
MCUXpresso SDK API Reference Manual

NXP Semiconductors

Document Number: MCUXSDKKW24APIRM
Rev. 0
Mar 2017



Contents

Chapter [Introduction](#)

Chapter [Driver errors status](#)

Chapter [Architectural Overview](#)

Chapter [Trademarks](#)

Chapter [ADC16: 16-bit SAR Analog-to-Digital Converter Driver](#)

5.1	Overview	11
5.2	Typical use case	11
5.2.1	Polling Configuration	11
5.2.2	Interrupt Configuration	11
5.3	Data Structure Documentation	14
5.3.1	struct adc16_config_t	14
5.3.2	struct adc16_hardware_compare_config_t	15
5.3.3	struct adc16_channel_config_t	16
5.4	Macro Definition Documentation	16
5.4.1	FSL_ADC16_DRIVER_VERSION	16
5.5	Enumeration Type Documentation	16
5.5.1	_adc16_channel_status_flags	16
5.5.2	_adc16_status_flags	16
5.5.3	adc16_clock_divider_t	16
5.5.4	adc16_resolution_t	17
5.5.5	adc16_clock_source_t	17
5.5.6	adc16_long_sample_mode_t	17
5.5.7	adc16_reference_voltage_source_t	17
5.5.8	adc16_hardware_compare_mode_t	18
5.6	Function Documentation	18
5.6.1	ADC16_Init	18
5.6.2	ADC16_Deinit	18
5.6.3	ADC16_GetDefaultConfig	18

Contents

Section Number	Title	Page Number
5.6.4	ADC16_EnableHardwareTrigger	19
5.6.5	ADC16_SetHardwareCompareConfig	19
5.6.6	ADC16_GetStatusFlags	19
5.6.7	ADC16_ClearStatusFlags	19
5.6.8	ADC16_SetChannelConfig	20
5.6.9	ADC16_GetChannelConversionValue	20
5.6.10	ADC16_GetChannelStatusFlags	21
Chapter	Clock Driver	
6.1	Overview	23
6.2	Get frequency	23
6.3	External clock frequency	23
6.4	Data Structure Documentation	30
6.4.1	struct sim_clock_config_t	30
6.4.2	struct oscr_config_t	31
6.4.3	struct osc_config_t	31
6.4.4	struct mcg_pll_config_t	32
6.4.5	struct mcg_config_t	32
6.5	Macro Definition Documentation	33
6.5.1	MCG_CONFIG_CHECK_PARAM	33
6.5.2	FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL	34
6.5.3	FSL_CLOCK_DRIVER_VERSION	34
6.5.4	DMAMUX_CLOCKS	34
6.5.5	RTC_CLOCKS	34
6.5.6	SAI_CLOCKS	34
6.5.7	PORT_CLOCKS	34
6.5.8	EWM_CLOCKS	35
6.5.9	PIT_CLOCKS	35
6.5.10	DSPI_CLOCKS	35
6.5.11	LPTMR_CLOCKS	35
6.5.12	FTM_CLOCKS	35
6.5.13	EDMA_CLOCKS	36
6.5.14	ADC16_CLOCKS	36
6.5.15	CMT_CLOCKS	36
6.5.16	UART_CLOCKS	36
6.5.17	RNGA_CLOCKS	36
6.5.18	CRC_CLOCKS	37
6.5.19	I2C_CLOCKS	37
6.5.20	PDB_CLOCKS	37
6.5.21	CMP_CLOCKS	37

Contents

Section Number	Title	Page Number
6.5.22	FTF_CLOCKS	37
6.5.23	SYS_CLK	38
6.6	Enumeration Type Documentation	38
6.6.1	clock_name_t	38
6.6.2	clock_usb_src_t	38
6.6.3	clock_ip_name_t	38
6.6.4	osc_mode_t	38
6.6.5	_osc_cap_load	39
6.6.6	_oscer_enable_mode	39
6.6.7	mcg_fll_src_t	39
6.6.8	mcg_irc_mode_t	39
6.6.9	mcg_dmx32_t	39
6.6.10	mcg_drs_t	40
6.6.11	mcg_pll_ref_src_t	40
6.6.12	mcg_clkout_src_t	40
6.6.13	mcg_atm_select_t	40
6.6.14	mcg_oscsel_t	40
6.6.15	mcg_pll_clk_select_t	41
6.6.16	mcg_monitor_mode_t	41
6.6.17	_mcg_status	41
6.6.18	_mcg_status_flags_t	41
6.6.19	_mcg_ircclk_enable_mode	41
6.6.20	_mcg_pll_enable_mode	42
6.6.21	mcg_mode_t	42
6.7	Function Documentation	42
6.7.1	CLOCK_EnableClock	42
6.7.2	CLOCK_DisableClock	42
6.7.3	CLOCK_SetEr32kClock	43
6.7.4	CLOCK_SetTraceClock	43
6.7.5	CLOCK_SetPllFllSelClock	43
6.7.6	CLOCK_SetClkOutClock	43
6.7.7	CLOCK_SetRtcClkOutClock	43
6.7.8	CLOCK_EnableUsbfs0Clock	44
6.7.9	CLOCK_DisableUsbfs0Clock	44
6.7.10	CLOCK_SetOutDiv	44
6.7.11	CLOCK_GetFreq	44
6.7.12	CLOCK_GetCoreSysClkFreq	45
6.7.13	CLOCK_GetPlatClkFreq	45
6.7.14	CLOCK_GetBusClkFreq	45
6.7.15	CLOCK_GetFlashClkFreq	45
6.7.16	CLOCK_GetPllFllSelClkFreq	45
6.7.17	CLOCK_GetEr32kClkFreq	46
6.7.18	CLOCK_GetOsc0ErClkFreq	46

Contents

Section Number	Title	Page Number
6.7.19	CLOCK_SetSimConfig	46
6.7.20	CLOCK_SetSimSafeDivs	46
6.7.21	CLOCK_GetOutClkFreq	46
6.7.22	CLOCK_GetFllFreq	47
6.7.23	CLOCK_GetInternalRefClkFreq	47
6.7.24	CLOCK_GetFixedFreqClkFreq	47
6.7.25	CLOCK_GetPll0Freq	47
6.7.26	CLOCK_SetLowPowerEnable	47
6.7.27	CLOCK_SetInternalRefClkConfig	48
6.7.28	CLOCK_SetExternalRefClkConfig	48
6.7.29	CLOCK_SetFllExtRefDiv	49
6.7.30	CLOCK_EnablePll0	49
6.7.31	CLOCK_DisablePll0	49
6.7.32	CLOCK_CalcPllDiv	49
6.7.33	CLOCK_SetOsc0MonitorMode	50
6.7.34	CLOCK_SetRtcOscMonitorMode	50
6.7.35	CLOCK_SetPll0MonitorMode	50
6.7.36	CLOCK_GetStatusFlags	50
6.7.37	CLOCK_ClearStatusFlags	51
6.7.38	OSC_SetExtRefClkConfig	51
6.7.39	OSC_SetCapLoad	52
6.7.40	CLOCK_InitOsc0	52
6.7.41	CLOCK_DeinitOsc0	52
6.7.42	CLOCK_SetXtal0Freq	52
6.7.43	CLOCK_SetXtal32Freq	53
6.7.44	CLOCK_TrimInternalRefClk	53
6.7.45	CLOCK_GetMode	54
6.7.46	CLOCK_SetFeiMode	54
6.7.47	CLOCK_SetFeeMode	54
6.7.48	CLOCK_SetFbiMode	55
6.7.49	CLOCK_SetFbeMode	56
6.7.50	CLOCK_SetBlpiMode	57
6.7.51	CLOCK_SetBlpeMode	57
6.7.52	CLOCK_SetPbeMode	58
6.7.53	CLOCK_SetPeeMode	58
6.7.54	CLOCK_ExternalModeToFbeModeQuick	59
6.7.55	CLOCK_InternalModeToFbiModeQuick	59
6.7.56	CLOCK_BootToFeiMode	60
6.7.57	CLOCK_BootToFeeMode	60
6.7.58	CLOCK_BootToBlpiMode	61
6.7.59	CLOCK_BootToBlpeMode	61
6.7.60	CLOCK_BootToPeeMode	62
6.7.61	CLOCK_SetMcgConfig	62
6.8	Variable Documentation	63

Contents

Section Number	Title	Page Number
6.8.1	g_xtal0Freq	63
6.8.2	g_xtal32Freq	63
6.9	Multipurpose Clock Generator (MCG)	64
6.9.1	Function description	64
6.9.2	Typical use case	66
6.9.3	Code Configuration Option	69
Chapter	CMP: Analog Comparator Driver	
7.1	Overview	71
7.2	Typical use case	71
7.2.1	Polling Configuration	71
7.2.2	Interrupt Configuration	72
7.3	Data Structure Documentation	74
7.3.1	struct cmp_config_t	74
7.3.2	struct cmp_filter_config_t	74
7.3.3	struct cmp_dac_config_t	75
7.4	Macro Definition Documentation	75
7.4.1	FSL_CMP_DRIVER_VERSION	75
7.5	Enumeration Type Documentation	75
7.5.1	_cmp_interrupt_enable	75
7.5.2	_cmp_status_flags	75
7.5.3	cmp_hysteresis_mode_t	76
7.5.4	cmp_reference_voltage_source_t	76
7.6	Function Documentation	76
7.6.1	CMP_Init	76
7.6.2	CMP_Deinit	76
7.6.3	CMP_Enable	78
7.6.4	CMP_GetDefaultConfig	78
7.6.5	CMP_SetInputChannels	78
7.6.6	CMP_SetFilterConfig	79
7.6.7	CMP_SetDACConfig	79
7.6.8	CMP_EnableInterrupts	79
7.6.9	CMP_DisableInterrupts	79
7.6.10	CMP_GetStatusFlags	80
7.6.11	CMP_ClearStatusFlags	80
Chapter	CMT: Carrier Modulator Transmitter Driver	
8.1	Overview	81

Contents

Section Number	Title	Page Number
8.2	Clock formulas	81
8.3	Typical use case	81
8.4	Data Structure Documentation	84
8.4.1	struct cmt_modulate_config_t	84
8.4.2	struct cmt_config_t	85
8.5	Macro Definition Documentation	85
8.5.1	FSL_CMT_DRIVER_VERSION	85
8.6	Enumeration Type Documentation	85
8.6.1	cmt_mode_t	85
8.6.2	cmt_primary_clkdiv_t	86
8.6.3	cmt_second_clkdiv_t	86
8.6.4	cmt_infrared_output_polarity_t	87
8.6.5	cmt_infrared_output_state_t	87
8.6.6	_cmt_interrupt_enable	87
8.7	Function Documentation	87
8.7.1	CMT_GetDefaultConfig	87
8.7.2	CMT_Init	87
8.7.3	CMT_Deinit	88
8.7.4	CMT_SetMode	88
8.7.5	CMT_GetMode	88
8.7.6	CMT_GetCMTFrequency	89
8.7.7	CMT_SetCarrirGenerateCountOne	90
8.7.8	CMT_SetCarrirGenerateCountTwo	90
8.7.9	CMT_SetModulateMarkSpace	91
8.7.10	CMT_EnableExtendedSpace	91
8.7.11	CMT_SetIroState	92
8.7.12	CMT_EnableInterrupts	93
8.7.13	CMT_DisableInterrupts	93
8.7.14	CMT_GetStatusFlags	93
Chapter	CRC: Cyclic Redundancy Check Driver	
9.1	Overview	95
9.2	CRC Driver Initialization and Configuration	95
9.3	CRC Write Data	95
9.4	CRC Get Checksum	95
9.5	Comments about API usage in RTOS	96

Contents

Section Number	Title	Page Number
9.6	Comments about API usage in interrupt handler	96
9.7	CRC Driver Examples	96
9.7.1	Simple examples	96
9.7.2	Advanced examples	97
9.8	Data Structure Documentation	100
9.8.1	struct crc_config_t	100
9.9	Macro Definition Documentation	100
9.9.1	FSL_CRC_DRIVER_VERSION	100
9.9.2	CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT	101
9.10	Enumeration Type Documentation	101
9.10.1	crc_bits_t	101
9.10.2	crc_result_t	101
9.11	Function Documentation	101
9.11.1	CRC_Init	101
9.11.2	CRC_Deinit	101
9.11.3	CRC_GetDefaultConfig	102
9.11.4	CRC_WriteData	102
9.11.5	CRC_Get32bitResult	102
9.11.6	CRC_Get16bitResult	103
Chapter	DMAMUX: Direct Memory Access Multiplexer Driver	
10.1	Overview	105
10.2	Typical use case	105
10.2.1	DMAMUX Operation	105
10.3	Macro Definition Documentation	105
10.3.1	FSL_DMAMUX_DRIVER_VERSION	105
10.4	Function Documentation	106
10.4.1	DMAMUX_Init	106
10.4.2	DMAMUX_Deinit	107
10.4.3	DMAMUX_EnableChannel	107
10.4.4	DMAMUX_DisableChannel	107
10.4.5	DMAMUX_SetSource	108
Chapter	DSPI: Serial Peripheral Interface Driver	
11.1	Overview	109

Contents

Section Number	Title	Page Number
11.2	DSPI Driver	110
11.2.1	Overview	110
11.2.2	Typical use case	110
11.2.3	Data Structure Documentation	117
11.2.4	Macro Definition Documentation	124
11.2.5	Typedef Documentation	125
11.2.6	Enumeration Type Documentation	126
11.2.7	Function Documentation	130
11.3	DSPI DMA Driver	149
11.3.1	Overview	149
11.3.2	Data Structure Documentation	150
11.3.3	Typedef Documentation	153
11.3.4	Function Documentation	154
11.4	DSPI eDMA Driver	159
11.4.1	Overview	159
11.4.2	Data Structure Documentation	160
11.4.3	Typedef Documentation	163
11.4.4	Function Documentation	164
11.5	DSPI FreeRTOS Driver	169
11.5.1	Overview	169
11.5.2	Function Documentation	169
Chapter	eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver	
12.1	Overview	171
12.2	Typical use case	171
12.2.1	eDMA Operation	171
12.3	Data Structure Documentation	177
12.3.1	struct edma_config_t	177
12.3.2	struct edma_transfer_config_t	177
12.3.3	struct edma_channel_Preemption_config_t	178
12.3.4	struct edma_minor_offset_config_t	179
12.3.5	struct edma_tcd_t	179
12.3.6	struct edma_handle_t	180
12.4	Macro Definition Documentation	181
12.4.1	FSL_EDMA_DRIVER_VERSION	181
12.5	Typedef Documentation	181
12.5.1	edma_callback	181

Contents

Section Number	Title	Page Number
12.6	Enumeration Type Documentation	181
12.6.1	edma_transfer_size_t	181
12.6.2	edma_modulo_t	181
12.6.3	edma_bandwidth_t	182
12.6.4	edma_channel_link_type_t	182
12.6.5	_edma_channel_status_flags	182
12.6.6	_edma_error_status_flags	183
12.6.7	edma_interrupt_enable_t	183
12.6.8	edma_transfer_type_t	183
12.6.9	_edma_transfer_status	183
12.7	Function Documentation	184
12.7.1	EDMA_Init	184
12.7.2	EDMA_Deinit	185
12.7.3	EDMA_GetDefaultConfig	185
12.7.4	EDMA_ResetChannel	185
12.7.5	EDMA_SetTransferConfig	186
12.7.6	EDMA_SetMinorOffsetConfig	186
12.7.7	EDMA_SetChannelPreemptionConfig	187
12.7.8	EDMA_SetChannelLink	187
12.7.9	EDMA_SetBandWidth	188
12.7.10	EDMA_SetModulo	188
12.7.11	EDMA_EnableAutoStopRequest	189
12.7.12	EDMA_EnableChannelInterrupts	189
12.7.13	EDMA_DisableChannelInterrupts	189
12.7.14	EDMA_TcdReset	190
12.7.15	EDMA_TcdSetTransferConfig	190
12.7.16	EDMA_TcdSetMinorOffsetConfig	191
12.7.17	EDMA_TcdSetChannelLink	192
12.7.18	EDMA_TcdSetBandWidth	192
12.7.19	EDMA_TcdSetModulo	193
12.7.20	EDMA_TcdEnableAutoStopRequest	193
12.7.21	EDMA_TcdEnableInterrupts	193
12.7.22	EDMA_TcdDisableInterrupts	194
12.7.23	EDMA_EnableChannelRequest	194
12.7.24	EDMA_DisableChannelRequest	194
12.7.25	EDMA_TriggerChannelStart	194
12.7.26	EDMA_GetRemainingMajorLoopCount	195
12.7.27	EDMA_GetErrorStatusFlags	195
12.7.28	EDMA_GetChannelStatusFlags	196
12.7.29	EDMA_ClearChannelStatusFlags	196
12.7.30	EDMA_CreateHandle	196
12.7.31	EDMA_InstallTCDDMemory	197
12.7.32	EDMA_SetCallback	197
12.7.33	EDMA_PrepareTransfer	197

Contents

Section Number	Title	Page Number
12.7.34	EDMA_SubmitTransfer	198
12.7.35	EDMA_StartTransfer	199
12.7.36	EDMA_StopTransfer	199
12.7.37	EDMA_AbortTransfer	199
12.7.38	EDMA_HandleIRQ	199
Chapter	EWM: External Watchdog Monitor Driver	
13.1	Overview	201
13.2	Typical use case	201
13.3	Data Structure Documentation	202
13.3.1	struct ewm_config_t	202
13.4	Macro Definition Documentation	202
13.4.1	FSL_EWM_DRIVER_VERSION	202
13.5	Enumeration Type Documentation	202
13.5.1	_ewm_interrupt_enable_t	202
13.5.2	_ewm_status_flags_t	202
13.6	Function Documentation	203
13.6.1	EWM_Init	203
13.6.2	EWM_Deinit	203
13.6.3	EWM_GetDefaultConfig	203
13.6.4	EWM_EnableInterrupts	204
13.6.5	EWM_DisableInterrupts	204
13.6.6	EWM_GetStatusFlags	204
13.6.7	EWM_Refresh	205
Chapter	C90TFS Flash Driver	
14.1	Overview	207
14.2	Data Structure Documentation	216
14.2.1	struct flash_execute_in_ram_function_config_t	216
14.2.2	struct flash_swap_state_config_t	216
14.2.3	struct flash_swap_ifr_field_config_t	216
14.2.4	union flash_swap_ifr_field_data_t	217
14.2.5	union pflash_protection_status_low_t	217
14.2.6	struct pflash_protection_status_t	218
14.2.7	struct flash_prefetch_speculation_status_t	218
14.2.8	struct flash_protection_config_t	218
14.2.9	struct flash_access_config_t	219
14.2.10	struct flash_operation_config_t	219

Contents

Section Number	Title	Page Number
14.2.11	struct flash_config_t	220
14.3	Macro Definition Documentation	222
14.3.1	MAKE_VERSION	222
14.3.2	FSL_FLASH_DRIVER_VERSION	222
14.3.3	FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT	222
14.3.4	FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT	222
14.3.5	FLASH_DRIVER_IS_FLASH_RESIDENT	222
14.3.6	FLASH_DRIVER_IS_EXPORTED	222
14.3.7	kStatusGroupGeneric	222
14.3.8	MAKE_STATUS	222
14.3.9	FOUR_CHAR_CODE	222
14.4	Enumeration Type Documentation	222
14.4.1	_flash_driver_version_constants	222
14.4.2	_flash_status	223
14.4.3	_flash_driver_api_keys	223
14.4.4	flash_margin_value_t	224
14.4.5	flash_security_state_t	224
14.4.6	flash_protection_state_t	224
14.4.7	flash_execute_only_access_state_t	224
14.4.8	flash_property_tag_t	224
14.4.9	_flash_execute_in_ram_function_constants	225
14.4.10	flash_read_resource_option_t	225
14.4.11	_flash_read_resource_range	225
14.4.12	_k3_flash_read_once_index	226
14.4.13	flash_flexram_function_option_t	226
14.4.14	flash_swap_function_option_t	226
14.4.15	flash_swap_control_option_t	226
14.4.16	flash_swap_state_t	227
14.4.17	flash_swap_block_status_t	227
14.4.18	flash_partition_flexram_load_option_t	227
14.4.19	flash_memory_index_t	227
14.4.20	flash_cache_controller_index_t	227
14.4.21	flash_cache_clear_process_t	228
14.5	Function Documentation	228
14.5.1	FLASH_Init	228
14.5.2	FLASH_SetCallback	228
14.5.3	FLASH_PrepareExecuteInRamFunctions	229
14.5.4	FLASH_EraseAll	229
14.5.5	FLASH_Erase	230
14.5.6	FLASH_EraseAllExecuteOnlySegments	231
14.5.7	FLASH_Program	232
14.5.8	FLASH_ProgramOnce	233

