------------------------------------

42.5.16 static bool SerialManager_needPollingIsr(void) [inline], [static]

This function is used to check if need polling ISR.

Return values

<i>TRUE</i>	if need polling.
-------------	------------------

42.5.17 serial_manager_status_t SerialManager_EnterLowpower(serial_handle_t serialHandle)

This function is used to prepare to enter low power consumption.

Parameters

<i>serialHandle</i>	The serial manager module handle pointer.
---------------------	---

Return values

<i>kStatus_SerialManager_-Success</i>	Successful operation.
---------------------------------------	-----------------------

42.5.18 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.
---------------------------------------	-----------------------

42.5.19 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.
-------------------	---

42.6 Serial Port Uart

42.6.1 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

42.6.2 Enumeration Type Documentation

42.6.2.1 enum serial_port_uart_parity_mode_t

Enumerator

kSerialManager_UartParityDisabled Parity disabled.
kSerialManager_UartParityEven Parity even enabled.
kSerialManager_UartParityOdd Parity odd enabled.

42.6.2.2 enum serial_port_uart_stop_bit_count_t

Enumerator

kSerialManager_UartOneStopBit One stop bit.
kSerialManager_UartTwoStopBit Two stop bits.

42.7 Serial Port SWO

42.7.1 Overview

Data Structures

- struct `serial_port_swo_config_t`
serial port swo config struct [More...](#)

Macros

- #define `SERIAL_PORT_SWO_HANDLE_SIZE` (12U)
serial port swo handle size

Enumerations

- enum `serial_port_swo_protocol_t` {

`kSerialManager_SwoProtocolManchester` = 1U,
`kSerialManager_SwoProtocolNrz` = 2U }

serial port swo protocol

42.7.2 Data Structure Documentation

42.7.2.1 struct `serial_port_swo_config_t`

Data Fields

- `uint32_t clockRate`
clock rate
- `uint32_t baudRate`
baud rate
- `uint32_t port`
Port used to transfer data.
- `serial_port_swo_protocol_t protocol`
SWO protocol.

42.7.3 Enumeration Type Documentation

42.7.3.1 enum `serial_port_swo_protocol_t`

Enumerator

- `kSerialManager_SwoProtocolManchester` SWO Manchester protocol.
`kSerialManager_SwoProtocolNrz` SWO UART/NRZ protocol.

42.7.4 CODEC Adapter

42.7.4.1 Overview

Enumerations

- enum {
 kCODEC_WM8904,
 kCODEC_WM8960,
 kCODEC_WM8524,
 kCODEC_SGTL5000,
 kCODEC_DA7212,
 kCODEC_CS42888,
 kCODEC_CS42448,
 kCODEC_AK4497,
 kCODEC_AK4458,
 kCODEC_TFA9XXX,
 kCODEC_TFA9896,
 kCODEC_WM8962 }
 codec type

42.7.4.2 Enumeration Type Documentation

42.7.4.2.1 anonymous enum

Enumerator

kCODEC_WM8904 wm8904
kCODEC_WM8960 wm8960
kCODEC_WM8524 wm8524
kCODEC_SGTL5000 sgtl5000
kCODEC_DA7212 da7212
kCODEC_CS42888 CS42888.
kCODEC_CS42448 CS42448.
kCODEC_AK4497 AK4497.
kCODEC_AK4458 ak4458
kCODEC_TFA9XXX tfa9xxx
kCODEC_TFA9896 tfa9896
kCODEC_WM8962 wm8962

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.