Contents

Section Number	Title	Page Number
14.5.9	FLASH_ReadOnce	234
14.5.10	FLASH_GetSecurityState	236
14.5.11	FLASH_SecurityBypass	236
14.5.12	FLASH_VerifyEraseAll	237
14.5.13	FLASH_VerifyErase	238
14.5.14	FLASH_VerifyProgram	239
14.5.15	FLASH_VerifyEraseAllExecuteOnlySegments	240
14.5.16	FLASH_IsProtected	240
14.5.17	FLASH_IsExecuteOnly	241
14.5.18	FLASH_GetProperty	242
14.5.19	FLASH_SetProperty	242
14.5.20	FLASH_PflashSetProtection	243
14.5.21	FLASH_PflashGetProtection	243
Chapter	FTM: FlexTimer Driver	
15.1	Overview	245
15.2	Function groups	245
15.2.1	Initialization and deinitialization	245
15.2.2	PWM Operations	245
15.2.3	Input capture operations	245
15.2.4	Output compare operations	246
15.2.5	Quad decode	246
15.2.6	Fault operation	246
15.3	Register Update	246
15.4	Typical use case	247
15.4.1	PWM output	247
15.5	Data Structure Documentation	254
15.5.1	struct ftm_chnl_pwm_signal_param_t	254
15.5.2	struct ftm_dual_edge_capture_param_t	254
15.5.3	struct ftm_phase_params_t	255
15.5.4	struct ftm_fault_param_t	255
15.5.5	struct ftm_config_t	255
15.6	Enumeration Type Documentation	256
15.6.1	ftm_chnl_t	256
15.6.2	ftm_fault_input_t	257
15.6.3	ftm_pwm_mode_t	257
15.6.4	ftm_pwm_level_select_t	257
15.6.5	ftm_output_compare_mode_t	257
15.6.6	ftm_input_capture_edge_t	257

Contents

Section Number	Title	Page Number
15.6.7	ftm_dual_edge_capture_mode_t	258
15.6.8	ftm_quad_decode_mode_t	258
15.6.9	ftm_phase_polarity_t	258
15.6.10	ftm_deadtime_prescale_t	258
15.6.11	ftm_clock_source_t	258
15.6.12	ftm_clock_prescale_t	259
15.6.13	ftm_bdm_mode_t	259
15.6.14	ftm_fault_mode_t	259
15.6.15	ftm_external_trigger_t	259
15.6.16	ftm_pwm_sync_method_t	260
15.6.17	ftm_reload_point_t	260
15.6.18	ftm_interrupt_enable_t	261
15.6.19	ftm_status_flags_t	261
15.6.20	_ftm_quad_decoder_flags	262
15.7	Function Documentation	262
15.7.1	FTM_Init	262
15.7.2	FTM_Deinit	262
15.7.3	FTM_GetDefaultConfig	262
15.7.4	FTM_SetupPwm	263
15.7.5	FTM_UpdatePwmDutycycle	263
15.7.6	FTM_UpdateChnlEdgeLevelSelect	264
15.7.7	FTM_SetupInputCapture	264
15.7.8	FTM_SetupOutputCompare	265
15.7.9	FTM_SetupDualEdgeCapture	265
15.7.10	FTM_SetupFault	265
15.7.11	FTM_EnableInterrupts	266
15.7.12	FTM_DisableInterrupts	266
15.7.13	FTM_GetEnabledInterrupts	266
15.7.14	FTM_GetStatusFlags	266
15.7.15	FTM_ClearStatusFlags	267
15.7.16	FTM_SetTimerPeriod	267
15.7.17	FTM_GetCurrentTimerCount	267
15.7.18	FTM_StartTimer	268
15.7.19	FTM_StopTimer	268
15.7.20	FTM_SetSoftwareCtrlEnable	268
15.7.21	FTM_SetSoftwareCtrlVal	269
15.7.22	FTM_SetGlobalTimeBaseOutputEnable	269
15.7.23	FTM_SetOutputMask	269
15.7.24	FTM_SetFaultControlEnable	269
15.7.25	FTM_SetDeadTimeEnable	270
15.7.26	FTM_SetComplementaryEnable	270
15.7.27	FTM_SetInvertEnable	270
15.7.28	FTM_SetupQuadDecode	271
15.7.29	FTM_GetQuadDecoderFlags	271

Contents

Section Number	Title	Page Number
15.7.30	FTM_SetQuadDecoderModuloValue	271
15.7.31	FTM_GetQuadDecoderCounterValue	272
15.7.32	FTM_ClearQuadDecoderCounterValue	272
15.7.33	FTM_SetSoftwareTrigger	272
15.7.34	FTM_SetWriteProtection	272
Chapter	GPIO: General-Purpose Input/Output Driver	
16.1	Overview	275
16.2	Data Structure Documentation	275
16.2.1	struct gpio_pin_config_t	275
16.3	Macro Definition Documentation	276
16.3.1	FSL_GPIO_DRIVER_VERSION	276
16.4	Enumeration Type Documentation	276
16.4.1	gpio_pin_direction_t	276
16.5	GPIO Driver	277
16.5.1	Overview	277
16.5.2	Typical use case	277
16.5.3	Function Documentation	278
16.6	FGPIO Driver	281
16.6.1	Typical use case	281
Chapter	I2C: Inter-Integrated Circuit Driver	
17.1	Overview	283
17.2	I2C Driver	284
17.2.1	Overview	284
17.2.2	Typical use case	284
17.2.3	Data Structure Documentation	291
17.2.4	Macro Definition Documentation	296
17.2.5	Typedef Documentation	296
17.2.6	Enumeration Type Documentation	296
17.2.7	Function Documentation	298
17.3	I2C eDMA Driver	312
17.3.1	Overview	312
17.3.2	Data Structure Documentation	312
17.3.3	Typedef Documentation	313
17.3.4	Function Documentation	313

Contents

Section Number	Title	Page Number
17.4	I2C DMA Driver	316
17.4.1	Overview	316
17.4.2	Data Structure Documentation	316
17.4.3	Typedef Documentation	317
17.4.4	Function Documentation	317
17.5	I2C FreeRTOS Driver	319
17.5.1	Overview	319
17.5.2	Function Documentation	319
Chapter	LLWU: Low-Leakage Wakeup Unit Driver	
18.1	Overview	321
18.2	External wakeup pins configurations	321
18.3	Internal wakeup modules configurations	321
18.4	Digital pin filter for external wakeup pin configurations	321
18.5	Data Structure Documentation	322
18.5.1	struct llwu_external_pin_filter_mode_t	322
18.6	Macro Definition Documentation	322
18.6.1	FSL_LLWU_DRIVER_VERSION	322
18.7	Enumeration Type Documentation	323
18.7.1	llwu_external_pin_mode_t	323
18.7.2	llwu_pin_filter_mode_t	323
18.8	Function Documentation	323
18.8.1	LLWU_SetExternalWakeupPinMode	323
18.8.2	LLWU_GetExternalWakeupPinFlag	323
18.8.3	LLWU_ClearExternalWakeupPinFlag	324
18.8.4	LLWU_EnableInternalModuleInterruptWakup	324
18.8.5	LLWU_GetInternalWakeupModuleFlag	324
18.8.6	LLWU_SetPinFilterMode	325
18.8.7	LLWU_GetPinFilterFlag	325
18.8.8	LLWU_ClearPinFilterFlag	325
18.8.9	LLWU_SetResetPinMode	326
Chapter	LPTMR: Low-Power Timer	
19.1	Overview	327
19.2	Function groups	327

Contents

Section Number	Title	Page Number
19.2.1	Initialization and deinitialization	327
19.2.2	Timer period Operations	327
19.2.3	Start and Stop timer operations	327
19.2.4	Status	328
19.2.5	Interrupt	328
19.3	Typical use case	328
19.3.1	LPTMR tick example	328
19.4	Data Structure Documentation	330
19.4.1	struct lptmr_config_t	330
19.5	Enumeration Type Documentation	331
19.5.1	lptmr_pin_select_t	331
19.5.2	lptmr_pin_polarity_t	331
19.5.3	lptmr_timer_mode_t	331
19.5.4	lptmr_prescaler_glitch_value_t	332
19.5.5	lptmr_prescaler_clock_select_t	332
19.5.6	lptmr_interrupt_enable_t	332
19.5.7	lptmr_status_flags_t	333
19.6	Function Documentation	333
19.6.1	LPTMR_Init	333
19.6.2	LPTMR_Deinit	333
19.6.3	LPTMR_GetDefaultConfig	333
19.6.4	LPTMR_EnableInterrupts	334
19.6.5	LPTMR_DisableInterrupts	334
19.6.6	LPTMR_GetEnabledInterrupts	334
19.6.7	LPTMR_GetStatusFlags	334
19.6.8	LPTMR_ClearStatusFlags	335
19.6.9	LPTMR_SetTimerPeriod	335
19.6.10	LPTMR_GetCurrentTimerCount	335
19.6.11	LPTMR_StartTimer	336
19.6.12	LPTMR_StopTimer	336
Chapter	PDB: Programmable Delay Block	
20.1	Overview	337
20.2	Typical use case	337
20.2.1	Working as basic PDB counter with a PDB interrupt.	337
20.2.2	Working with an additional trigger. The ADC trigger is used as an example.	338
20.3	Data Structure Documentation	342
20.3.1	struct pdb_config_t	342
20.3.2	struct pdb_adc_pretrigger_config_t	342

Contents

Section Number	Title	Page Number
20.3.3	struct pdb_dac_trigger_config_t	343
20.4	Macro Definition Documentation	343
20.4.1	FSL_PDB_DRIVER_VERSION	343
20.5	Enumeration Type Documentation	343
20.5.1	_pdb_status_flags	343
20.5.2	_pdb_adc_pretrigger_flags	344
20.5.3	_pdb_interrupt_enable	344
20.5.4	pdb_load_value_mode_t	344
20.5.5	pdb_prescaler_divider_t	344
20.5.6	pdb_divider_multiplication_factor_t	345
20.5.7	pdb_trigger_input_source_t	345
20.6	Function Documentation	346
20.6.1	PDB_Init	346
20.6.2	PDB_Deinit	346
20.6.3	PDB_GetDefaultConfig	346
20.6.4	PDB_Enable	346
20.6.5	PDB_DoSoftwareTrigger	347
20.6.6	PDB_DoLoadValues	347
20.6.7	PDB_EnableDMA	347
20.6.8	PDB_EnableInterrupts	347
20.6.9	PDB_DisableInterrupts	348
20.6.10	PDB_GetStatusFlags	348
20.6.11	PDB_ClearStatusFlags	348
20.6.12	PDB_SetModulusValue	348
20.6.13	PDB_GetCounterValue	349
20.6.14	PDB_SetCounterDelayValue	349
20.6.15	PDB_SetADCPreTriggerConfig	349
20.6.16	PDB_SetADCPreTriggerDelayValue	350
20.6.17	PDB_GetADCPreTriggerStatusFlags	351
20.6.18	PDB_ClearADCPreTriggerStatusFlags	351
20.6.19	PDB_EnablePulseOutTrigger	351
20.6.20	PDB_SetPulseOutTriggerDelayValue	352
Chapter	PIT: Periodic Interrupt Timer	
21.1	Overview	353
21.2	Function groups	353
21.2.1	Initialization and deinitialization	353
21.2.2	Timer period Operations	353
21.2.3	Start and Stop timer operations	353
21.2.4	Status	354

Contents

Section Number	Title	Page Number
21.2.5	Interrupt	354
21.3	Typical use case	354
21.3.1	PIT tick example	354
21.4	Data Structure Documentation	356
21.4.1	struct pit_config_t	356
21.5	Enumeration Type Documentation	356
21.5.1	pit_chnl_t	356
21.5.2	pit_interrupt_enable_t	357
21.5.3	pit_status_flags_t	357
21.6	Function Documentation	357
21.6.1	PIT_Init	357
21.6.2	PIT_Deinit	357
21.6.3	PIT_GetDefaultConfig	357
21.6.4	PIT_EnableInterrupts	358
21.6.5	PIT_DisableInterrupts	358
21.6.6	PIT_GetEnabledInterrupts	358
21.6.7	PIT_GetStatusFlags	359
21.6.8	PIT_ClearStatusFlags	360
21.6.9	PIT_SetTimerPeriod	360
21.6.10	PIT_GetCurrentTimerCount	361
21.6.11	PIT_StartTimer	361
21.6.12	PIT_StopTimer	361
Chapter	PMC: Power Management Controller	
22.1	Overview	363
22.2	Data Structure Documentation	363
22.2.1	struct pmc_low_volt_detect_config_t	363
22.2.2	struct pmc_low_volt_warning_config_t	364
22.3	Macro Definition Documentation	364
22.3.1	FSL_PMC_DRIVER_VERSION	364
22.4	Function Documentation	364
22.4.1	PMC_ConfigureLowVoltDetect	364
22.4.2	PMC_GetLowVoltDetectFlag	364
22.4.3	PMC_ClearLowVoltDetectFlag	365
22.4.4	PMC_ConfigureLowVoltWarning	365
22.4.5	PMC_GetLowVoltWarningFlag	365
22.4.6	PMC_ClearLowVoltWarningFlag	366

Contents

Section Number	Title	Page Number
Chapter	PORT: Port Control and Interrupts	
23.1	Overview	369
23.2	Typical configuration use case	369
23.2.1	Input PORT configuration	369
23.2.2	I2C PORT Configuration	369
23.3	Macro Definition Documentation	370
23.3.1	FSL_PORT_DRIVER_VERSION	370
23.4	Enumeration Type Documentation	370
23.4.1	port_interrupt_t	370
Chapter	RCM: Reset Control Module Driver	
24.1	Overview	371
24.2	Data Structure Documentation	372
24.2.1	struct rcm_reset_pin_filter_config_t	372
24.3	Macro Definition Documentation	372
24.3.1	FSL_RCM_DRIVER_VERSION	372
24.4	Enumeration Type Documentation	372
24.4.1	rcm_reset_source_t	372
24.4.2	rcm_run_wait_filter_mode_t	372
24.5	Function Documentation	373
24.5.1	RCM_GetPreviousResetSources	373
24.5.2	RCM_ConfigureResetPinFilter	373
Chapter	RNGA: Random Number Generator Accelerator Driver	
25.1	Overview	375
25.2	RNGA Initialization	375
25.3	Get random data from RNGA	375
25.4	RNGA Set/Get Working Mode	375
25.5	Seed RNGA	375
25.6	Macro Definition Documentation	377
25.6.1	FSL_RNGA_DRIVER_VERSION	377

Contents

Section Number	Title	Page Number
25.7	Enumeration Type Documentation	377
25.7.1	rnga_mode_t	377
25.8	Function Documentation	377
25.8.1	RNGA_Init	377
25.8.2	RNGA_Deinit	377
25.8.3	RNGA_GetRandomData	378
25.8.4	RNGA_Seed	378
25.8.5	RNGA_SetMode	378
25.8.6	RNGA_GetMode	379
Chapter	RTC: Real Time Clock	
26.1	Overview	381
26.2	Function groups	381
26.2.1	Initialization and deinitialization	381
26.2.2	Set & Get Datetime	381
26.2.3	Set & Get Alarm	381
26.2.4	Start & Stop timer	382
26.2.5	Status	382
26.2.6	Interrupt	382
26.2.7	RTC Oscillator	382
26.2.8	Monotonic Counter	382
26.3	Typical use case	382
26.3.1	RTC tick example	382
26.4	Data Structure Documentation	385
26.4.1	struct rtc_datetime_t	385
26.4.2	struct rtc_config_t	386
26.5	Enumeration Type Documentation	386
26.5.1	rtc_interrupt_enable_t	386
26.5.2	rtc_status_flags_t	387
26.6	Function Documentation	387
26.6.1	RTC_Init	387
26.6.2	RTC_Deinit	387
26.6.3	RTC_GetDefaultConfig	387
26.6.4	RTC_SetDatetime	388
26.6.5	RTC_GetDatetime	388
26.6.6	RTC_SetAlarm	388
26.6.7	RTC_GetAlarm	389
26.6.8	RTC_EnableInterrupts	389
26.6.9	RTC_DisableInterrupts	389

Contents

Section Number	Title	Page Number
26.6.10	RTC_GetEnabledInterrupts	389
26.6.11	RTC_GetStatusFlags	390
26.6.12	RTC_ClearStatusFlags	390
26.6.13	RTC_StartTimer	390
26.6.14	RTC_StopTimer	391
26.6.15	RTC_Reset	391
Chapter	SAI: Serial Audio Interface	
27.1	Overview	393
27.2	Typical use case	393
27.2.1	SAI Send/receive using an interrupt method	393
27.2.2	SAI Send/receive using a DMA method	394
27.3	Data Structure Documentation	399
27.3.1	struct sai_config_t	399
27.3.2	struct sai_transfer_format_t	400
27.3.3	struct sai_transfer_t	400
27.3.4	struct _sai_handle	401
27.4	Macro Definition Documentation	401
27.4.1	SAI_XFER_QUEUE_SIZE	401
27.5	Enumeration Type Documentation	401
27.5.1	_sai_status_t	401
27.5.2	sai_protocol_t	402
27.5.3	sai_master_slave_t	402
27.5.4	sai_mono_stereo_t	402
27.5.5	sai_sync_mode_t	402
27.5.6	sai_mclk_source_t	402
27.5.7	sai_bclk_source_t	403
27.5.8	_sai_interrupt_enable_t	403
27.5.9	_sai_dma_enable_t	403
27.5.10	_sai_flags	403
27.5.11	sai_reset_type_t	403
27.5.12	sai_sample_rate_t	404
27.5.13	sai_word_width_t	404
27.6	Function Documentation	404
27.6.1	SAI_TxInit	404
27.6.2	SAI_RxInit	405
27.6.3	SAI_TxGetDefaultConfig	405
27.6.4	SAI_RxGetDefaultConfig	405
27.6.5	SAI_Deinit	406

Contents

Section Number	Title	Page Number
27.6.6	SAI_TxReset	406
27.6.7	SAI_RxReset	406
27.6.8	SAI_TxEnable	406
27.6.9	SAI_RxEnable	407
27.6.10	SAI_TxGetStatusFlag	407
27.6.11	SAI_TxClearStatusFlags	407
27.6.12	SAI_RxGetStatusFlag	407
27.6.13	SAI_RxClearStatusFlags	408
27.6.14	SAI_TxEnableInterrupts	408
27.6.15	SAI_RxEnableInterrupts	408
27.6.16	SAI_TxDisableInterrupts	409
27.6.17	SAI_RxDisableInterrupts	410
27.6.18	SAI_TxEnableDMA	410
27.6.19	SAI_RxEnableDMA	410
27.6.20	SAI_TxGetDataRegisterAddress	411
27.6.21	SAI_RxGetDataRegisterAddress	412
27.6.22	SAI_TxSetFormat	412
27.6.23	SAI_RxSetFormat	413
27.6.24	SAI_WriteBlocking	413
27.6.25	SAI_WriteData	413
27.6.26	SAI_ReadBlocking	414
27.6.27	SAI_ReadData	414
27.6.28	SAI_TransferTxCreateHandle	414
27.6.29	SAI_TransferRxCreateHandle	415
27.6.30	SAI_TransferTxSetFormat	415
27.6.31	SAI_TransferRxSetFormat	416
27.6.32	SAI_TransferSendNonBlocking	416
27.6.33	SAI_TransferReceiveNonBlocking	417
27.6.34	SAI_TransferGetSendCount	417
27.6.35	SAI_TransferGetReceiveCount	418
27.6.36	SAI_TransferAbortSend	418
27.6.37	SAI_TransferAbortReceive	419
27.6.38	SAI_TransferTxHandleIRQ	419
27.6.39	SAI_TransferRxHandleIRQ	419
27.7	SAI DMA Driver	420
27.7.1	Overview	420
27.7.2	Data Structure Documentation	421
27.7.3	Function Documentation	421
27.8	SAI eDMA Driver	427
27.8.1	Overview	427
27.8.2	Data Structure Documentation	428
27.8.3	Function Documentation	429

Contents

Section Number	Title	Page Number
Chapter	SIM: System Integration Module Driver	
28.1	Overview	435
28.2	Data Structure Documentation	435
28.2.1	struct sim_uid_t	435
28.3	Enumeration Type Documentation	436
28.3.1	_sim_flash_mode	436
28.4	Function Documentation	436
28.4.1	SIM_GetUniqueId	436
28.4.2	SIM_SetFlashMode	436
Chapter	SMC: System Mode Controller Driver	
29.1	Overview	437
29.2	Typical use case	437
29.2.1	Enter wait or stop modes	437
29.3	Macro Definition Documentation	439
29.3.1	FSL_SMC_DRIVER_VERSION	439
29.4	Enumeration Type Documentation	439
29.4.1	smc_power_mode_protection_t	439
29.4.2	smc_power_state_t	439
29.4.3	smc_run_mode_t	439
29.4.4	smc_stop_mode_t	439
29.4.5	smc_partial_stop_option_t	440
29.4.6	_smc_status	440
29.5	Function Documentation	440
29.5.1	SMC_SetPowerModeProtection	440
29.5.2	SMC_GetPowerModeState	440
29.5.3	SMC_PreEnterStopModes	441
29.5.4	SMC_PostExitStopModes	441
29.5.5	SMC_PreEnterWaitModes	441
29.5.6	SMC_PostExitWaitModes	441
29.5.7	SMC_SetPowerModeRun	441
29.5.8	SMC_SetPowerModeWait	441
29.5.9	SMC_SetPowerModeStop	442
29.5.10	SMC_SetPowerModeVlpr	442
29.5.11	SMC_SetPowerModeVlps	442
29.5.12	SMC_SetPowerModeVlps	443

Contents

Section Number	Title	Page Number
Chapter	UART: Universal Asynchronous Receiver/Transmitter Driver	
30.1	Overview	445
30.2	UART Driver	446
30.2.1	Overview	446
30.2.2	Typical use case	446
30.2.3	Data Structure Documentation	454
30.2.4	Macro Definition Documentation	456
30.2.5	Typedef Documentation	456
30.2.6	Enumeration Type Documentation	456
30.2.7	Function Documentation	458
30.3	UART DMA Driver	470
30.3.1	Overview	470
30.3.2	Data Structure Documentation	470
30.3.3	Typedef Documentation	471
30.3.4	Function Documentation	471
30.4	UART eDMA Driver	475
30.4.1	Overview	475
30.4.2	Data Structure Documentation	475
30.4.3	Typedef Documentation	476
30.4.4	Function Documentation	476
30.5	UART FreeRTOS Driver	480
30.5.1	Overview	480
30.5.2	Data Structure Documentation	480
30.5.3	Function Documentation	481
Chapter	Debug Console	
31.1	Overview	485
31.2	Function groups	485
31.2.1	Initialization	485
31.2.2	Advanced Feature	486
31.3	Typical use case	489
31.4	Semihosting	491
31.4.1	Guide Semihosting for IAR	491
31.4.2	Guide Semihosting for Keil μ Vision	491
31.4.3	Guide Semihosting for KDS	493
31.4.4	Guide Semihosting for ATL	493
31.4.5	Guide Semihosting for ARMGCC	494

Contents

Section Number	Title	Page Number
Chapter	Notification Framework	
32.1	Overview	497
32.2	Notifier Overview	497
32.3	Data Structure Documentation	499
32.3.1	struct notifier_notification_block_t	499
32.3.2	struct notifier_callback_config_t	500
32.3.3	struct notifier_handle_t	500
32.4	Typedef Documentation	501
32.4.1	notifier_user_config_t	501
32.4.2	notifier_user_function_t	501
32.4.3	notifier_callback_t	502
32.5	Enumeration Type Documentation	502
32.5.1	_notifier_status	502
32.5.2	notifier_policy_t	503
32.5.3	notifier_notification_type_t	503
32.5.4	notifier_callback_type_t	503
32.6	Function Documentation	504
32.6.1	NOTIFIER_CreateHandle	504
32.6.2	NOTIFIER_SwitchConfig	505
32.6.3	NOTIFIER_GetErrorCallbackIndex	506
Chapter	Shell	
33.1	Overview	507
33.2	Function groups	507
33.2.1	Initialization	507
33.2.2	Advanced Feature	507
33.2.3	Shell Operation	508
33.3	Data Structure Documentation	509
33.3.1	struct shell_context_struct	509
33.3.2	struct shell_command_context_t	510
33.3.3	struct shell_command_context_list_t	510
33.4	Macro Definition Documentation	511
33.4.1	SHELL_USE_HISTORY	511
33.4.2	SHELL_SEARCH_IN_HIST	511
33.4.3	SHELL_USE_FILE_STREAM	511
33.4.4	SHELL_AUTO_COMPLETE	511

Contents

Section Number	Title	Page Number
33.4.5	SHELL_BUFFER_SIZE	511
33.4.6	SHELL_MAX_ARGS	511
33.4.7	SHELL_HIST_MAX	511
33.4.8	SHELL_MAX_CMD	511
33.5	Typedef Documentation	511
33.5.1	send_data_cb_t	511
33.5.2	recv_data_cb_t	511
33.5.3	printf_data_t	511
33.5.4	cmd_function_t	511
33.6	Enumeration Type Documentation	511
33.6.1	fun_key_status_t	511
33.7	Function Documentation	512
33.7.1	SHELL_Init	512
33.7.2	SHELL_RegisterCommand	512
33.7.3	SHELL_Main	512
Chapter	DMA Manager	
34.1	Overview	515
34.2	Function groups	515
34.2.1	DMAMGR Initialization and De-initialization	515
34.2.2	DMAMGR Operation	515
34.3	Typical use case	515
34.3.1	DMAMGR static channel allocattion	515
34.3.2	DMAMGR dynamic channel allocation	515
34.4	Data Structure Documentation	516
34.4.1	struct dmamanager_handle_t	516
34.5	Macro Definition Documentation	517
34.5.1	DMAMGR_DYNAMIC_ALLOCATE	517
34.6	Enumeration Type Documentation	517
34.6.1	_dma_manager_status	517
34.7	Function Documentation	517
34.7.1	DMAMGR_Init	517
34.7.2	DMAMGR_Deinit	518
34.7.3	DMAMGR_RequestChannel	518
34.7.4	DMAMGR_ReleaseChannel	519
34.7.5	DMAMGR_IsChannelOccupied	520

Contents

Section Number	Title	Page Number
Chapter	Memory-Mapped Cryptographic Acceleration Unit (MMCAU)	
35.1	Overview	521
35.2	Purpose	521
35.3	Library Features	521
35.4	CAU and mmCAU software library overview	521
35.5	mmCAU software library usage	522
35.6	Function Documentation	524
35.6.1	cau_aes_set_key	524
35.6.2	cau_aes_encrypt	525
35.6.3	cau_aes_decrypt	525
35.6.4	cau_des_chk_parity	526
35.6.5	cau_des_encrypt	526
35.6.6	cau_des_decrypt	527
35.6.7	cau_md5_initialize_output	527
35.6.8	cau_md5_hash_n	528
35.6.9	cau_md5_update	529
35.6.10	cau_md5_hash	529
35.6.11	cau_sha1_initialize_output	530
35.6.12	cau_sha1_hash_n	530
35.6.13	cau_sha1_update	531
35.6.14	cau_sha1_hash	532
35.6.15	cau_sha256_initialize_output	532
35.6.16	cau_sha256_hash_n	533
35.6.17	cau_sha256_update	533
35.6.18	cau_sha256_hash	534
35.6.19	MMCAU_AES_SetKey	534
35.6.20	MMCAU_AES_EncryptEcb	535
35.6.21	MMCAU_AES_DecryptEcb	536
35.6.22	MMCAU_DES_ChkParity	536
35.6.23	MMCAU_DES_EncryptEcb	537
35.6.24	MMCAU_DES_DecryptEcb	537
35.6.25	MMCAU_MD5_InitializeOutput	538
35.6.26	MMCAU_MD5_HashN	538
35.6.27	MMCAU_MD5_Update	539
35.6.28	MMCAU_SHA1_InitializeOutput	540
35.6.29	MMCAU_SHA1_HashN	540
35.6.30	MMCAU_SHA1_Update	541
35.6.31	MMCAU_SHA256_InitializeOutput	542
35.6.32	MMCAU_SHA256_HashN	542
35.6.33	MMCAU_SHA256_Update	543

Contents

Section Number Chapter	Title	Page Number
	Secure Digital Card/Embedded MultiMedia Card (CARD)	
36.1	Overview	545
36.2	Data Structure Documentation	550
36.2.1	struct sd_card_t	550
36.2.2	struct sdio_card_t	551
36.2.3	struct mmc_card_t	552
36.2.4	struct mmc_boot_config_t	553
36.3	Macro Definition Documentation	553
36.3.1	FSL_SDMMC_DRIVER_VERSION	553
36.4	Enumeration Type Documentation	554
36.4.1	_sdmmc_status	554
36.4.2	_sd_card_flag	555
36.4.3	_mmc_card_flag	555
36.4.4	card_operation_voltage_t	555
36.4.5	_host_endian_mode	556
36.5	Function Documentation	556
36.5.1	SD_Init	556
36.5.2	SD_Deinit	557
36.5.3	SD_CheckReadOnly	557
36.5.4	SD_ReadBlocks	557
36.5.5	SD_WriteBlocks	558
36.5.6	SD_EraseBlocks	559
36.5.7	MMC_Init	560
36.5.8	MMC_Deinit	561
36.5.9	MMC_CheckReadOnly	561
36.5.10	MMC_ReadBlocks	561
36.5.11	MMC_WriteBlocks	562
36.5.12	MMC_EraseGroups	563
36.5.13	MMC_SelectPartition	563
36.5.14	MMC_SetBootConfig	564
36.5.15	SDIO_CardInActive	564
36.5.16	SDIO_IO_Write_Direct	565
36.5.17	SDIO_IO_Read_Direct	565
36.5.18	SDIO_IO_Write_Extended	566
36.5.19	SDIO_IO_Read_Extended	566
36.5.20	SDIO_GetCardCapability	567
36.5.21	SDIO_SetBlockSize	567
36.5.22	SDIO_CardReset	568
36.5.23	SDIO_SetDataBusWidth	568
36.5.24	SDIO_SwitchToHighSpeed	569

Contents

Section Number	Title	Page Number
36.5.25	SDIO_ReadCIS	569
36.5.26	SDIO_Init	570
36.5.27	SDIO_EnableIOInterrupt	571
36.5.28	SDIO_EnableIO	571
36.5.29	SDIO_SelectIO	571
36.5.30	SDIO_AbortIO	572
36.5.31	SDIO_DeInit	572
36.5.32	HOST_NotSupport	572
36.5.33	CardInsertDetect	573
36.5.34	HOST_Init	573
36.5.35	HOST_Deinit	573

Chapter SPI based Secure Digital Card (SDSPI)

37.1	Overview	575
37.2	Data Structure Documentation	577
37.2.1	struct sdspi_command_t	577
37.2.2	struct sdspi_host_t	577
37.2.3	struct sdspi_card_t	577
37.3	Enumeration Type Documentation	578
37.3.1	_sdspi_status	578
37.3.2	_sdspi_card_flag	579
37.3.3	sdspi_response_type_t	579
37.4	Function Documentation	579
37.4.1	SDSPI_Init	579
37.4.2	SDSPI_Deinit	580
37.4.3	SDSPI_CheckReadOnly	580
37.4.4	SDSPI_ReadBlocks	581
37.4.5	SDSPI_WriteBlocks	581

Chapter 1

Introduction

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

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

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

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

- IAR Embedded Workbench
- Keil MDK
- MCUXpresso IDE

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

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

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

Table 2: MCUXpresso SDK Folder Structure

Chapter 2

Driver errors status

- `kStatus_DSPI_Error` = 601
- `kStatus_EDMA_QueueFull` = 5100
- `kStatus_EDMA_Busy` = 5101
- `kStatus_SAI_TxBusy` = 1900
- `kStatus_SAI_RxBusy` = 1901
- `kStatus_SAI_TxError` = 1902
- `kStatus_SAI_RxError` = 1903
- `kStatus_SAI_QueueFull` = 1904
- `kStatus_SAI_TxIdle` = 1905
- `kStatus_SAI_RxIdle` = 1906
- `kStatus_SMC_StopAbort` = 3900
- `kStatus_NOTIFIER_ErrorNotificationBefore` = 9800
- `kStatus_NOTIFIER_ErrorNotificationAfter` = 9801
- `kStatus_DMAMGR_ChannelOccupied` = 5200
- `kStatus_DMAMGR_ChannelNotUsed` = 5201
- `kStatus_DMAMGR_NoFreeChannel` = 5202



Chapter 3

Architectural Overview

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

Overview

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

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

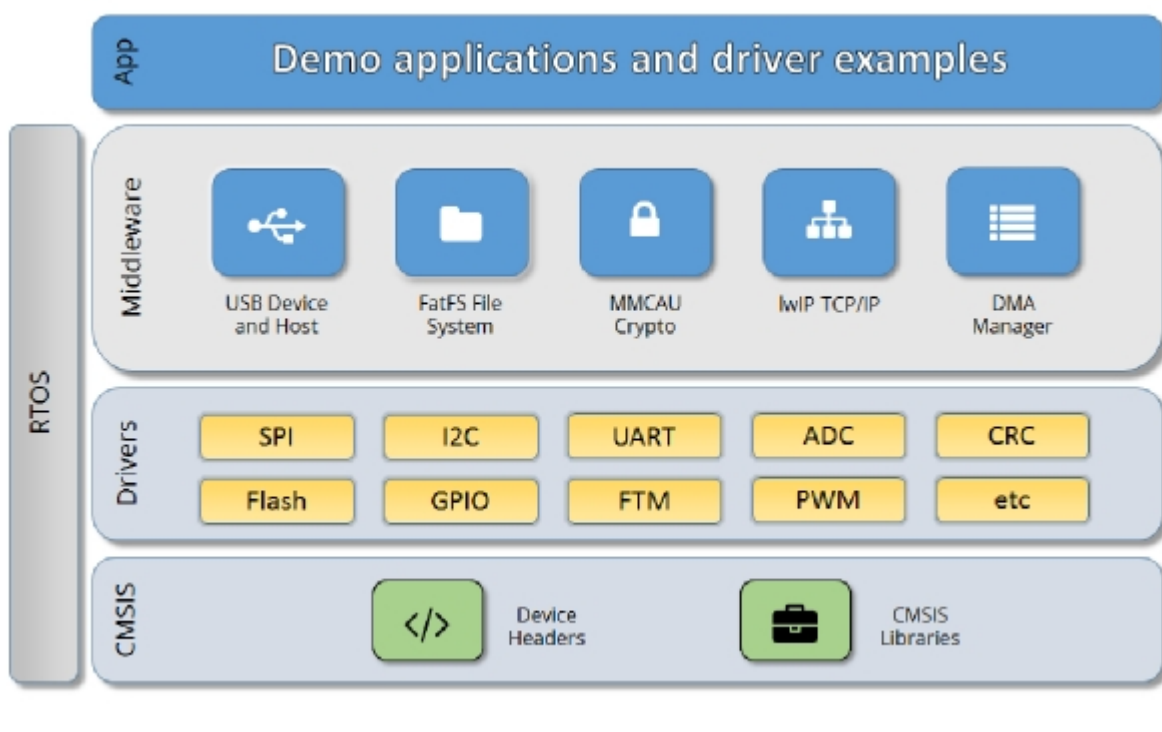


Figure 1: MCUXpresso SDK Block Diagram

MCU header files

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

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

CMSIS Support

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

MCUXpresso SDK Peripheral Drivers

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

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

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

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

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

Interrupt handling for transactional APIs

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

```
PUBWEAK SPI0_IRQHandler
PUBWEAK SPI0_DriverIRQHandler
SPI0_IRQHandler
```

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

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

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

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

Feature Header Files

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

Application

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



Chapter 4

Trademarks

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

How to Reach Us:

Home Page: nxp.com

Web Support: nxp.com/support

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

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

© 2017 NXP B.V.



Chapter 5

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

5.1 Overview

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

5.2 Typical use case

5.2.1 Polling Configuration

```
adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

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

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

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

5.2.2 Interrupt Configuration

```
volatile bool g_Adc16ConversionDoneFlag = false;
volatile uint32_t g_Adc16ConversionValue;
volatile uint32_t g_Adc16InterruptCount = 0U;
```

Typical use case

```
// ...

adc16_config_t adc16ConfigStruct;
adc16_channel_config_t adc16ChannelConfigStruct;

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

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

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

// ...

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

Data Structures

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

Enumerations

- enum [_adc16_channel_status_flags](#) { [kADC16_ChannelConversionDoneFlag](#) = ADC_SC1_COCO_MASK }

- *Channel status flags.*
- enum `_adc16_status_flags` { `kADC16_ActiveFlag` = `ADC_SC2_ADACT_MASK` }
- *Converter status flags.*
- enum `adc16_clock_divider_t` {
`kADC16_ClockDivider1` = 0U,
`kADC16_ClockDivider2` = 1U,
`kADC16_ClockDivider4` = 2U,
`kADC16_ClockDivider8` = 3U }
- *Clock divider for the converter.*
- enum `adc16_resolution_t` {
`kADC16_Resolution8or9Bit` = 0U,
`kADC16_Resolution12or13Bit` = 1U,
`kADC16_Resolution10or11Bit` = 2U,
`kADC16_ResolutionSE8Bit` = `kADC16_Resolution8or9Bit`,
`kADC16_ResolutionSE12Bit` = `kADC16_Resolution12or13Bit`,
`kADC16_ResolutionSE10Bit` = `kADC16_Resolution10or11Bit` }
- *Converter's resolution.*
- enum `adc16_clock_source_t` {
`kADC16_ClockSourceAlt0` = 0U,
`kADC16_ClockSourceAlt1` = 1U,
`kADC16_ClockSourceAlt2` = 2U,
`kADC16_ClockSourceAlt3` = 3U,
`kADC16_ClockSourceAsynchronousClock` = `kADC16_ClockSourceAlt3` }
- *Clock source.*
- enum `adc16_long_sample_mode_t` {
`kADC16_LongSampleCycle24` = 0U,
`kADC16_LongSampleCycle16` = 1U,
`kADC16_LongSampleCycle10` = 2U,
`kADC16_LongSampleCycle6` = 3U,
`kADC16_LongSampleDisabled` = 4U }
- *Long sample mode.*
- enum `adc16_reference_voltage_source_t` {
`kADC16_ReferenceVoltageSourceVref` = 0U,
`kADC16_ReferenceVoltageSourceValt` = 1U }
- *Reference voltage source.*
- enum `adc16_hardware_compare_mode_t` {
`kADC16_HardwareCompareMode0` = 0U,
`kADC16_HardwareCompareMode1` = 1U,
`kADC16_HardwareCompareMode2` = 2U,
`kADC16_HardwareCompareMode3` = 3U }
- *Hardware compare mode.*

Driver version

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

Initialization

- void [ADC16_Init](#) (ADC_Type *base, const [adc16_config_t](#) *config)
Initializes the ADC16 module.
- void [ADC16_Deinit](#) (ADC_Type *base)
De-initializes the ADC16 module.
- void [ADC16_GetDefaultConfig](#) ([adc16_config_t](#) *config)
Gets an available pre-defined settings for the converter's configuration.

Advanced Features

- static void [ADC16_EnableHardwareTrigger](#) (ADC_Type *base, bool enable)
Enables the hardware trigger mode.
- void [ADC16_SetHardwareCompareConfig](#) (ADC_Type *base, const [adc16_hardware_compare_config_t](#) *config)
Configures the hardware compare mode.
- uint32_t [ADC16_GetStatusFlags](#) (ADC_Type *base)
Gets the status flags of the converter.
- void [ADC16_ClearStatusFlags](#) (ADC_Type *base, uint32_t mask)
Clears the status flags of the converter.

Conversion Channel

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

5.3 Data Structure Documentation

5.3.1 struct [adc16_config_t](#)

Data Fields

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

- *Enable the high-speed mode.*
bool [enableLowPower](#)
- *Enable low power.*
bool [enableContinuousConversion](#)
- *Enable continuous conversion mode.*

5.3.1.0.0.1 Field Documentation

5.3.1.0.0.1.1 `adc16_reference_voltage_source_t` `adc16_config_t::referenceVoltageSource`

5.3.1.0.0.1.2 `adc16_clock_source_t` `adc16_config_t::clockSource`

5.3.1.0.0.1.3 `bool` `adc16_config_t::enableAsynchronousClock`

5.3.1.0.0.1.4 `adc16_clock_divider_t` `adc16_config_t::clockDivider`

5.3.1.0.0.1.5 `adc16_resolution_t` `adc16_config_t::resolution`

5.3.1.0.0.1.6 `adc16_long_sample_mode_t` `adc16_config_t::longSampleMode`

5.3.1.0.0.1.7 `bool` `adc16_config_t::enableHighSpeed`

5.3.1.0.0.1.8 `bool` `adc16_config_t::enableLowPower`

5.3.1.0.0.1.9 `bool` `adc16_config_t::enableContinuousConversion`

5.3.2 struct `adc16_hardware_compare_config_t`

Data Fields

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

5.3.2.0.0.2 Field Documentation

5.3.2.0.0.2.1 `adc16_hardware_compare_mode_t` `adc16_hardware_compare_config_t::hardwareCompareMode`

See "`adc16_hardware_compare_mode_t`".

Enumeration Type Documentation

5.3.2.0.0.2.2 int16_t adc16_hardware_compare_config_t::value1

5.3.2.0.0.2.3 int16_t adc16_hardware_compare_config_t::value2

5.3.3 struct adc16_channel_config_t

Data Fields

- uint32_t [channelNumber](#)
Setting the conversion channel number.
- bool [enableInterruptOnConversionCompleted](#)
Generate an interrupt request once the conversion is completed.

5.3.3.0.0.3 Field Documentation

5.3.3.0.0.3.1 uint32_t adc16_channel_config_t::channelNumber

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

5.3.3.0.0.3.2 bool adc16_channel_config_t::enableInterruptOnConversionCompleted

5.4 Macro Definition Documentation

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

5.5 Enumeration Type Documentation

5.5.1 enum _adc16_channel_status_flags

Enumerator

kADC16_ChannelConversionDoneFlag Conversion done.

5.5.2 enum _adc16_status_flags

Enumerator

kADC16_ActiveFlag Converter is active.

5.5.3 enum adc16_clock_divider_t

Enumerator

kADC16_ClockDivider1 For divider 1 from the input clock to the module.

kADC16_ClockDivider2 For divider 2 from the input clock to the module.

kADC16_ClockDivider4 For divider 4 from the input clock to the module.

kADC16_ClockDivider8 For divider 8 from the input clock to the module.

5.5.4 enum adc16_resolution_t

Enumerator

kADC16_Resolution8or9Bit Single End 8-bit or Differential Sample 9-bit.

kADC16_Resolution12or13Bit Single End 12-bit or Differential Sample 13-bit.

kADC16_Resolution10or11Bit Single End 10-bit or Differential Sample 11-bit.

kADC16_ResolutionSE8Bit Single End 8-bit.

kADC16_ResolutionSE12Bit Single End 12-bit.

kADC16_ResolutionSE10Bit Single End 10-bit.

5.5.5 enum adc16_clock_source_t

Enumerator

kADC16_ClockSourceAlt0 Selection 0 of the clock source.

kADC16_ClockSourceAlt1 Selection 1 of the clock source.

kADC16_ClockSourceAlt2 Selection 2 of the clock source.

kADC16_ClockSourceAlt3 Selection 3 of the clock source.

kADC16_ClockSourceAsynchronousClock Using internal asynchronous clock.

5.5.6 enum adc16_long_sample_mode_t

Enumerator

kADC16_LongSampleCycle24 20 extra ADCK cycles, 24 ADCK cycles total.

kADC16_LongSampleCycle16 12 extra ADCK cycles, 16 ADCK cycles total.

kADC16_LongSampleCycle10 6 extra ADCK cycles, 10 ADCK cycles total.

kADC16_LongSampleCycle6 2 extra ADCK cycles, 6 ADCK cycles total.

kADC16_LongSampleDisabled Disable the long sample feature.

5.5.7 enum adc16_reference_voltage_source_t

Enumerator

kADC16_ReferenceVoltageSourceVref For external pins pair of VrefH and VrefL.

kADC16_ReferenceVoltageSourceValt For alternate reference pair of ValtH and ValtL.

Function Documentation

5.5.8 enum adc16_hardware_compare_mode_t

Enumerator

kADC16_HardwareCompareMode0 $x < \text{value1}$.
kADC16_HardwareCompareMode1 $x > \text{value1}$.
kADC16_HardwareCompareMode2 if $\text{value1} \leq \text{value2}$, then $x < \text{value1} \parallel x > \text{value2}$; else, $\text{value1} > x > \text{value2}$.
kADC16_HardwareCompareMode3 if $\text{value1} \leq \text{value2}$, then $\text{value1} \leq x \leq \text{value2}$; else $x \geq \text{value1} \parallel x \leq \text{value2}$.

5.6 Function Documentation

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

Parameters

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

5.6.2 void ADC16_Deinit (ADC_Type * *base*)

Parameters

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

5.6.3 void ADC16_GetDefaultConfig (adc16_config_t * *config*)

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

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

Parameters

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

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

Parameters

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

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

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

Parameters

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

5.6.6 uint32_t ADC16_GetStatusFlags (ADC_Type * *base*)

Parameters

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

Returns

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

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

Function Documentation

Parameters

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

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

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

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

Parameters

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

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

Parameters

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

Returns

Conversion value.

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

Parameters

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

Returns

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

Chapter 6 Clock Driver

6.1 Overview

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

6.2 Get frequency

A centralized function `CLOCK_GetFreq` gets different clock type frequencies by passing a clock name. For example, pass a `kCLOCK_CoreSysClk` to get the core clock and pass a `kCLOCK_BusClk` to get the bus clock. Additionally, there are separate functions to get the frequency. For example, use `CLOCK_GetCoreSysClkFreq` to get the core clock frequency and `CLOCK_GetBusClkFreq` to get the bus clock frequency. Using these functions reduces the image size.

6.3 External clock frequency

The external clocks `EXTAL0/EXTAL1/EXTAL32` are decided by the board level design. The Clock driver uses variables `g_xtal0Freq/g_xtal1Freq/g_xtal32Freq` to save clock frequencies. Likewise, the APIs `CLOCK_SetXtal0Freq`, `CLOCK_SetXtal1Freq`, and `CLOCK_SetXtal32Freq` are used to set these variables.

The upper layer must set these values correctly. For example, after `OSC0(SYSOSC)` is initialized using `CLOCK_InitOsc0` or `CLOCK_InitSysOsc`, the upper layer should call the `CLOCK_SetXtal0Freq`. Otherwise, the clock frequency get functions may not receive valid values. This is useful for multicore platforms where only one core calls `CLOCK_InitOsc0` to initialize `OSC0` and other cores call `CLOCK_SetXtal0Freq`.

Modules

- [Multipurpose Clock Generator \(MCG\)](#)

Files

- file [fsl_clock.h](#)

Data Structures

- struct [sim_clock_config_t](#)
SIM configuration structure for clock setting. [More...](#)
- struct [oscer_config_t](#)
OSC configuration for OSCERCLK. [More...](#)
- struct [osc_config_t](#)
OSC Initialization Configuration Structure. [More...](#)
- struct [mcg_pll_config_t](#)

External clock frequency

- *MCG PLL configuration. [More...](#)*
- struct [mcg_config_t](#)
MCG mode change configuration structure. [More...](#)

Macros

- #define [MCG_CONFIG_CHECK_PARAM](#) 0U
Configures whether to check a parameter in a function.
- #define [FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL](#) 0
Configure whether driver controls clock.
- #define [DMAMUX_CLOCKS](#)
Clock ip name array for DMAMUX.
- #define [RTC_CLOCKS](#)
Clock ip name array for RTC.
- #define [SAI_CLOCKS](#)
Clock ip name array for SAI.
- #define [PORT_CLOCKS](#)
Clock ip name array for PORT.
- #define [EWM_CLOCKS](#)
Clock ip name array for EWM.
- #define [PIT_CLOCKS](#)
Clock ip name array for PIT.
- #define [DSPI_CLOCKS](#)
Clock ip name array for DSPI.
- #define [LPTMR_CLOCKS](#)
Clock ip name array for LPTMR.
- #define [FTM_CLOCKS](#)
Clock ip name array for FTM.
- #define [EDMA_CLOCKS](#)
Clock ip name array for EDMA.
- #define [ADC16_CLOCKS](#)
Clock ip name array for ADC16.
- #define [CMT_CLOCKS](#)
Clock ip name array for CMT.
- #define [UART_CLOCKS](#)
Clock ip name array for UART.
- #define [RNGA_CLOCKS](#)
Clock ip name array for RNGA.
- #define [CRC_CLOCKS](#)
Clock ip name array for CRC.
- #define [I2C_CLOCKS](#)
Clock ip name array for I2C.
- #define [PDB_CLOCKS](#)
Clock ip name array for PDB.
- #define [CMP_CLOCKS](#)
Clock ip name array for CMP.
- #define [FTF_CLOCKS](#)
Clock ip name array for FTF.
- #define [LPO_CLK_FREQ](#) 1000U
LPO clock frequency.
- #define [SYS_CLK](#) kCLOCK_CoreSysClk

Peripherals clock source definition.

Enumerations

- enum `clock_name_t` {
`kCLOCK_CoreSysClk`,
`kCLOCK_PlatClk`,
`kCLOCK_BusClk`,
`kCLOCK_FlashClk`,
`kCLOCK_PllFllSelClk`,
`kCLOCK_Er32kClk`,
`kCLOCK_Osc0ErClk`,
`kCLOCK_McgFixedFreqClk`,
`kCLOCK_McgInternalRefClk`,
`kCLOCK_McgFllClk`,
`kCLOCK_McgPll0Clk`,
`kCLOCK_McgPll1Clk`,
`kCLOCK_McgExtPllClk`,
`kCLOCK_McgPeriphClk`,
`kCLOCK_LpoClk` }
Clock name used to get clock frequency.
- enum `clock_usb_src_t` {
`kCLOCK_UsbSrcPll0` = `SIM_SOPT2_USBSRC(1U) | SIM_SOPT2_PLLFLLSEL(1U)`,
`kCLOCK_UsbSrcExt` = `SIM_SOPT2_USBSRC(0U)` }
USB clock source definition.
- enum `clock_ip_name_t`
Clock gate name used for `CLOCK_EnableClock/CLOCK_DisableClock`.
- enum `osc_mode_t` {
`kOSC_ModeExt` = `0U`,
`kOSC_ModeOscLowPower` = `MCG_C2_EREFS0_MASK`,
`kOSC_ModeOscHighGain` }
OSC work mode.
- enum `_osc_cap_load` {
`kOSC_Cap2P` = `OSC_CR_SC2P_MASK`,
`kOSC_Cap4P` = `OSC_CR_SC4P_MASK`,
`kOSC_Cap8P` = `OSC_CR_SC8P_MASK`,
`kOSC_Cap16P` = `OSC_CR_SC16P_MASK` }
Oscillator capacitor load setting.
- enum `_oscer_enable_mode` {
`kOSC_ErClkEnable` = `OSC_CR_ERCLKEN_MASK`,
`kOSC_ErClkEnableInStop` = `OSC_CR_EREFS0_MASK` }
OSCERCLK enable mode.
- enum `mcg_fll_src_t` {
`kMCG_FllSrcExternal`,
`kMCG_FllSrcInternal` }
MCG FLL reference clock source select.
- enum `mcg_irc_mode_t` {

External clock frequency

- `kMCG_IrcSlow,`
 - `kMCG_IrcFast }`
 - MCG internal reference clock select.*
- enum `mcg_dmx32_t` {
 - `kMCG_Dmx32Default,`
 - `kMCG_Dmx32Fine }`
 - MCG DCO Maximum Frequency with 32.768 kHz Reference.*
- enum `mcg_drs_t` {
 - `kMCG_DrsLow,`
 - `kMCG_DrsMid,`
 - `kMCG_DrsMidHigh,`
 - `kMCG_DrsHigh }`
 - MCG DCO range select.*
- enum `mcg_pll_ref_src_t` {
 - `kMCG_PllRefOsc0,`
 - `kMCG_PllRefOsc1 }`
 - MCG PLL reference clock select.*
- enum `mcg_clkout_src_t` {
 - `kMCG_ClkOutSrcOut,`
 - `kMCG_ClkOutSrcInternal,`
 - `kMCG_ClkOutSrcExternal }`
 - MCGOUT clock source.*
- enum `mcg_atm_select_t` {
 - `kMCG_AtmSel32k,`
 - `kMCG_AtmSel4m }`
 - MCG Automatic Trim Machine Select.*
- enum `mcg_oscsel_t` {
 - `kMCG_OscselOsc,`
 - `kMCG_OscselRtc }`
 - MCG OSC Clock Select.*
- enum `mcg_pll_clk_select_t` { `kMCG_PllClkSelPll0` }
 - MCG PLLCS select.*
- enum `mcg_monitor_mode_t` {
 - `kMCG_MonitorNone,`
 - `kMCG_MonitorInt,`
 - `kMCG_MonitorReset }`
 - MCG clock monitor mode.*
- enum `_mcg_status` {
 - `kStatus_MCG_ModeUnreachable` = MAKE_STATUS(kStatusGroup_MCG, 0),
 - `kStatus_MCG_ModeInvalid` = MAKE_STATUS(kStatusGroup_MCG, 1),
 - `kStatus_MCG_AtmBusClockInvalid` = MAKE_STATUS(kStatusGroup_MCG, 2),
 - `kStatus_MCG_AtmDesiredFreqInvalid` = MAKE_STATUS(kStatusGroup_MCG, 3),
 - `kStatus_MCG_AtmIrcUsed` = MAKE_STATUS(kStatusGroup_MCG, 4),
 - `kStatus_MCG_AtmHardwareFail` = MAKE_STATUS(kStatusGroup_MCG, 5),
 - `kStatus_MCG_SourceUsed` = MAKE_STATUS(kStatusGroup_MCG, 6) }
 - MCG status.*
- enum `_mcg_status_flags_t` {

```

kMCG_Osc0LostFlag = (1U << 0U),
kMCG_Osc0InitFlag = (1U << 1U),
kMCG_RtcOscLostFlag = (1U << 4U),
kMCG_Pll0LostFlag = (1U << 5U),
kMCG_Pll0LockFlag = (1U << 6U) }

    MCG status flags.
• enum _mcg_ircclk_enable_mode {
    kMCG_IrcclkEnable = MCG_C1_IRCLKEN_MASK,
    kMCG_IrcclkEnableInStop = MCG_C1_IREFSTEN_MASK }
    MCG internal reference clock (MCGIRCLK) enable mode definition.
• enum _mcg_pll_enable_mode {
    kMCG_PllEnableIndependent = MCG_C5_PLLCLKEN0_MASK,
    kMCG_PllEnableInStop = MCG_C5_PLLSTEN0_MASK }
    MCG PLL clock enable mode definition.
• enum mcg_mode_t {
    kMCG_ModeFEI = 0U,
    kMCG_ModeFBI,
    kMCG_ModeBLPI,
    kMCG_ModeFEE,
    kMCG_ModeFBE,
    kMCG_ModeBLPE,
    kMCG_ModePBE,
    kMCG_ModePEE,
    kMCG_ModeError }
    MCG mode definitions.

```

Functions

- static void [CLOCK_EnableClock](#) ([clock_ip_name_t](#) name)
Enable the clock for specific IP.
- static void [CLOCK_DisableClock](#) ([clock_ip_name_t](#) name)
Disable the clock for specific IP.
- static void [CLOCK_SetEr32kClock](#) ([uint32_t](#) src)
Set ERCLK32K source.
- static void [CLOCK_SetTraceClock](#) ([uint32_t](#) src)
Set debug trace clock source.
- static void [CLOCK_SetPllFllSelClock](#) ([uint32_t](#) src)
Set PLLFLLSEL clock source.
- static void [CLOCK_SetClkOutClock](#) ([uint32_t](#) src)
Set CLKOUT source.
- static void [CLOCK_SetRtcClkOutClock](#) ([uint32_t](#) src)
Set RTC_CLKOUT source.
- bool [CLOCK_EnableUsbfs0Clock](#) ([clock_usb_src_t](#) src, [uint32_t](#) freq)
Enable USB FS clock.
- static void [CLOCK_DisableUsbfs0Clock](#) (void)
Disable USB FS clock.
- static void [CLOCK_SetOutDiv](#) ([uint32_t](#) outdiv1, [uint32_t](#) outdiv2, [uint32_t](#) outdiv4)
System clock divider.
- [uint32_t](#) [CLOCK_GetFreq](#) ([clock_name_t](#) clockName)

External clock frequency

- `uint32_t CLOCK_GetCoreSysClkFreq (void)`
Gets the clock frequency for a specific clock name.
- `uint32_t CLOCK_GetPlatClkFreq (void)`
Get the core clock or system clock frequency.
- `uint32_t CLOCK_GetBusClkFreq (void)`
Get the platform clock frequency.
- `uint32_t CLOCK_GetFlashClkFreq (void)`
Get the bus clock frequency.
- `uint32_t CLOCK_GetFlashClkFreq (void)`
Get the flash clock frequency.
- `uint32_t CLOCK_GetPllFllSelClkFreq (void)`
Get the output clock frequency selected by SIM[PLLFLSEL].
- `uint32_t CLOCK_GetEr32kClkFreq (void)`
Get the external reference 32K clock frequency (ERCLK32K).
- `uint32_t CLOCK_GetOsc0ErClkFreq (void)`
Get the OSC0 external reference clock frequency (OSC0ERCLK).
- `void CLOCK_SetSimConfig (sim_clock_config_t const *config)`
Set the clock configure in SIM module.
- `static void CLOCK_SetSimSafeDivs (void)`
Set the system clock dividers in SIM to safe value.

Variables

- `uint32_t g_xtal0Freq`
External XTAL0 (OSC0) clock frequency.
- `uint32_t g_xtal32Freq`
External XTAL32/EXTAL32/RTC_CLKIN clock frequency.

Driver version

- `#define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 2, 1))`
CLOCK driver version 2.2.1.

MCG frequency functions.

- `uint32_t CLOCK_GetOutClkFreq (void)`
Gets the MCG output clock (MCGOUTCLK) frequency.
- `uint32_t CLOCK_GetFllFreq (void)`
Gets the MCG FLL clock (MCGFLLCLK) frequency.
- `uint32_t CLOCK_GetInternalRefClkFreq (void)`
Gets the MCG internal reference clock (MCGIRCLK) frequency.
- `uint32_t CLOCK_GetFixedFreqClkFreq (void)`
Gets the MCG fixed frequency clock (MCGFFCLK) frequency.
- `uint32_t CLOCK_GetPll0Freq (void)`
Gets the MCG PLL0 clock (MCGPLL0CLK) frequency.

MCG clock configuration.

- `static void CLOCK_SetLowPowerEnable (bool enable)`
Enables or disables the MCG low power.
- `status_t CLOCK_SetInternalRefClkConfig (uint8_t enableMode, mcg_irc_mode_t ircs, uint8_t fcr-div)`

- *Configures the Internal Reference clock (MCGIRCLK).*
- status_t [CLOCK_SetExternalRefClkConfig](#) (mcg_oscsel_t oscsel)
Selects the MCG external reference clock.
- static void [CLOCK_SetFllExtRefDiv](#) (uint8_t frdiv)
Set the FLL external reference clock divider value.
- void [CLOCK_EnablePll0](#) (mcg_pll_config_t const *config)
Enables the PLL0 in FLL mode.
- static void [CLOCK_DisablePll0](#) (void)
Disables the PLL0 in FLL mode.
- uint32_t [CLOCK_CalcPllDiv](#) (uint32_t refFreq, uint32_t desireFreq, uint8_t *prdiv, uint8_t *vdiv)
Calculates the PLL divider setting for a desired output frequency.

MCG clock lock monitor functions.

- void [CLOCK_SetOsc0MonitorMode](#) (mcg_monitor_mode_t mode)
Sets the OSC0 clock monitor mode.
- void [CLOCK_SetRtcOscMonitorMode](#) (mcg_monitor_mode_t mode)
Sets the RTC OSC clock monitor mode.
- void [CLOCK_SetPll0MonitorMode](#) (mcg_monitor_mode_t mode)
Sets the PLL0 clock monitor mode.
- uint32_t [CLOCK_GetStatusFlags](#) (void)
Gets the MCG status flags.
- void [CLOCK_ClearStatusFlags](#) (uint32_t mask)
Clears the MCG status flags.

OSC configuration

- static void [OSC_SetExtRefClkConfig](#) (OSC_Type *base, oscar_config_t const *config)
Configures the OSC external reference clock (OSCERCLK).
- static void [OSC_SetCapLoad](#) (OSC_Type *base, uint8_t capLoad)
Sets the capacitor load configuration for the oscillator.
- void [CLOCK_InitOsc0](#) (osc_config_t const *config)
Initializes the OSC0.
- void [CLOCK_DeinitOsc0](#) (void)
Deinitializes the OSC0.

External clock frequency

- static void [CLOCK_SetXtal0Freq](#) (uint32_t freq)
Sets the XTAL0 frequency based on board settings.
- static void [CLOCK_SetXtal32Freq](#) (uint32_t freq)
Sets the XTAL32/RTC_CLKIN frequency based on board settings.

MCG auto-trim machine.

- status_t [CLOCK_TrimInternalRefClk](#) (uint32_t extFreq, uint32_t desireFreq, uint32_t *actualFreq, mcg_atm_select_t atms)
Auto trims the internal reference clock.

MCG mode functions.

- `mcb_mode_t` `CLOCK_GetMode` (void)
Gets the current MCG mode.
- `status_t` `CLOCK_SetFeiMode` (`mcb_dmx32_t` `dmx32`, `mcb_drs_t` `drs`, void(*fllStableDelay)(void))
Sets the MCG to FEI mode.
- `status_t` `CLOCK_SetFeeMode` (uint8_t `frdiv`, `mcb_dmx32_t` `dmx32`, `mcb_drs_t` `drs`, void(*fllStableDelay)(void))
Sets the MCG to FEE mode.
- `status_t` `CLOCK_SetFbiMode` (`mcb_dmx32_t` `dmx32`, `mcb_drs_t` `drs`, void(*fllStableDelay)(void))
Sets the MCG to FBI mode.
- `status_t` `CLOCK_SetFbeMode` (uint8_t `frdiv`, `mcb_dmx32_t` `dmx32`, `mcb_drs_t` `drs`, void(*fllStableDelay)(void))
Sets the MCG to FBE mode.
- `status_t` `CLOCK_SetBlpiMode` (void)
Sets the MCG to BLPI mode.
- `status_t` `CLOCK_SetBlpeMode` (void)
Sets the MCG to BLPE mode.
- `status_t` `CLOCK_SetPbeMode` (`mcb_pll_clk_select_t` `pllcs`, `mcb_pll_config_t` const *`config`)
Sets the MCG to PBE mode.
- `status_t` `CLOCK_SetPeeMode` (void)
Sets the MCG to PEE mode.
- `status_t` `CLOCK_ExternalModeToFbeModeQuick` (void)
Switches the MCG to FBE mode from the external mode.
- `status_t` `CLOCK_InternalModeToFbiModeQuick` (void)
Switches the MCG to FBI mode from internal modes.
- `status_t` `CLOCK_BootToFeiMode` (`mcb_dmx32_t` `dmx32`, `mcb_drs_t` `drs`, void(*fllStableDelay)(void))
Sets the MCG to FEI mode during system boot up.
- `status_t` `CLOCK_BootToFeeMode` (`mcb_oscsel_t` `oscsel`, uint8_t `frdiv`, `mcb_dmx32_t` `dmx32`, `mcb_drs_t` `drs`, void(*fllStableDelay)(void))
Sets the MCG to FEE mode during system boot up.
- `status_t` `CLOCK_BootToBlpiMode` (uint8_t `frdiv`, `mcb_irc_mode_t` `ircs`, uint8_t `ircEnableMode`)
Sets the MCG to BLPI mode during system boot up.
- `status_t` `CLOCK_BootToBlpeMode` (`mcb_oscsel_t` `oscsel`)
Sets the MCG to BLPE mode during system boot up.
- `status_t` `CLOCK_BootToPeeMode` (`mcb_oscsel_t` `oscsel`, `mcb_pll_clk_select_t` `pllcs`, `mcb_pll_config_t` const *`config`)
Sets the MCG to PEE mode during system boot up.
- `status_t` `CLOCK_SetMcbConfig` (`mcb_config_t` const *`config`)
Sets the MCG to a target mode.

6.4 Data Structure Documentation

6.4.1 struct `sim_clock_config_t`

Data Fields

- uint8_t `pllFllSel`
PLL/FLL/IRC48M selection.

- uint8_t [er32kSrc](#)
ERCLK32K source selection.
- uint32_t [clkdiv1](#)
SIM_CLKDIV1.

6.4.1.0.0.4 Field Documentation

6.4.1.0.0.4.1 uint8_t sim_clock_config_t::pllFllSel

6.4.1.0.0.4.2 uint8_t sim_clock_config_t::er32kSrc

6.4.1.0.0.4.3 uint32_t sim_clock_config_t::clkdiv1

6.4.2 struct oscr_config_t

Data Fields

- uint8_t [enableMode](#)
OSCERCLK enable mode.

6.4.2.0.0.5 Field Documentation

6.4.2.0.0.5.1 uint8_t oscr_config_t::enableMode

OR'ed value of [_oscer_enable_mode](#).

6.4.3 struct osc_config_t

Defines the configuration data structure to initialize the OSC. When porting to a new board, set the following members according to the board setting:

1. freq: The external frequency.
2. workMode: The OSC module mode.

Data Fields

- uint32_t [freq](#)
External clock frequency.
- uint8_t [capLoad](#)
Capacitor load setting.
- [osc_mode_t](#) [workMode](#)
OSC work mode setting.
- [oscer_config_t](#) [oscerConfig](#)
Configuration for OSCERCLK.

Data Structure Documentation

6.4.3.0.0.6 Field Documentation

6.4.3.0.0.6.1 `uint32_t osc_config_t::freq`

6.4.3.0.0.6.2 `uint8_t osc_config_t::capLoad`

6.4.3.0.0.6.3 `osc_mode_t osc_config_t::workMode`

6.4.3.0.0.6.4 `oscer_config_t osc_config_t::oscerConfig`

6.4.4 `struct mcg_pll_config_t`

Data Fields

- `uint8_t enableMode`
Enable mode.
- `uint8_t prdiv`
Reference divider PRDIV.
- `uint8_t vdiv`
VCO divider VDIV.

6.4.4.0.0.7 Field Documentation

6.4.4.0.0.7.1 `uint8_t mcg_pll_config_t::enableMode`

OR'ed value of `_mcg_pll_enable_mode`.

6.4.4.0.0.7.2 `uint8_t mcg_pll_config_t::prdiv`

6.4.4.0.0.7.3 `uint8_t mcg_pll_config_t::vdiv`

6.4.5 `struct mcg_config_t`

When porting to a new board, set the following members according to the board setting:

1. `frdiv`: If the FLL uses the external reference clock, set this value to ensure that the external reference clock divided by `frdiv` is in the 31.25 kHz to 39.0625 kHz range.
2. The PLL reference clock divider `PRDIV`: PLL reference clock frequency after `PRDIV` should be in the `FSL_FEATURE_MCG_PLL_REF_MIN` to `FSL_FEATURE_MCG_PLL_REF_MAX` range.

Data Fields

- `mcg_mode_t mcgMode`
MCG mode.
- `uint8_t irclkEnableMode`
MCGIRCLK enable mode.
- `mcg_irc_mode_t ircs`
Source, MCG_C2[IRCS].

- `uint8_t fcrdiv`
Divider, MCG_SC[FCRDIV].
- `uint8_t frdiv`
Divider MCG_C1[FRDIV].
- `mcg_drs_t drs`
DCO range MCG_C4[DRST_DRS].
- `mcg_dmx32_t dmx32`
MCG_C4[DMX32].
- `mcg_oscsel_t oscsel`
OSC select MCG_C7[OSCSEL].
- `mcg_pll_config_t pll0Config`
MCGPLL0CLK configuration.

6.4.5.0.0.8 Field Documentation

6.4.5.0.0.8.1 `mcg_mode_t mcg_config_t::mcgMode`

6.4.5.0.0.8.2 `uint8_t mcg_config_t::ircclkEnableMode`

6.4.5.0.0.8.3 `mcg_irc_mode_t mcg_config_t::ircs`

6.4.5.0.0.8.4 `uint8_t mcg_config_t::fcrdiv`

6.4.5.0.0.8.5 `uint8_t mcg_config_t::frdiv`

6.4.5.0.0.8.6 `mcg_drs_t mcg_config_t::drs`

6.4.5.0.0.8.7 `mcg_dmx32_t mcg_config_t::dmx32`

6.4.5.0.0.8.8 `mcg_oscsel_t mcg_config_t::oscsel`

6.4.5.0.0.8.9 `mcg_pll_config_t mcg_config_t::pll0Config`

6.5 Macro Definition Documentation

6.5.1 `#define MCG_CONFIG_CHECK_PARAM 0U`

Some MCG settings must be changed with conditions, for example:

1. MCGIRCLK settings, such as the source, divider, and the trim value should not change when MCGIRCLK is used as a system clock source.
2. MCG_C7[OSCSEL] should not be changed when the external reference clock is used as a system clock source. For example, in FBE/BLPE/PBE modes.
3. The users should only switch between the supported clock modes.

MCG functions check the parameter and MCG status before setting, if not allowed to change, the functions return error. The parameter checking increases code size, if code size is a critical requirement, change `MCG_CONFIG_CHECK_PARAM` to 0 to disable parameter checking.

6.5.2 #define FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL 0

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note

All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

6.5.3 #define FSL_CLOCK_DRIVER_VERSION (MAKE_VERSION(2, 2, 1))

6.5.4 #define DMAMUX_CLOCKS

Value:

```
{  
    \kCLOCK_Dmamux0 \  
}
```

6.5.5 #define RTC_CLOCKS

Value:

```
{  
    \kCLOCK_Rtc0 \  
}
```

6.5.6 #define SAI_CLOCKS

Value:

```
{  
    \kCLOCK_Sai0 \  
}
```

6.5.7 #define PORT_CLOCKS

Value:

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

6.5.8 #define EWM_CLOCKS

Value:

```
{  
    \kCLOCK_Ewm0 \  
}
```

6.5.9 #define PIT_CLOCKS

Value:

```
{  
    \kCLOCK_Pit0 \  
}
```

6.5.10 #define DSPI_CLOCKS

Value:

```
{  
    \kCLOCK_Spi0, kCLOCK_Spi1 \  
}
```

6.5.11 #define LPTMR_CLOCKS

Value:

```
{  
    \kCLOCK_Lptmr0 \  
}
```

6.5.12 #define FTM_CLOCKS

Value:

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

6.5.13 #define EDMA_CLOCKS

Value:

```
{  
    \kCLOCK_Dma0 \  
}
```

6.5.14 #define ADC16_CLOCKS

Value:

```
{  
    \kCLOCK_Adc0 \  
}
```

6.5.15 #define CMT_CLOCKS

Value:

```
{  
    \kCLOCK_Cmt0 \  
}
```

6.5.16 #define UART_CLOCKS

Value:

```
{  
    \kCLOCK_Uart0, kCLOCK_Uart1, kCLOCK_Uart2 \  
}
```

6.5.17 #define RNGA_CLOCKS

Value:

```
{  
    \kCLOCK_Rnga0 \  
}
```

6.5.18 #define CRC_CLOCKS

Value:

```
{  
    \kCLOCK_Crc0 \  
}
```

6.5.19 #define I2C_CLOCKS

Value:

```
{  
    \kCLOCK_I2c0, kCLOCK_I2c1 \  
}
```

6.5.20 #define PDB_CLOCKS

Value:

```
{  
    \kCLOCK_Pdb0 \  
}
```

6.5.21 #define CMP_CLOCKS

Value:

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

6.5.22 #define FTF_CLOCKS

Value:

```
{  
    \kCLOCK_Ftf0 \  
}
```

Enumeration Type Documentation

6.5.23 #define SYS_CLK kCLOCK_CoreSysClk

6.6 Enumeration Type Documentation

6.6.1 enum clock_name_t

Enumerator

kCLOCK_CoreSysClk Core/system clock.
kCLOCK_PlatClk Platform clock.
kCLOCK_BusClk Bus clock.
kCLOCK_FlashClk Flash clock.
kCLOCK_PllFllSelClk The clock after SIM[PLL_FLLSEL].
kCLOCK_Er32kClk External reference 32K clock (ERCLK32K)
kCLOCK_Osc0ErClk OSC0 external reference clock (OSC0ERCLK)
kCLOCK_McgFixedFreqClk MCG fixed frequency clock (MCGFFCLK)
kCLOCK_McgInternalRefClk MCG internal reference clock (MCGIRCLK)
kCLOCK_McgFllClk MCGFLLCLK.
kCLOCK_McgPll0Clk MCGPLL0CLK.
kCLOCK_McgPll1Clk MCGPLL1CLK.
kCLOCK_McgExtPllClk EXT_PLLCLK.
kCLOCK_McgPeriphClk MCG peripheral clock (MCGPCLK)
kCLOCK_LpoClk LPO clock.

6.6.2 enum clock_usb_src_t

Enumerator

kCLOCK_UsbSrcPll0 Use PLL0.
kCLOCK_UsbSrcExt Use USB_CLKIN.

6.6.3 enum clock_ip_name_t

6.6.4 enum osc_mode_t

Enumerator

kOSC_ModeExt Use an external clock.
kOSC_ModeOscLowPower Oscillator low power.
kOSC_ModeOscHighGain Oscillator high gain.

6.6.5 enum _osc_cap_load

Enumerator

kOSC_Cap2P 2 pF capacitor load
kOSC_Cap4P 4 pF capacitor load
kOSC_Cap8P 8 pF capacitor load
kOSC_Cap16P 16 pF capacitor load

6.6.6 enum _oscer_enable_mode

Enumerator

kOSC_ErClkEnable Enable.
kOSC_ErClkEnableInStop Enable in stop mode.

6.6.7 enum mcg_fl_src_t

Enumerator

kMCG_FllSrcExternal External reference clock is selected.
kMCG_FllSrcInternal The slow internal reference clock is selected.

6.6.8 enum mcg_irc_mode_t

Enumerator

kMCG_IrcSlow Slow internal reference clock selected.
kMCG_IrcFast Fast internal reference clock selected.

6.6.9 enum mcg_dmx32_t

Enumerator

kMCG_Dmx32Default DCO has a default range of 25%.
kMCG_Dmx32Fine DCO is fine-tuned for maximum frequency with 32.768 kHz reference.

Enumeration Type Documentation

6.6.10 enum mcg_drs_t

Enumerator

kMCG_DrsLow Low frequency range.
kMCG_DrsMid Mid frequency range.
kMCG_DrsMidHigh Mid-High frequency range.
kMCG_DrsHigh High frequency range.

6.6.11 enum mcg_pll_ref_src_t

Enumerator

kMCG_PllRefOsc0 Selects OSC0 as PLL reference clock.
kMCG_PllRefOsc1 Selects OSC1 as PLL reference clock.

6.6.12 enum mcg_clkout_src_t

Enumerator

kMCG_ClkOutSrcOut Output of the FLL is selected (reset default)
kMCG_ClkOutSrcInternal Internal reference clock is selected.
kMCG_ClkOutSrcExternal External reference clock is selected.

6.6.13 enum mcg_atm_select_t

Enumerator

kMCG_AtmSel32k 32 kHz Internal Reference Clock selected
kMCG_AtmSel4m 4 MHz Internal Reference Clock selected

6.6.14 enum mcg_oscsel_t

Enumerator

kMCG_OscselOsc Selects System Oscillator (OSCCLK)
kMCG_OscselRtc Selects 32 kHz RTC Oscillator.

6.6.15 enum mcg_pll_clk_select_t

Enumerator

kMCG_PllClkSelPll0 PLL0 output clock is selected.

6.6.16 enum mcg_monitor_mode_t

Enumerator

kMCG_MonitorNone Clock monitor is disabled.

kMCG_MonitorInt Trigger interrupt when clock lost.

kMCG_MonitorReset System reset when clock lost.

6.6.17 enum _mcg_status

Enumerator

kStatus_MCG_ModeUnreachable Can't switch to target mode.

kStatus_MCG_ModeInvalid Current mode invalid for the specific function.

kStatus_MCG_AtmBusClockInvalid Invalid bus clock for ATM.

kStatus_MCG_AtmDesiredFreqInvalid Invalid desired frequency for ATM.

kStatus_MCG_AtmIrcUsed IRC is used when using ATM.

kStatus_MCG_AtmHardwareFail Hardware fail occurs during ATM.

kStatus_MCG_SourceUsed Can't change the clock source because it is in use.

6.6.18 enum _mcg_status_flags_t

Enumerator

kMCG_Osc0LostFlag OSC0 lost.

kMCG_Osc0InitFlag OSC0 crystal initialized.

kMCG_RtcOscLostFlag RTC OSC lost.

kMCG_Pll0LostFlag PLL0 lost.

kMCG_Pll0LockFlag PLL0 locked.

6.6.19 enum _mcg_ircclk_enable_mode

Enumerator

kMCG_IrcclkEnable MCGIRCLK enable.

kMCG_IrcclkEnableInStop MCGIRCLK enable in stop mode.

Function Documentation

6.6.20 enum _mcg_pll_enable_mode

Enumerator

kMCG_PllEnableIndependent MCGPLLCLK enable independent of the MCG clock mode. Generally, the PLL is disabled in FLL modes (FEI/FBI/FEE/FBE). Setting the PLL clock enable independent, enables the PLL in the FLL modes.

kMCG_PllEnableInStop MCGPLLCLK enable in STOP mode.

6.6.21 enum mcg_mode_t

Enumerator

kMCG_ModeFEI FEI - FLL Engaged Internal.

kMCG_ModeFBI FBI - FLL Bypassed Internal.

kMCG_ModeBLPI BLPI - Bypassed Low Power Internal.

kMCG_ModeFEE FEE - FLL Engaged External.

kMCG_ModeFBE FBE - FLL Bypassed External.

kMCG_ModeBLPE BLPE - Bypassed Low Power External.

kMCG_ModePBE PBE - PLL Bypassed External.

kMCG_ModePEE PEE - PLL Engaged External.

kMCG_ModeError Unknown mode.

6.7 Function Documentation

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

Parameters

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

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

Parameters

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

6.7.3 static void CLOCK_SetEr32kClock (uint32_t src) [inline], [static]

Parameters

<i>src</i>	The value to set ERCLK32K clock source.
------------	---

6.7.4 static void CLOCK_SetTraceClock (uint32_t src) [inline], [static]

Parameters

<i>src</i>	The value to set debug trace clock source.
------------	--

6.7.5 static void CLOCK_SetPIIFllSelClock (uint32_t src) [inline], [static]

Parameters

<i>src</i>	The value to set PLLFLLSEL clock source.
------------	--

6.7.6 static void CLOCK_SetClkOutClock (uint32_t src) [inline], [static]

Parameters

<i>src</i>	The value to set CLKOUT source.
------------	---------------------------------

6.7.7 static void CLOCK_SetRtcClkOutClock (uint32_t src) [inline], [static]

Parameters

Function Documentation

<i>src</i>	The value to set RTC_CLKOUT source.
------------	-------------------------------------

6.7.8 bool CLOCK_EnableUsbfs0Clock (clock_usb_src_t *src*, uint32_t *freq*)

Parameters

<i>src</i>	USB FS clock source.
<i>freq</i>	The frequency specified by <i>src</i> .

Return values

<i>true</i>	The clock is set successfully.
<i>false</i>	The clock source is invalid to get proper USB FS clock.

6.7.9 static void CLOCK_DisableUsbfs0Clock (void) [inline], [static]

Disable USB FS clock.

6.7.10 static void CLOCK_SetOutDiv (uint32_t *outdiv1*, uint32_t *outdiv2*, uint32_t *outdiv4*) [inline], [static]

Set the SIM_CLKDIV1[OUTDIV1], SIM_CLKDIV1[OUTDIV2], SIM_CLKDIV1[OUTDIV4].

Parameters

<i>outdiv1</i>	Clock 1 output divider value.
<i>outdiv2</i>	Clock 2 output divider value.
<i>outdiv4</i>	Clock 4 output divider value.

6.7.11 uint32_t CLOCK_GetFreq (clock_name_t *clockName*)

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in *clock_name_t*. The MCG must be properly configured before using this function.

Parameters

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

Returns

Clock frequency value in Hertz

6.7.12 uint32_t CLOCK_GetCoreSysClkFreq (void)

Returns

Clock frequency in Hz.

6.7.13 uint32_t CLOCK_GetPlatClkFreq (void)

Returns

Clock frequency in Hz.

6.7.14 uint32_t CLOCK_GetBusClkFreq (void)

Returns

Clock frequency in Hz.

6.7.15 uint32_t CLOCK_GetFlashClkFreq (void)

Returns

Clock frequency in Hz.

6.7.16 uint32_t CLOCK_GetPIIFIISeIClkFreq (void)

Returns

Clock frequency in Hz.

Function Documentation

6.7.17 uint32_t CLOCK_GetEr32kClkFreq (void)

Returns

Clock frequency in Hz.

6.7.18 uint32_t CLOCK_GetOsc0ErClkFreq (void)

Returns

Clock frequency in Hz.

6.7.19 void CLOCK_SetSimConfig (sim_clock_config_t const * *config*)

This function sets system layer clock settings in SIM module.

Parameters

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

6.7.20 static void CLOCK_SetSimSafeDivs (void) [inline], [static]

The system level clocks (core clock, bus clock, flexbus clock and flash clock) must be in allowed ranges. During MCG clock mode switch, the MCG output clock changes then the system level clocks may be out of range. This function could be used before MCG mode change, to make sure system level clocks are in allowed range.

Parameters

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

6.7.21 uint32_t CLOCK_GetOutClkFreq (void)

This function gets the MCG output clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGOUTCLK.

6.7.22 uint32_t CLOCK_GetFllFreq (void)

This function gets the MCG FLL clock frequency in Hz based on the current MCG register value. The FLL is enabled in FEI/FBI/FEE/FBE mode and disabled in low power state in other modes.

Returns

The frequency of MCGFLLCLK.

6.7.23 uint32_t CLOCK_GetInternalRefClkFreq (void)

This function gets the MCG internal reference clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGIRCLK.

6.7.24 uint32_t CLOCK_GetFixedFreqClkFreq (void)

This function gets the MCG fixed frequency clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGFFCLK.

6.7.25 uint32_t CLOCK_GetPll0Freq (void)

This function gets the MCG PLL0 clock frequency in Hz based on the current MCG register value.

Returns

The frequency of MCGPLL0CLK.

**6.7.26 static void CLOCK_SetLowPowerEnable (bool *enable*) [inline],
[static]**

Enabling the MCG low power disables the PLL and FLL in bypass modes. In other words, in FBE and PBE modes, enabling low power sets the MCG to BLPE mode. In FBI and PBI modes, enabling low power sets the MCG to BLPI mode. When disabling the MCG low power, the PLL or FLL are enabled based on MCG settings.

Function Documentation

Parameters

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

6.7.27 **status_t** CLOCK_SetInternalRefClkConfig (**uint8_t** *enableMode*, **mcg_irc_mode_t** *ircs*, **uint8_t** *fcrdiv*)

This function sets the MCGIRCLK base on parameters. It also selects the IRC source. If the fast IRC is used, this function sets the fast IRC divider. This function also sets whether the MCGIRCLK is enabled in stop mode. Calling this function in FBI/PBI/BLPI modes may change the system clock. As a result, using the function in these modes it is not allowed.

Parameters

<i>enableMode</i>	MCGIRCLK enable mode, OR'ed value of _mcg_ircclk_enable_mode .
<i>ircs</i>	MCGIRCLK clock source, choose fast or slow.
<i>fcrdiv</i>	Fast IRC divider setting (FCRDIV).

Return values

<i>kStatus_MCG_Source-Used</i>	Because the internal reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.
<i>kStatus_Success</i>	MCGIRCLK configuration finished successfully.

6.7.28 **status_t** CLOCK_SetExternalRefClkConfig (**mcg_oscsel_t** *oscsel*)

Selects the MCG external reference clock source, changes the MCG_C7[OSCSEL], and waits for the clock source to be stable. Because the external reference clock should not be changed in FEE/FBE/BLP-E/PBE/PEE modes, do not call this function in these modes.

Parameters

<i>oscsel</i>	MCG external reference clock source, MCG_C7[OSCSEL].
---------------	--

Return values

<i>kStatus_MCG_Source-Used</i>	Because the external reference clock is used as a clock source, the configuration should not be changed. Otherwise, a glitch occurs.
<i>kStatus_Success</i>	External reference clock set successfully.

6.7.29 static void CLOCK_SetFllExtRefDiv (uint8_t *frdiv*) [inline], [static]

Sets the FLL external reference clock divider value, the register MCG_C1[FRDIV].

Parameters

<i>frdiv</i>	The FLL external reference clock divider value, MCG_C1[FRDIV].
--------------	--

6.7.30 void CLOCK_EnablePll0 (mcg_pll_config_t const * *config*)

This function sets up the PLL0 in FLL mode and reconfigures the PLL0. Ensure that the PLL reference clock is enabled before calling this function and that the PLL0 is not used as a clock source. The function CLOCK_CalcPllDiv gets the correct PLL divider values.

Parameters

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

6.7.31 static void CLOCK_DisablePll0 (void) [inline], [static]

This function disables the PLL0 in FLL mode. It should be used together with the [CLOCK_EnablePll0](#).

6.7.32 uint32_t CLOCK_CalcPllDiv (uint32_t *refFreq*, uint32_t *desireFreq*, uint8_t * *prdiv*, uint8_t * *vdiv*)

This function calculates the correct reference clock divider (PRDIV) and VCO divider (VDIV) to generate a desired PLL output frequency. It returns the closest frequency match with the corresponding PRDIV/VDIV returned from parameters. If a desired frequency is not valid, this function returns 0.

Parameters

Function Documentation

<i>refFreq</i>	PLL reference clock frequency.
<i>desiredFreq</i>	Desired PLL output frequency.
<i>prdiv</i>	PRDIV value to generate desired PLL frequency.
<i>vdiv</i>	VDIV value to generate desired PLL frequency.

Returns

Closest frequency match that the PLL was able generate.

6.7.33 void CLOCK_SetOsc0MonitorMode (mcg_monitor_mode_t mode)

This function sets the OSC0 clock monitor mode. See [mcg_monitor_mode_t](#) for details.

Parameters

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

6.7.34 void CLOCK_SetRtcOscMonitorMode (mcg_monitor_mode_t mode)

This function sets the RTC OSC clock monitor mode. See [mcg_monitor_mode_t](#) for details.

Parameters

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

6.7.35 void CLOCK_SetPll0MonitorMode (mcg_monitor_mode_t mode)

This function sets the PLL0 clock monitor mode. See [mcg_monitor_mode_t](#) for details.

Parameters

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

6.7.36 uint32_t CLOCK_GetStatusFlags (void)

This function gets the MCG clock status flags. All status flags are returned as a logical OR of the enumeration [_mcg_status_flags_t](#). To check a specific flag, compare the return value with the flag.

Example:

```
// To check the clock lost lock status of OSC0 and PLL0.
uint32_t mcgFlags;

mcgFlags = CLOCK_GetStatusFlags();

if (mcgFlags & kMCG_Osc0LostFlag)
{
    // OSC0 clock lock lost. Do something.
}
if (mcgFlags & kMCG_Pll0LostFlag)
{
    // PLL0 clock lock lost. Do something.
}
```

Returns

Logical OR value of the [_mcg_status_flags_t](#).

6.7.37 void CLOCK_ClearStatusFlags (uint32_t *mask*)

This function clears the MCG clock lock lost status. The parameter is a logical OR value of the flags to clear. See [_mcg_status_flags_t](#).

Example:

```
// To clear the clock lost lock status flags of OSC0 and PLL0.
CLOCK_ClearStatusFlags(kMCG_Osc0LostFlag | kMCG_Pll0LostFlag);
```

Parameters

<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration _mcg_status_flags_t .
-------------	---

6.7.38 static void OSC_SetExtRefClkConfig (OSC_Type * *base*, oscr_config_t const * *config*) [inline], [static]

This function configures the OSC external reference clock (OSCERCLK). This is an example to enable the OSCERCLK in normal and stop modes and also set the output divider to 1:

```
oscr_config_t config =
{
    .enableMode = kOSC_ErClkEnable |
                 kOSC_ErClkEnableInStop,
    .erclkDiv    = 1U,
};

OSC_SetExtRefClkConfig(OSC, &config);
```

Function Documentation

Parameters

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

6.7.39 static void OSC_SetCapLoad (OSC_Type * *base*, uint8_t *capLoad*) [inline], [static]

This function sets the specified capacitors configuration for the oscillator. This should be done in the early system level initialization function call based on the system configuration.

Parameters

<i>base</i>	OSC peripheral address.
<i>capLoad</i>	OR'ed value for the capacitor load option, see _osc_cap_load .

Example:

```
// To enable only 2 pF and 8 pF capacitor load, please use like this.  
OSC_SetCapLoad(OSC, kOSC_Cap2P | kOSC_Cap8P);
```

6.7.40 void CLOCK_InitOsc0 (osc_config_t const * *config*)

This function initializes the OSC0 according to the board configuration.

Parameters

<i>config</i>	Pointer to the OSC0 configuration structure.
---------------	--

6.7.41 void CLOCK_DeinitOsc0 (void)

This function deinitializes the OSC0.

6.7.42 static void CLOCK_SetXtal0Freq (uint32_t *freq*) [inline], [static]

Parameters

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

6.7.43 static void CLOCK_SetXtal32Freq (uint32_t *freq*) [inline], [static]

Parameters

<i>freq</i>	The XTAL32/EXTAL32/RTC_CLKIN input clock frequency in Hz.
-------------	---

6.7.44 status_t CLOCK_TrimInternalRefClk (uint32_t *extFreq*, uint32_t *desiredFreq*, uint32_t * *actualFreq*, mcg_atm_select_t *atms*)

This function trims the internal reference clock by using the external clock. If successful, it returns the `kStatus_Success` and the frequency after trimming is received in the parameter `actualFreq`. If an error occurs, the error code is returned.

Parameters

<i>extFreq</i>	External clock frequency, which should be a bus clock.
<i>desiredFreq</i>	Frequency to trim to.
<i>actualFreq</i>	Actual frequency after trimming.
<i>atms</i>	Trim fast or slow internal reference clock.

Return values

<i>kStatus_Success</i>	ATM success.
<i>kStatus_MCG_AtmBus-ClockInvalid</i>	The bus clock is not in allowed range for the ATM.
<i>kStatus_MCG_Atm-DesiredFreqInvalid</i>	MCGIRCLK could not be trimmed to the desired frequency.
<i>kStatus_MCG_AtmIrc-Used</i>	Could not trim because MCGIRCLK is used as a bus clock source.

Function Documentation

<i>kStatus_MCG_Atm-HardwareFail</i>	Hardware fails while trimming.
-------------------------------------	--------------------------------

6.7.45 **mcg_mode_t** CLOCK_GetMode (void)

This function checks the MCG registers and determines the current MCG mode.

Returns

Current MCG mode or error code; See [mcg_mode_t](#).

6.7.46 **status_t** CLOCK_SetFeiMode (mcg_dmx32_t *dmx32*, mcg_drs_t *drs*, void(*)(void) *fllStableDelay*)

This function sets the MCG to FEI mode. If setting to FEI mode fails from the current mode, this function returns an error.

Parameters

<i>dmx32</i>	DMX32 in FEI mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to ensure that the FLL is stable. Passing NULL does not cause a delay.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

Note

If *dmx32* is set to *kMCG_Dmx32Fine*, the slow IRC must not be trimmed to a frequency above 32768 Hz.

6.7.47 **status_t** CLOCK_SetFeeMode (uint8_t *frdiv*, mcg_dmx32_t *dmx32*, mcg_drs_t *drs*, void(*)(void) *fllStableDelay*)

This function sets the MCG to FEE mode. If setting to FEE mode fails from the current mode, this function returns an error.

Parameters

<i>frdiv</i>	FLL reference clock divider setting, FRDIV.
<i>dmx32</i>	DMX32 in FEE mode.
<i>drs</i>	The DCO range selection.
<i>flStableDelay</i>	Delay function to make sure FLL is stable. Passing NULL does not cause a delay.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

6.7.48 **status_t** CLOCK_SetFbiMode (**mcg_dmx32_t** *dmx32*, **mcg_drs_t** *drs*, **void(*)**(**void**) *flStableDelay*)

This function sets the MCG to FBI mode. If setting to FBI mode fails from the current mode, this function returns an error.

Parameters

<i>dmx32</i>	DMX32 in FBI mode.
<i>drs</i>	The DCO range selection.
<i>flStableDelay</i>	Delay function to make sure FLL is stable. If the FLL is not used in FBI mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

Function Documentation

Note

If `dmx32` is set to `kMCG_Dmx32Fine`, the slow IRC must not be trimmed to frequency above 32768 Hz.

6.7.49 **status_t** CLOCK_SetFbeMode (**uint8_t** *frdiv*, **mcg_dmx32_t** *dmx32*, **mcg_drs_t** *drs*, **void(*)**(**void**) *fillStableDelay*)

This function sets the MCG to FBE mode. If setting to FBE mode fails from the current mode, this function returns an error.

Parameters

<i>frdiv</i>	FLL reference clock divider setting, FRDIV.
<i>dmx32</i>	DMX32 in FBE mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to make sure FLL is stable. If the FLL is not used in FBE mode, this parameter can be NULL. Passing NULL does not cause a delay.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

6.7.50 status_t CLOCK_SetBlpiMode (void)

This function sets the MCG to BLPI mode. If setting to BLPI mode fails from the current mode, this function returns an error.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

6.7.51 status_t CLOCK_SetBlpeMode (void)

This function sets the MCG to BLPE mode. If setting to BLPE mode fails from the current mode, this function returns an error.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
-------------------------------------	--------------------------------------

Function Documentation

<i>kStatus_Success</i>	Switched to the target mode successfully.
------------------------	---

6.7.52 **status_t** CLOCK_SetPbeMode (**mcg_pll_clk_select_t** *pllcs*, **mcg_pll_config_t** const * *config*)

This function sets the MCG to PBE mode. If setting to PBE mode fails from the current mode, this function returns an error.

Parameters

<i>pllcs</i>	The PLL selection, PLLCS.
<i>config</i>	Pointer to the PLL configuration.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

Note

1. The parameter *pllcs* selects the PLL. For platforms with only one PLL, the parameter *pllcs* is kept for interface compatibility.
2. The parameter *config* is the PLL configuration structure. On some platforms, it is possible to choose the external PLL directly, which renders the configuration structure not necessary. In this case, pass in NULL. For example: `CLOCK_SetPbeMode(kMCG_OscselOsc, kMCG_PllClkSelExtPll, NULL);`

6.7.53 **status_t** CLOCK_SetPeeMode (**void**)

This function sets the MCG to PEE mode.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
-------------------------------------	--------------------------------------

<i>kStatus_Success</i>	Switched to the target mode successfully.
------------------------	---

Note

This function only changes the CLKS to use the PLL/FLL output. If the PRDIV/VDIV are different than in the PBE mode, set them up in PBE mode and wait. When the clock is stable, switch to PEE mode.

6.7.54 **status_t** CLOCK_ExternalModeToFbeModeQuick (void)

This function switches the MCG from external modes (PEE/PBE/BLPE/FEE) to the FBE mode quickly. The external clock is used as the system clock source and PLL is disabled. However, the FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEE mode to FEI mode:

```
* CLOCK_ExternalModeToFbeModeQuick();
* CLOCK_SetFeiMode(...);
*
```

Return values

<i>kStatus_Success</i>	Switched successfully.
<i>kStatus_MCG_Mode-Invalid</i>	If the current mode is not an external mode, do not call this function.

6.7.55 **status_t** CLOCK_InternalModeToFbiModeQuick (void)

This function switches the MCG from internal modes (PEI/PBI/BLPI/FEI) to the FBI mode quickly. The MCGIRCLK is used as the system clock source and PLL is disabled. However, FLL settings are not configured. This is a lite function with a small code size, which is useful during the mode switch. For example, to switch from PEI mode to FEE mode:

```
* CLOCK_InternalModeToFbiModeQuick();
* CLOCK_SetFeeMode(...);
*
```

Function Documentation

Return values

<i>kStatus_Success</i>	Switched successfully.
<i>kStatus_MCG_Mode-Invalid</i>	If the current mode is not an internal mode, do not call this function.

6.7.56 **status_t CLOCK_BootToFeiMode (mcg_dmx32_t *dmx32*, mcg_drs_t *drs*, void(*)*(void) flStableDelay*)**

This function sets the MCG to FEI mode from the reset mode. It can also be used to set up MCG during system boot up.

Parameters

<i>dmx32</i>	DMX32 in FEI mode.
<i>drs</i>	The DCO range selection.
<i>flStableDelay</i>	Delay function to ensure that the FLL is stable.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

Note

If *dmx32* is set to *kMCG_Dmx32Fine*, the slow IRC must not be trimmed to frequency above 32768 Hz.

6.7.57 **status_t CLOCK_BootToFeeMode (mcg_oscsel_t *oscsel*, uint8_t *frdiv*, mcg_dmx32_t *dmx32*, mcg_drs_t *drs*, void(*)*(void) flStableDelay*)**

This function sets MCG to FEE mode from the reset mode. It can also be used to set up the MCG during system boot up.

Parameters

<i>oscsel</i>	OSC clock select, OSCSEL.
<i>frdiv</i>	FLL reference clock divider setting, FRDIV.
<i>dmx32</i>	DMX32 in FEE mode.
<i>drs</i>	The DCO range selection.
<i>fllStableDelay</i>	Delay function to ensure that the FLL is stable.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

6.7.58 **status_t** CLOCK_BootToBlpiMode (uint8_t *fcrdiv*, mcg_irc_mode_t *ircs*, uint8_t *ircEnableMode*)

This function sets the MCG to BLPI mode from the reset mode. It can also be used to set up the MCG during sytem boot up.

Parameters

<i>fcrdiv</i>	Fast IRC divider, FCRDIV.
<i>ircs</i>	The internal reference clock to select, IRCS.
<i>ircEnableMode</i>	The MCGIRCLK enable mode, OR'ed value of _mcg_ircclk_enable_mode .

Return values

<i>kStatus_MCG_Source-Used</i>	Could not change MCGIRCLK setting.
<i>kStatus_Success</i>	Switched to the target mode successfully.

6.7.59 **status_t** CLOCK_BootToBlpeMode (mcg_oscsel_t *oscsel*)

This function sets the MCG to BLPE mode from the reset mode. It can also be used to set up the MCG during sytem boot up.

Function Documentation

Parameters

<i>oscsel</i>	OSC clock select, MCG_C7[OSCSEL].
---------------	-----------------------------------

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

6.7.60 **status_t** CLOCK_BootToPeeMode (**mcg_oscsel_t** *oscsel*, **mcg_pll_clk_select_t** *pllcs*, **mcg_pll_config_t** *config*)

This function sets the MCG to PEE mode from reset mode. It can also be used to set up the MCG during system boot up.

Parameters

<i>oscsel</i>	OSC clock select, MCG_C7[OSCSEL].
<i>pllcs</i>	The PLL selection, PLLCS.
<i>config</i>	Pointer to the PLL configuration.

Return values

<i>kStatus_MCG_Mode-Unreachable</i>	Could not switch to the target mode.
<i>kStatus_Success</i>	Switched to the target mode successfully.

6.7.61 **status_t** CLOCK_SetMcgConfig (**mcg_config_t** *config*)

This function sets MCG to a target mode defined by the configuration structure. If switching to the target mode fails, this function chooses the correct path.

Parameters

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

Returns

Return `kStatus_Success` if switched successfully; Otherwise, it returns an error code [_mcg_status](#).

Note

If the external clock is used in the target mode, ensure that it is enabled. For example, if the OSC0 is used, set up OSC0 correctly before calling this function.

6.8 Variable Documentation

6.8.1 uint32_t g_xtal0Freq

The XTAL0/EXTAL0 (OSC0) clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetXtal0Freq` to set the value in the clock driver. For example, if XTAL0 is 8 MHz:

```
* CLOCK_InitOsc0(...); // Set up the OSC0
* CLOCK_SetXtal0Freq(8000000); // Set the XTAL0 value to the clock driver.
*
```

This is important for the multicore platforms where only one core needs to set up the OSC0 using the `CLOCK_InitOsc0`. All other cores need to call the `CLOCK_SetXtal0Freq` to get a valid clock frequency.

6.8.2 uint32_t g_xtal32Freq

The XTAL32/EXTAL32/RTC_CLKIN clock frequency in Hz. When the clock is set up, use the function `CLOCK_SetXtal32Freq` to set the value in the clock driver.

This is important for the multicore platforms where only one core needs to set up the clock. All other cores need to call the `CLOCK_SetXtal32Freq` to get a valid clock frequency.

6.9 Multipurpose Clock Generator (MCG)

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

6.9.1 Function description

MCG driver provides these functions:

- Functions to get the MCG clock frequency.
- Functions to configure the MCG clock, such as PLLCLK and MCGIRCLK.
- Functions for the MCG clock lock lost monitor.
- Functions for the OSC configuration.
- Functions for the MCG auto-trim machine.
- Functions for the MCG mode.

6.9.1.1 MCG frequency functions

MCG module provides clocks, such as MCGOUTCLK, MCGIRCLK, MCGFFCLK, MCGFLLCLK and MCGPLLCLK. The MCG driver provides functions to get the frequency of these clocks, such as [CLOCK_GetOutClkFreq\(\)](#), [CLOCK_GetInternalRefClkFreq\(\)](#), [CLOCK_GetFixedFreqClkFreq\(\)](#), [CLOCK_GetFllFreq\(\)](#), [CLOCK_GetPli0Freq\(\)](#), [CLOCK_GetPli1Freq\(\)](#), and [CLOCK_GetExtPliFreq\(\)](#). These functions get the clock frequency based on the current MCG registers.

6.9.1.2 MCG clock configuration

The MCG driver provides functions to configure the internal reference clock (MCGIRCLK), the external reference clock, and MCGPLLCLK.

The function [CLOCK_SetInternalRefClkConfig\(\)](#) configures the MCGIRCLK, including the source and the divider. Do not change MCGIRCLK when the MCG mode is BLPI/FBI/PBI because the MCGIRCLK is used as a system clock in these modes and changing settings makes the system clock unstable.

The function [CLOCK_SetExternalRefClkConfig\(\)](#) configures the external reference clock source (MCG_C7[OSCSEL]). Do not call this function when the MCG mode is BLPE/FBE/PBE/FEE/PEE because the external reference clock is used as a clock source in these modes. Changing the external reference clock source requires at least a 50 microseconds wait. The function [CLOCK_SetExternalRefClkConfig\(\)](#) implements a for loop delay internally. The for loop delay assumes that the system clock is 96 MHz, which ensures at least 50 micro seconds delay. However, when the system clock is slow, the delay time may significantly increase. This for loop count can be optimized for better performance for specific cases.

The MCGPLLCLK is disabled in FBE/FEE/FBI/FEI modes by default. Applications can enable the MCGPLLCLK in these modes using the functions [CLOCK_EnablePli0\(\)](#) and [CLOCK_EnablePli1\(\)](#). To enable the MCGPLLCLK, the PLL reference clock divider (PRDIV) and the PLL VCO divider (VDIV) must be set to a proper value. The function [CLOCK_CalcPliDiv\(\)](#) helps to get the PRDIV/VDIV.

6.9.1.3 MCG clock lock monitor functions

The MCG module monitors the OSC and the PLL clock lock status. The MCG driver provides the functions to set the clock monitor mode, check the clock lost status, and clear the clock lost status.

6.9.1.4 OSC configuration

The MCG is needed together with the OSC module to enable the OSC clock. The function [CLOCK_InitOsc0\(\)](#) [CLOCK_InitOsc1](#) uses the MCG and OSC to initialize the OSC. The OSC should be configured based on the board design.

6.9.1.5 MCG auto-trim machine

The MCG provides an auto-trim machine to trim the MCG internal reference clock based on the external reference clock (BUS clock). During clock trimming, the MCG must not work in FEI/FBI/BLPI/PBI/PEI modes. The function [CLOCK_TrimInternalRefClk\(\)](#) is used for the auto clock trimming.

6.9.1.6 MCG mode functions

The function [CLOCK_GetMcgMode](#) returns the current MCG mode. The MCG can only switch between the neighbouring modes. If the target mode is not current mode's neighbouring mode, the application must choose the proper switch path. For example, to switch to PEE mode from FEI mode, use FEI -> FBE -> PBE -> PEE.

For the MCG modes, the MCG driver provides three kinds of functions:

The first type of functions involve functions [CLOCK_SetXxxMode](#), such as [CLOCK_SetFeiMode\(\)](#). These functions only set the MCG mode from neighbouring modes. If switching to the target mode directly from current mode is not possible, the functions return an error.

The second type of functions are the functions [CLOCK_BootToXxxMode](#), such as [CLOCK_BootToFeiMode\(\)](#). These functions set the MCG to specific modes from reset mode. Because the source mode and target mode are specific, these functions choose the best switch path. The functions are also useful to set up the system clock during boot up.

The third type of functions is the [CLOCK_SetMcgConfig\(\)](#). This function chooses the right path to switch to the target mode. It is easy to use, but introduces a large code size.

Whenever the FLL settings change, there should be a 1 millisecond delay to ensure that the FLL is stable. The function [CLOCK_SetMcgConfig\(\)](#) implements a for loop delay internally to ensure that the FLL is stable. The for loop delay assumes that the system clock is 96 MHz, which ensures at least 1 millisecond delay. However, when the system clock is slow, the delay time may increase significantly. The for loop count can be optimized for better performance according to a specific use case.

Multipurpose Clock Generator (MCG)

6.9.2 Typical use case

The function `CLOCK_SetMcgConfig` is used to switch between any modes. However, this heavy-light function introduces a large code size. This section shows how to use the mode function to implement a quick and light-weight switch between typical specific modes. Note that the step to enable the external clock is not included in the following steps. Enable the corresponding clock before using it as a clock source.

6.9.2.1 Switch between BLPI and FEI

Use case	Steps	Functions
BLPI -> FEI	BLPI -> FBI	<code>CLOCK_InternalModeToFbiModeQuick(...)</code>
	FBI -> FEI	<code>CLOCK_SetFeiMode(...)</code>
	Configure MCGIRCLK if need	<code>CLOCK_SetInternalRefClkConfig(...)</code>
FEI -> BLPI	Configure MCGIRCLK if need	<code>CLOCK_SetInternalRefClkConfig(...)</code>
	FEI -> FBI	<code>CLOCK_SetFbiMode(...)</code> with <code>flStableDelay=NULL</code>
	FBI -> BLPI	<code>CLOCK_SetLowPowerEnable(true)</code>

6.9.2.2 Switch between BLPI and FEE

Use case	Steps	Functions
BLPI -> FEE	BLPI -> FBI	<code>CLOCK_InternalModeToFbiModeQuick(...)</code>
	Change external clock source if need	<code>CLOCK_SetExternalRefClkConfig(...)</code>
	FBI -> FEE	<code>CLOCK_SetFeeMode(...)</code>
FEE -> BLPI	Configure MCGIRCLK if need	<code>CLOCK_SetInternalRefClkConfig(...)</code>
	FEE -> FBI	<code>CLOCK_SetFbiMode(...)</code> with <code>flStableDelay=NULL</code>
	FBI -> BLPI	<code>CLOCK_SetLowPowerEnable(true)</code>

6.9.2.3 Switch between BLPI and PEE

Use case	Steps	Functions
BLPI -> PEE	BLPI -> FBI	CLOCK_InternalModeToFbi-ModeQuick(...)
	Change external clock source if need	CLOCK_SetExternalRefClk-Config(...)
	FBI -> FBE	CLOCK_SetFbeMode(...) // fl-StableDelay=NULL
	FBE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPI	PEE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config(...)
	FBE -> FBI	CLOCK_SetFbiMode(...) with flStableDelay=NULL
	FBI -> BLPI	CLOCK_SetLowPower-Enable(true)

6.9.2.4 Switch between BLPE and PEE

This table applies when using the same external clock source (MCG_C7[OSCSEL]) in BLPE mode and PEE mode.

Use case	Steps	Functions
BLPE -> PEE	BLPE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPE	PEE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)
	FBE -> BLPE	CLOCK_SetLowPower-Enable(true)

If using different external clock sources (MCG_C7[OSCSEL]) in BLPE mode and PEE mode, call the [CLOCK_SetExternalRefClkConfig\(\)](#) in FBI or FEI mode to change the external reference clock.

Use case	Steps	Functions
	BLPE -> FBE	CLOCK_ExternalModeToFbe-ModeQuick(...)

Multipurpose Clock Generator (MCG)

	FBE -> FBI	CLOCK_SetFbiMode(...) with flStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	FBI -> FBE	CLOCK_SetFbeMode(...) with flStableDelay=NULL
	FBE -> PBE	CLOCK_SetPbeMode(...)
	PBE -> PEE	CLOCK_SetPeeMode(...)
PEE -> BLPE	PEE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)
	FBE -> FBI	CLOCK_SetFbiMode(...) with flStableDelay=NULL
	Change source	CLOCK_SetExternalRefClkConfig(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with flStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPowerEnable(true)

6.9.2.5 Switch between BLPE and FEE

This table applies when using the same external clock source (MCG_C7[OSCSEL]) in BLPE mode and FEE mode.

Use case	Steps	Functions
BLPE -> FEE	BLPE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)
	FBE -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	PEE -> FBE	CLOCK_SetPbeMode(...)
	FBE -> BLPE	CLOCK_SetLowPowerEnable(true)

If using different external clock sources (MCG_C7[OSCSEL]) in BLPE mode and FEE mode, call the [CLOCK_SetExternalRefClkConfig\(\)](#) in FBI or FEI mode to change the external reference clock.

Use case	Steps	Functions
BLPE -> FEE	BLPE -> FBE	CLOCK_ExternalModeToFbeModeQuick(...)

Multipurpose Clock Generator (MCG)

	FBE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClk-Config(...)
	FBI -> FEE	CLOCK_SetFeeMode(...)
FEE -> BLPE	FEE -> FBI	CLOCK_SetFbiMode(...) with fllStableDelay=NULL
	Change source	CLOCK_SetExternalRefClk-Config(...)
	PBI -> FBE	CLOCK_SetFbeMode(...) with fllStableDelay=NULL
	FBE -> BLPE	CLOCK_SetLowPower-Enable(true)

6.9.2.6 Switch between BLPI and PEI

Use case	Steps	Functions
BLPI -> PEI	BLPI -> PBI	CLOCK_SetPbiMode(...)
	PBI -> PEI	CLOCK_SetPeiMode(...)
	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config(...)
PEI -> BLPI	Configure MCGIRCLK if need	CLOCK_SetInternalRefClk-Config
	PEI -> FBI	CLOCK_InternalModeToFbi-ModeQuick(...)
	FBI -> BLPI	CLOCK_SetLowPower-Enable(true)

6.9.3 Code Configuration Option

6.9.3.1 MCG_USER_CONFIG_FLL_STABLE_DELAY_EN

When switching to use FLL with function [CLOCK_SetFeiMode\(\)](#) and [CLOCK_SetFeeMode\(\)](#), there is an internal function [CLOCK_FllStableDelay\(\)](#). It is used to delay a few ms so that to wait the FLL to be stable enough. By default, it is implemented in driver code like:

```
#ifndef MCG_USER_CONFIG_FLL_STABLE_DELAY_EN
void CLOCK_FllStableDelay(void)
{
    /*
```

Multipurpose Clock Generator (MCG)

```
    Should wait at least 1ms. Because in these modes, the core clock is 100MHz
    at most, so this function could obtain the 1ms delay.
*/
volatile uint32_t i = 300000;
while (i--)
{
    __NOP();
}
}
#endif /* MCG_USER_CONFIG_FLL_STABLE_DELAY_EN */
```

Once user is willing to create his own delay function, just assert the macro `MCG_USER_CONFIG_FLL_STABLE_DELAY_EN`, and then define function `CLOCK_FllStableDelay` in the application code.

Chapter 7

CMP: Analog Comparator Driver

7.1 Overview

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

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

7.2 Typical use case

7.2.1 Polling Configuration

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

    // ...

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

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

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

Typical use case

7.2.2 Interrupt Configuration

```
volatile uint32_t g_CmpFlags = 0U;

// ...

void DEMO_CMP_IRQ_HANDLER_FUNC(void)
{
    g_CmpFlags = CMP_GetStatusFlags(DEMO_CMP_INSTANCE);
    CMP_ClearStatusFlags(DEMO_CMP_INSTANCE, kCMP_OutputRisingEventFlag |
        kCMP_OutputFallingEventFlag);
    if (0U != (g_CmpFlags & kCMP_OutputRisingEventFlag))
    {
        // Do something.
    }
    else if (0U != (g_CmpFlags & kCMP_OutputFallingEventFlag))
    {
        // Do something.
    }
}

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

    // ...
    EnableIRQ(DEMO_CMP_IRQ_ID);
    // ...

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

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

    // Enables the output rising and falling interrupts.
    CMP_EnableInterrupts(DEMO_CMP_INSTANCE,
        kCMP_OutputRisingInterruptEnable |
        kCMP_OutputFallingInterruptEnable);

    while (1)
    {
    }
}
```

Data Structures

- struct `cmp_config_t`
Configures the comparator. [More...](#)
- struct `cmp_filter_config_t`
Configures the filter. [More...](#)
- struct `cmp_dac_config_t`
Configures the internal DAC. [More...](#)

Enumerations

- enum `_cmp_interrupt_enable` {
`kCMP_OutputRisingInterruptEnable` = `CMP_SCR_IER_MASK`,
`kCMP_OutputFallingInterruptEnable` = `CMP_SCR_IEF_MASK` }
Interrupt enable/disable mask.
- enum `_cmp_status_flags` {
`kCMP_OutputRisingEventFlag` = `CMP_SCR_CFR_MASK`,
`kCMP_OutputFallingEventFlag` = `CMP_SCR_CFF_MASK`,
`kCMP_OutputAssertEventFlag` = `CMP_SCR_COUT_MASK` }
Status flags' mask.
- enum `cmp_hysteresis_mode_t` {
`kCMP_HysteresisLevel0` = 0U,
`kCMP_HysteresisLevel1` = 1U,
`kCMP_HysteresisLevel2` = 2U,
`kCMP_HysteresisLevel3` = 3U }
CMP Hysteresis mode.
- enum `cmp_reference_voltage_source_t` {
`kCMP_VrefSourceVin1` = 0U,
`kCMP_VrefSourceVin2` = 1U }
CMP Voltage Reference source.

Driver version

- #define `FSL_CMP_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 0))
CMP driver version 2.0.0.

Initialization

- void `CMP_Init` (`CMP_Type` *base, const `cmp_config_t` *config)
Initializes the CMP.
- void `CMP_Deinit` (`CMP_Type` *base)
De-initializes the CMP module.
- static void `CMP_Enable` (`CMP_Type` *base, bool enable)
Enables/disables the CMP module.
- void `CMP_GetDefaultConfig` (`cmp_config_t` *config)
Initializes the CMP user configuration structure.
- void `CMP_SetInputChannels` (`CMP_Type` *base, uint8_t positiveChannel, uint8_t negativeChannel)
Sets the input channels for the comparator.

Advanced Features

- void `CMP_SetFilterConfig` (`CMP_Type` *base, const `cmp_filter_config_t` *config)
Configures the filter.
- void `CMP_SetDACConfig` (`CMP_Type` *base, const `cmp_dac_config_t` *config)
Configures the internal DAC.
- void `CMP_EnableInterrupts` (`CMP_Type` *base, uint32_t mask)
Enables the interrupts.
- void `CMP_DisableInterrupts` (`CMP_Type` *base, uint32_t mask)
Disables the interrupts.

Results

- uint32_t [CMP_GetStatusFlags](#) (CMP_Type *base)
Gets the status flags.
- void [CMP_ClearStatusFlags](#) (CMP_Type *base, uint32_t mask)
Clears the status flags.

7.3 Data Structure Documentation

7.3.1 struct cmp_config_t

Data Fields

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

7.3.1.0.0.9 Field Documentation

7.3.1.0.0.9.1 bool cmp_config_t::enableCmp

7.3.1.0.0.9.2 cmp_hysteresis_mode_t cmp_config_t::hysteresisMode

7.3.1.0.0.9.3 bool cmp_config_t::enableHighSpeed

7.3.1.0.0.9.4 bool cmp_config_t::enableInvertOutput

7.3.1.0.0.9.5 bool cmp_config_t::useUnfilteredOutput

7.3.1.0.0.9.6 bool cmp_config_t::enablePinOut

7.3.2 struct cmp_filter_config_t

Data Fields

- uint8_t [filterCount](#)
Filter Sample Count.
- uint8_t [filterPeriod](#)
Filter Sample Period.

7.3.2.0.0.10 Field Documentation

7.3.2.0.0.10.1 uint8_t cmp_filter_config_t::filterCount

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

7.3.2.0.0.10.2 uint8_t cmp_filter_config_t::filterPeriod

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

7.3.3 struct cmp_dac_config_t

Data Fields

- [cmp_reference_voltage_source_t referenceVoltageSource](#)
Supply voltage reference source.
- uint8_t [DACValue](#)
Value for the DAC Output Voltage.

7.3.3.0.0.11 Field Documentation

7.3.3.0.0.11.1 cmp_reference_voltage_source_t cmp_dac_config_t::referenceVoltageSource

7.3.3.0.0.11.2 uint8_t cmp_dac_config_t::DACValue

Available range is 0-63.

7.4 Macro Definition Documentation

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

7.5 Enumeration Type Documentation

7.5.1 enum _cmp_interrupt_enable

Enumerator

- kCMP_OutputRisingInterruptEnable*** Comparator interrupt enable rising.
kCMP_OutputFallingInterruptEnable Comparator interrupt enable falling.

7.5.2 enum _cmp_status_flags

Enumerator

- kCMP_OutputRisingEventFlag*** Rising-edge on the comparison output has occurred.
kCMP_OutputFallingEventFlag Falling-edge on the comparison output has occurred.

Function Documentation

kCMP_OutputAssertEventFlag Return the current value of the analog comparator output.

7.5.3 enum cmp_hysteresis_mode_t

Enumerator

kCMP_HysteresisLevel0 Hysteresis level 0.
kCMP_HysteresisLevel1 Hysteresis level 1.
kCMP_HysteresisLevel2 Hysteresis level 2.
kCMP_HysteresisLevel3 Hysteresis level 3.

7.5.4 enum cmp_reference_voltage_source_t

Enumerator

kCMP_VrefSourceVin1 Vin1 is selected as a resistor ladder network supply reference Vin.
kCMP_VrefSourceVin2 Vin2 is selected as a resistor ladder network supply reference Vin.

7.6 Function Documentation

7.6.1 void CMP_Init (CMP_Type * *base*, const cmp_config_t * *config*)

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

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

Parameters

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

7.6.2 void CMP_Deinit (CMP_Type * *base*)

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

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

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

Function Documentation

Parameters

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

7.6.3 static void CMP_Enable (CMP_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

7.6.4 void CMP_GetDefaultConfig (cmp_config_t * *config*)

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

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

Parameters

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

7.6.5 void CMP_SetInputChannels (CMP_Type * *base*, uint8_t *positiveChannel*, uint8_t *negativeChannel*)

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

Parameters

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

7.6.6 void CMP_SetFilterConfig (CMP_Type * *base*, const cmp_filter_config_t * *config*)

Parameters

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

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

Parameters

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

7.6.8 void CMP_EnableInterrupts (CMP_Type * *base*, uint32_t *mask*)

Parameters

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

7.6.9 void CMP_DisableInterrupts (CMP_Type * *base*, uint32_t *mask*)

Function Documentation

Parameters

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

7.6.10 uint32_t CMP_GetStatusFlags (CMP_Type * *base*)

Parameters

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

Returns

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

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

Parameters

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

Chapter 8

CMT: Carrier Modulator Transmitter Driver

8.1 Overview

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

8.2 Clock formulas

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

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

8.3 Typical use case

This is an example code to initialize data.

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

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

CMT_GetDefaultConfig(&config);

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

Typical use case

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

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

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

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

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

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

Data Structures

- struct `cmt_modulate_config_t`
CMT carrier generator and modulator configuration structure. [More...](#)
- struct `cmt_config_t`
CMT basic configuration structure. [More...](#)

Enumerations

- enum `cmt_mode_t` {
 `kCMT_DirectIROCtl` = 0x00U,
 `kCMT_TimeMode` = 0x01U,
 `kCMT_FSKMode` = 0x05U,
 `kCMT_BasebandMode` = 0x09U }
The modes of CMT.
- enum `cmt_primary_clkdiv_t` {

```

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

```

The CMT clock divide primary prescaler.

- enum `cmt_second_clkdiv_t` {
`kCMT_SecondClkDiv1 = 0U,`
`kCMT_SecondClkDiv2 = 1U,`
`kCMT_SecondClkDiv4 = 2U,`
`kCMT_SecondClkDiv8 = 3U }`

The CMT clock divide secondary prescaler.

- enum `cmt_infrared_output_polarity_t` {
`kCMT_IROActiveLow = 0U,`
`kCMT_IROActiveHigh = 1U }`

The CMT infrared output polarity.

- enum `cmt_infrared_output_state_t` {
`kCMT_IROCtrlLow = 0U,`
`kCMT_IROCtrlHigh = 1U }`

The CMT infrared output signal state control.

- enum `_cmt_interrupt_enable` { `kCMT_EndOfCycleInterruptEnable = CMT_MSC_EOCIE_MASK`
}

CMT interrupt configuration structure, default settings all disabled.

Driver version

- #define `FSL_CMT_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 1)`)
CMT driver version 2.0.1.

Initialization and deinitialization

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

Disables the CMT module and gate control.

Basic Control Operations

- void [CMT_SetMode](#) (CMT_Type *base, [cmt_mode_t](#) mode, [cmt_modulate_config_t](#) *modulate-Config)
Selects the mode for CMT.
- [cmt_mode_t](#) [CMT_GetMode](#) (CMT_Type *base)
Gets the mode of the CMT module.
- uint32_t [CMT_GetCMTFrequency](#) (CMT_Type *base, uint32_t busClock_Hz)
Gets the actual CMT clock frequency.
- static void [CMT_SetCarrirGenerateCountOne](#) (CMT_Type *base, uint32_t highCount, uint32_t lowCount)
Sets the primary data set for the CMT carrier generator counter.
- static void [CMT_SetCarrirGenerateCountTwo](#) (CMT_Type *base, uint32_t highCount, uint32_t lowCount)
Sets the secondary data set for the CMT carrier generator counter.
- void [CMT_SetModulateMarkSpace](#) (CMT_Type *base, uint32_t markCount, uint32_t spaceCount)
Sets the modulation mark and space time period for the CMT modulator.
- static void [CMT_EnableExtendedSpace](#) (CMT_Type *base, bool enable)
Enables or disables the extended space operation.
- void [CMT_SetIroState](#) (CMT_Type *base, [cmt_infrared_output_state_t](#) state)
Sets the IRO (infrared output) signal state.
- static void [CMT_EnableInterrupts](#) (CMT_Type *base, uint32_t mask)
Enables the CMT interrupt.
- static void [CMT_DisableInterrupts](#) (CMT_Type *base, uint32_t mask)
Disables the CMT interrupt.
- static uint32_t [CMT_GetStatusFlags](#) (CMT_Type *base)
Gets the end of the cycle status flag.

8.4 Data Structure Documentation

8.4.1 struct cmt_modulate_config_t

Data Fields

- uint8_t [highCount1](#)
The high-time for carrier generator first register.
- uint8_t [lowCount1](#)
The low-time for carrier generator first register.
- uint8_t [highCount2](#)
The high-time for carrier generator second register for FSK mode.
- uint8_t [lowCount2](#)
The low-time for carrier generator second register for FSK mode.
- uint16_t [markCount](#)
The mark time for the modulator gate.
- uint16_t [spaceCount](#)
The space time for the modulator gate.

8.4.1.0.0.12 Field Documentation

8.4.1.0.0.12.1 `uint8_t cmt_modulate_config_t::highCount1`

8.4.1.0.0.12.2 `uint8_t cmt_modulate_config_t::lowCount1`

8.4.1.0.0.12.3 `uint8_t cmt_modulate_config_t::highCount2`

8.4.1.0.0.12.4 `uint8_t cmt_modulate_config_t::lowCount2`

8.4.1.0.0.12.5 `uint16_t cmt_modulate_config_t::markCount`

8.4.1.0.0.12.6 `uint16_t cmt_modulate_config_t::spaceCount`

8.4.2 struct cmt_config_t

Data Fields

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

8.4.2.0.0.13 Field Documentation

8.4.2.0.0.13.1 `bool cmt_config_t::isInterruptEnabled`

8.4.2.0.0.13.2 `bool cmt_config_t::isIroEnabled`

8.4.2.0.0.13.3 `cmt_infrared_output_polarity_t cmt_config_t::iroPolarity`

8.4.2.0.0.13.4 `cmt_second_clkdiv_t cmt_config_t::divider`

8.5 Macro Definition Documentation

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

8.6 Enumeration Type Documentation

8.6.1 enum cmt_mode_t

Enumerator

kCMT_DirectIROCtl Carrier modulator is disabled and the IRO signal is directly in software control.

kCMT_TimeMode Carrier modulator is enabled in time mode.

Enumeration Type Documentation

kCMT_FSKMode Carrier modulator is enabled in FSK mode.

kCMT_BasebandMode Carrier modulator is enabled in baseband mode.

8.6.2 enum cmt_primary_clkdiv_t

The primary clock divider is used to divider the bus clock to get the intermediate frequency to approximately equal to 8 MHZ. When the bus clock is 8 MHZ, set primary prescaler to "kCMT_PrimaryClkDiv1".

Enumerator

<i>kCMT_PrimaryClkDiv1</i>	The intermediate frequency is the bus clock divided by 1.
<i>kCMT_PrimaryClkDiv2</i>	The intermediate frequency is the bus clock divided by 2.
<i>kCMT_PrimaryClkDiv3</i>	The intermediate frequency is the bus clock divided by 3.
<i>kCMT_PrimaryClkDiv4</i>	The intermediate frequency is the bus clock divided by 4.
<i>kCMT_PrimaryClkDiv5</i>	The intermediate frequency is the bus clock divided by 5.
<i>kCMT_PrimaryClkDiv6</i>	The intermediate frequency is the bus clock divided by 6.
<i>kCMT_PrimaryClkDiv7</i>	The intermediate frequency is the bus clock divided by 7.
<i>kCMT_PrimaryClkDiv8</i>	The intermediate frequency is the bus clock divided by 8.
<i>kCMT_PrimaryClkDiv9</i>	The intermediate frequency is the bus clock divided by 9.
<i>kCMT_PrimaryClkDiv10</i>	The intermediate frequency is the bus clock divided by 10.
<i>kCMT_PrimaryClkDiv11</i>	The intermediate frequency is the bus clock divided by 11.
<i>kCMT_PrimaryClkDiv12</i>	The intermediate frequency is the bus clock divided by 12.
<i>kCMT_PrimaryClkDiv13</i>	The intermediate frequency is the bus clock divided by 13.
<i>kCMT_PrimaryClkDiv14</i>	The intermediate frequency is the bus clock divided by 14.
<i>kCMT_PrimaryClkDiv15</i>	The intermediate frequency is the bus clock divided by 15.
<i>kCMT_PrimaryClkDiv16</i>	The intermediate frequency is the bus clock divided by 16.

8.6.3 enum cmt_second_clkdiv_t

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

Enumerator

<i>kCMT_SecondClkDiv1</i>	The CMT clock is the intermediate frequency frequency divided by 1.
<i>kCMT_SecondClkDiv2</i>	The CMT clock is the intermediate frequency frequency divided by 2.
<i>kCMT_SecondClkDiv4</i>	The CMT clock is the intermediate frequency frequency divided by 4.
<i>kCMT_SecondClkDiv8</i>	The CMT clock is the intermediate frequency frequency divided by 8.

8.6.4 enum cmt_infrared_output_polarity_t

Enumerator

kCMT_IROActiveLow The CMT infrared output signal polarity is active-low.

kCMT_IROActiveHigh The CMT infrared output signal polarity is active-high.

8.6.5 enum cmt_infrared_output_state_t

Enumerator

kCMT_IROctlLow The CMT Infrared output signal state is controlled to low.

kCMT_IROctlHigh The CMT Infrared output signal state is controlled to high.

8.6.6 enum _cmt_interrupt_enable

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

Enumerator

kCMT_EndOfCycleInterruptEnable CMT end of cycle interrupt.

8.7 Function Documentation

8.7.1 void CMT_GetDefaultConfig (cmt_config_t * config)

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

Parameters

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

8.7.2 void CMT_Init (CMT_Type * base, const cmt_config_t * config, uint32_t busClock_Hz)

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

Function Documentation

Parameters

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

8.7.3 void CMT_Deinit (CMT_Type * *base*)

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

Parameters

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

8.7.4 void CMT_SetMode (CMT_Type * *base*, cmt_mode_t *mode*, cmt_modulate_config_t * *modulateConfig*)

Parameters

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

8.7.5 cmt_mode_t CMT_GetMode (CMT_Type * *base*)

Parameters

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

Returns

The CMT mode. kCMT_DirectIROctl Carrier modulator is disabled; the IRO signal is directly in software control. kCMT_TimeMode Carrier modulator is enabled in time mode. kCMT_FSKMode Carrier modulator is enabled in FSK mode. kCMT_BasebandMode Carrier modulator is enabled in baseband mode.

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

Function Documentation

Parameters

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

Returns

The CMT clock frequency.

8.7.7 static void CMT_SetCarrirGenerateCountOne (CMT_Type * *base*, uint32_t *highCount*, uint32_t *lowCount*) [inline], [static]

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

Parameters

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

8.7.8 static void CMT_SetCarrirGenerateCountTwo (CMT_Type * *base*, uint32_t *highCount*, uint32_t *lowCount*) [inline], [static]

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

Parameters

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

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

8.7.9 void CMT_SetModulateMarkSpace (CMT_Type * *base*, uint32_t *markCount*, uint32_t *spaceCount*)

This function sets the mark time period of the CMT modulator counter to control the mark time of the output modulated signal from the carrier generator output signal. If the CMT clock frequency is F_{cmt} and the carrier out signal frequency is f_{cg} :

- In Time and Baseband mode: The mark period of the generated signal equals $(markCount + 1) / (F_{cmt}/8)$. The space period of the generated signal equals $spaceCount / (F_{cmt}/8)$.
- In FSK mode: The mark period of the generated signal equals $(markCount + 1)/f_{cg}$. The space period of the generated signal equals $spaceCount / f_{cg}$.

Parameters

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

8.7.10 static void CMT_EnableExtendedSpace (CMT_Type * *base*, bool *enable*) [inline], [static]

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

Parameters

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

8.7.11 void CMT_SetIroState (CMT_Type * *base*, cmt_infrared_output_state_t *state*)

Changes the states of the IRO signal when the kCMT_DirectIROMode mode is set and the IRO signal is enabled.

Parameters

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

8.7.12 static void CMT_EnableInterrupts (CMT_Type * *base*, uint32_t *mask*) [inline], [static]

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

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

Parameters

<i>base</i>	CMT peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of _cmt_interrupt_enable .

8.7.13 static void CMT_DisableInterrupts (CMT_Type * *base*, uint32_t *mask*) [inline], [static]

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

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

Parameters

<i>base</i>	CMT peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of _cmt_interrupt_enable .

8.7.14 static uint32_t CMT_GetStatusFlags (CMT_Type * *base*) [inline], [static]

The flag is set:

Function Documentation

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

Parameters

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

Returns

Current status of the end of cycle status flag

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

Chapter 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 Driver Examples

[CRC_Init\(\)](#) / [CRC_WriteData\(\)](#) / [CRC_Get16bitResult\(\)](#) to get an intermediate checksum.

[CRC_Init\(\)](#) / [CRC_WriteData\(\)](#) / ... / [CRC_WriteData\(\)](#) / [CRC_Get16bitResult\(\)](#) to get an intermediate checksum.

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.

```
CRC_Module_RTOS_Mutex_Lock;  
CRC_Init();  
CRC_WriteData();  
CRC_Get16bitResult();  
CRC_Module_RTOS_Mutex_Unlock;
```

9.6 Comments about API usage in interrupt handler

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

9.7 CRC Driver Examples

9.7.1 Simple examples

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

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

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

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



```

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

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

```

9.7.2 Advanced examples

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

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

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

These are the corresponding initialization functions.

```

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

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

    CRC_Init(base, &config);
}

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

```

CRC Driver Examples

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

    CRC_Init(base, &config);
}

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

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

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

The following context switches show a possible API usage.

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

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

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

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

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

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

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

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

```

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

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

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

```

Data Structures

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

Macros

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

Enumerations

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

Functions

- void `CRC_Init` (CRC_Type *base, const `crc_config_t` *config)
 Enables and configures the CRC peripheral module.
- static void `CRC_Deinit` (CRC_Type *base)
 Disables the CRC peripheral module.
- void `CRC_GetDefaultConfig` (`crc_config_t` *config)
 Loads default values to the CRC protocol configuration structure.
- void `CRC_WriteData` (CRC_Type *base, const uint8_t *data, size_t dataSize)
 Writes data to the CRC module.
- uint32_t `CRC_Get32bitResult` (CRC_Type *base)
 Reads the 32-bit checksum from the CRC module.
- uint16_t `CRC_Get16bitResult` (CRC_Type *base)
 Reads a 16-bit checksum from the CRC module.

Driver version

- #define `FSL_CRC_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 1))
 CRC driver version.

Macro Definition Documentation

9.8 Data Structure Documentation

9.8.1 struct crc_config_t

This structure holds the configuration for the CRC protocol.

Data Fields

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

9.8.1.0.0.14 Field Documentation

9.8.1.0.0.14.1 uint32_t crc_config_t::polynomial

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

9.8.1.0.0.14.2 bool crc_config_t::reflectIn

9.8.1.0.0.14.3 bool crc_config_t::reflectOut

9.8.1.0.0.14.4 bool crc_config_t::complementChecksum

9.8.1.0.0.14.5 crc_bits_t crc_config_t::crcBits

9.9 Macro Definition Documentation

9.9.1 #define FSL_CRC_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

Version 2.0.1.

Current version: 2.0.1

Change log:

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

9.9.2 #define CRC_DRIVER_USE_CRC16_CCIT_FALSE_AS_DEFAULT 1

Use CRC16-CCIT-FALSE as default.

9.10 Enumeration Type Documentation

9.10.1 enum crc_bits_t

Enumerator

kCrcBits16 Generate 16-bit CRC code.

kCrcBits32 Generate 32-bit CRC code.

9.10.2 enum crc_result_t

Enumerator

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

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

9.11 Function Documentation

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

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

Parameters

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

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

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

Function Documentation

Parameters

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

9.11.3 void CRC_GetDefaultConfig (crc_config_t * *config*)

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

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

Parameters

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

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

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

Parameters

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

9.11.5 uint32_t CRC_Get32bitResult (CRC_Type * *base*)

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

Parameters

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

Returns

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

9.11.6 uint16_t CRC_Get16bitResult (CRC_Type * *base*)

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

Parameters

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

Returns

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

Chapter 10

DMAMUX: Direct Memory Access Multiplexer Driver

10.1 Overview

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

10.2 Typical use case

10.2.1 DMAMUX Operation

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

Driver version

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

DMAMUX Initialization and de-initialization

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

DMAMUX Channel Operation

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

10.3 Macro Definition Documentation

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

10.4 Function Documentation

10.4.1 void DMAMUX_Init (DMAMUX_Type * *base*)

This function ungates the DMAMUX clock.

Parameters

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

10.4.2 void DMAMUX_Deinit (DMAMUX_Type * *base*)

This function gates the DMAMUX clock.

Parameters

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

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

This function enables the DMAMUX channel.

Parameters

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

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

This function disables the DMAMUX channel.

Note

The user must disable the DMAMUX channel before configuring it.

Parameters

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

Function Documentation

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

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

Parameters

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



Chapter 11

DSPI: Serial Peripheral Interface Driver

11.1 Overview

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

Modules

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

DSPI Driver

11.2 DSPI Driver

11.2.1 Overview

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

11.2.2 Typical use case

11.2.2.1 Master Operation

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

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

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

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

11.2.2.2 Slave Operation

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

```

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

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

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

DSPI_SlaveTransferNonBlocking(&g_s_handle, &slaveXfer);

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

```

Data Structures

- struct [dspi_command_data_config_t](#)
DSPI master command data configuration used for the SPIx_PUSHR. [More...](#)
- struct [dspi_master_ctar_config_t](#)
DSPI master ctar configuration structure. [More...](#)
- struct [dspi_master_config_t](#)
DSPI master configuration structure. [More...](#)
- struct [dspi_slave_ctar_config_t](#)
DSPI slave ctar configuration structure. [More...](#)
- struct [dspi_slave_config_t](#)
DSPI slave configuration structure. [More...](#)
- struct [dspi_transfer_t](#)
DSPI master/slave transfer structure. [More...](#)
- struct [dspi_master_handle_t](#)
DSPI master transfer handle structure used for transactional API. [More...](#)
- struct [dspi_slave_handle_t](#)
DSPI slave transfer handle structure used for the transactional API. [More...](#)

Macros

- #define [DSPI_DUMMY_DATA](#) (0x00U)

DSPI Driver

- DSPI dummy data if there is no Tx data.*
- #define [DSPI_MASTER_CTAR_SHIFT](#) (0U)
DSPI master CTAR shift macro; used internally.
- #define [DSPI_MASTER_CTAR_MASK](#) (0x0FU)
DSPI master CTAR mask macro; used internally.
- #define [DSPI_MASTER_PCS_SHIFT](#) (4U)
DSPI master PCS shift macro; used internally.
- #define [DSPI_MASTER_PCS_MASK](#) (0xF0U)
DSPI master PCS mask macro; used internally.
- #define [DSPI_SLAVE_CTAR_SHIFT](#) (0U)
DSPI slave CTAR shift macro; used internally.
- #define [DSPI_SLAVE_CTAR_MASK](#) (0x07U)
DSPI slave CTAR mask macro; used internally.

Typedefs

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

Enumerations

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


```

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

```

DSPI interrupt source.

- enum `_dspi_dma_enable` {
`kDSPI_TxDmaEnable` = (SPI_RSER_TFFF_RE_MASK | SPI_RSER_TFFF_DIRS_MASK),
`kDSPI_RxDmaEnable` = (SPI_RSER_RFDF_RE_MASK | SPI_RSER_RFDF_DIRS_MASK) }

DSPI DMA source.

- enum `dspi_master_slave_mode_t` {
`kDSPI_Master` = 1U,
`kDSPI_Slave` = 0U }

DSPI master or slave mode configuration.

- enum `dspi_master_sample_point_t` {
`kDSPI_SckToSin0Clock` = 0U,
`kDSPI_SckToSin1Clock` = 1U,
`kDSPI_SckToSin2Clock` = 2U }

DSPI Sample Point: Controls when the DSPI master samples SIN in the Modified Transfer Format.

- enum `dspi_which_pcs_t` {
`kDSPI_Pcs0` = 1U << 0,
`kDSPI_Pcs1` = 1U << 1,
`kDSPI_Pcs2` = 1U << 2,
`kDSPI_Pcs3` = 1U << 3,
`kDSPI_Pcs4` = 1U << 4,
`kDSPI_Pcs5` = 1U << 5 }

DSPI Peripheral Chip Select (Pcs) configuration (which Pcs to configure).

- enum `dspi_pcs_polarity_config_t` {
`kDSPI_PcsActiveHigh` = 0U,
`kDSPI_PcsActiveLow` = 1U }

DSPI Peripheral Chip Select (Pcs) Polarity configuration.

- enum `_dspi_pcs_polarity` {
`kDSPI_Pcs0ActiveLow` = 1U << 0,
`kDSPI_Pcs1ActiveLow` = 1U << 1,
`kDSPI_Pcs2ActiveLow` = 1U << 2,
`kDSPI_Pcs3ActiveLow` = 1U << 3,
`kDSPI_Pcs4ActiveLow` = 1U << 4,
`kDSPI_Pcs5ActiveLow` = 1U << 5,
`kDSPI_PcsAllActiveLow` = 0xFFU }

DSPI Peripheral Chip Select (Pcs) Polarity.

- enum `dspi_clock_polarity_t` {
`kDSPI_ClockPolarityActiveHigh` = 0U,
`kDSPI_ClockPolarityActiveLow` = 1U }

DSPI clock polarity configuration for a given CTAR.

- enum `dspi_clock_phase_t` {

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

DSPI clock phase configuration for a given CTAR.

- enum `dspi_shift_direction_t` {
 `kDSPI_MsbFirst` = 0U,
 `kDSPI_LsbFirst` = 1U }

DSPI data shifter direction options for a given CTAR.

- enum `dspi_delay_type_t` {
 `kDSPI_PcsToSck` = 1U,
 `kDSPI_LastSckToPcs`,
 `kDSPI_BetweenTransfer` }

DSPI delay type selection.

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

DSPI Clock and Transfer Attributes Register (CTAR) selection.

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

Use this enumeration for the DSPI master transfer configFlags.

- enum `_dspi_transfer_config_flag_for_slave` { `kDSPI_SlaveCtar0` = 0U << DSPI_SLAVE_CTAR_SHIFT }

Use this enumeration for the DSPI slave transfer configFlags.

- enum `_dspi_transfer_state` {
 `kDSPI_Idle` = 0x0U,
 `kDSPI_Busy`,

`kDSPI_Error }`

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

Driver version

- #define `FSL_DSPI_DRIVER_VERSION` (`MAKE_VERSION(2, 1, 4)`)
DSPI driver version 2.1.4.

Initialization and deinitialization

- void `DSPI_MasterInit` (`SPI_Type *base`, const `dspi_master_config_t *masterConfig`, `uint32_t srcClock_Hz`)
Initializes the DSPI master.
- void `DSPI_MasterGetDefaultConfig` (`dspi_master_config_t *masterConfig`)
Sets the `dspi_master_config_t` structure to default values.
- void `DSPI_SlaveInit` (`SPI_Type *base`, const `dspi_slave_config_t *slaveConfig`)
DSPI slave configuration.
- void `DSPI_SlaveGetDefaultConfig` (`dspi_slave_config_t *slaveConfig`)
Sets the `dspi_slave_config_t` structure to a default value.
- void `DSPI_Deinit` (`SPI_Type *base`)
De-initializes the DSPI peripheral.
- static void `DSPI_Enable` (`SPI_Type *base`, bool enable)
Enables the DSPI peripheral and sets the MCR MDIS to 0.

Status

- static `uint32_t DSPI_GetStatusFlags` (`SPI_Type *base`)
Gets the DSPI status flag state.
- static void `DSPI_ClearStatusFlags` (`SPI_Type *base`, `uint32_t statusFlags`)
Clears the DSPI status flag.

Interrupts

- void `DSPI_EnableInterrupts` (`SPI_Type *base`, `uint32_t mask`)
Enables the DSPI interrupts.
- static void `DSPI_DisableInterrupts` (`SPI_Type *base`, `uint32_t mask`)
Disables the DSPI interrupts.

DMA Control

- static void `DSPI_EnableDMA` (`SPI_Type *base`, `uint32_t mask`)
Enables the DSPI DMA request.
- static void `DSPI_DisableDMA` (`SPI_Type *base`, `uint32_t mask`)
Disables the DSPI DMA request.

DSPI Driver

- static uint32_t [DSPI_MasterGetTxRegisterAddress](#) (SPI_Type *base)
Gets the DSPI master PUSHR data register address for the DMA operation.
- static uint32_t [DSPI_SlaveGetTxRegisterAddress](#) (SPI_Type *base)
Gets the DSPI slave PUSHR data register address for the DMA operation.
- static uint32_t [DSPI_GetRxRegisterAddress](#) (SPI_Type *base)
Gets the DSPI POPR data register address for the DMA operation.

Bus Operations

- static void [DSPI_SetMasterSlaveMode](#) (SPI_Type *base, [dspi_master_slave_mode_t](#) mode)
Configures the DSPI for master or slave.
- static bool [DSPI_IsMaster](#) (SPI_Type *base)
Returns whether the DSPI module is in master mode.
- static void [DSPI_StartTransfer](#) (SPI_Type *base)
Starts the DSPI transfers and clears HALT bit in MCR.
- static void [DSPI_StopTransfer](#) (SPI_Type *base)
Stops DSPI transfers and sets the HALT bit in MCR.
- static void [DSPI_SetFifoEnable](#) (SPI_Type *base, bool enableTxFifo, bool enableRxFifo)
Enables or disables the DSPI FIFOs.
- static void [DSPI_FlushFifo](#) (SPI_Type *base, bool flushTxFifo, bool flushRxFifo)
Flushes the DSPI FIFOs.
- static void [DSPI_SetAllPcsPolarity](#) (SPI_Type *base, uint32_t mask)
Configures the DSPI peripheral chip select polarity simultaneously.
- uint32_t [DSPI_MasterSetBaudRate](#) (SPI_Type *base, [dspi_ctar_selection_t](#) whichCtar, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
Sets the DSPI baud rate in bits per second.
- void [DSPI_MasterSetDelayScaler](#) (SPI_Type *base, [dspi_ctar_selection_t](#) whichCtar, uint32_t prescaler, uint32_t scaler, [dspi_delay_type_t](#) whichDelay)
Manually configures the delay prescaler and scaler for a particular CTAR.
- uint32_t [DSPI_MasterSetDelayTimes](#) (SPI_Type *base, [dspi_ctar_selection_t](#) whichCtar, [dspi_delay_type_t](#) whichDelay, uint32_t srcClock_Hz, uint32_t delayTimeInNanoSec)
Calculates the delay prescaler and scaler based on the desired delay input in nanoseconds.
- static void [DSPI_MasterWriteData](#) (SPI_Type *base, [dspi_command_data_config_t](#) *command, uint16_t data)
Writes data into the data buffer for master mode.
- void [DSPI_GetDefaultDataCommandConfig](#) ([dspi_command_data_config_t](#) *command)
Sets the [dspi_command_data_config_t](#) structure to default values.
- void [DSPI_MasterWriteDataBlocking](#) (SPI_Type *base, [dspi_command_data_config_t](#) *command, uint16_t data)
Writes data into the data buffer master mode and waits till complete to return.
- static uint32_t [DSPI_MasterGetFormattedCommand](#) ([dspi_command_data_config_t](#) *command)
Returns the DSPI command word formatted to the PUSHR data register bit field.
- void [DSPI_MasterWriteCommandDataBlocking](#) (SPI_Type *base, uint32_t data)
Writes a 32-bit data word (16-bit command appended with 16-bit data) into the data buffer master mode and waits till complete to return.
- static void [DSPI_SlaveWriteData](#) (SPI_Type *base, uint32_t data)
Writes data into the data buffer in slave mode.
- void [DSPI_SlaveWriteDataBlocking](#) (SPI_Type *base, uint32_t data)
Writes data into the data buffer in slave mode, waits till data was transmitted, and returns.

- static uint32_t **DSPI_ReadData** (SPI_Type *base)
Reads data from the data buffer.

Transactional

- void **DSPI_MasterTransferCreateHandle** (SPI_Type *base, dsp_i_master_handle_t *handle, dsp_i_master_transfer_callback_t callback, void *userData)
Initializes the DSPI master handle.
- status_t **DSPI_MasterTransferBlocking** (SPI_Type *base, dsp_i_transfer_t *transfer)
DSPI master transfer data using polling.
- status_t **DSPI_MasterTransferNonBlocking** (SPI_Type *base, dsp_i_master_handle_t *handle, dsp_i_transfer_t *transfer)
DSPI master transfer data using interrupts.
- status_t **DSPI_MasterTransferGetCount** (SPI_Type *base, dsp_i_master_handle_t *handle, size_t *count)
Gets the master transfer count.
- void **DSPI_MasterTransferAbort** (SPI_Type *base, dsp_i_master_handle_t *handle)
DSPI master aborts a transfer using an interrupt.
- void **DSPI_MasterTransferHandleIRQ** (SPI_Type *base, dsp_i_master_handle_t *handle)
DSPI Master IRQ handler function.
- void **DSPI_SlaveTransferCreateHandle** (SPI_Type *base, dsp_i_slave_handle_t *handle, dsp_i_slave_transfer_callback_t callback, void *userData)
Initializes the DSPI slave handle.
- status_t **DSPI_SlaveTransferNonBlocking** (SPI_Type *base, dsp_i_slave_handle_t *handle, dsp_i_transfer_t *transfer)
DSPI slave transfers data using an interrupt.
- status_t **DSPI_SlaveTransferGetCount** (SPI_Type *base, dsp_i_slave_handle_t *handle, size_t *count)
Gets the slave transfer count.
- void **DSPI_SlaveTransferAbort** (SPI_Type *base, dsp_i_slave_handle_t *handle)
DSPI slave aborts a transfer using an interrupt.
- void **DSPI_SlaveTransferHandleIRQ** (SPI_Type *base, dsp_i_slave_handle_t *handle)
DSPI Master IRQ handler function.

11.2.3 Data Structure Documentation

11.2.3.1 struct dsp_i_command_data_config_t

Data Fields

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

DSPI Driver

- Signals that the current transfer is the last in the queue.
- bool [clearTransferCount](#)
Clears the SPI Transfer Counter (SPI_TCNT) before transmission starts.

11.2.3.1.0.15 Field Documentation

11.2.3.1.0.15.1 bool [dspi_command_data_config_t::isPcsContinuous](#)

11.2.3.1.0.15.2 [dspi_ctar_selection_t](#) [dspi_command_data_config_t::whichCtar](#)

11.2.3.1.0.15.3 [dspi_which_pcs_t](#) [dspi_command_data_config_t::whichPcs](#)

11.2.3.1.0.15.4 bool [dspi_command_data_config_t::isEndOfQueue](#)

11.2.3.1.0.15.5 bool [dspi_command_data_config_t::clearTransferCount](#)

11.2.3.2 struct [dspi_master_ctar_config_t](#)

Data Fields

- uint32_t [baudRate](#)
Baud Rate for DSPI.
- uint32_t [bitsPerFrame](#)
Bits per frame, minimum 4, maximum 16.
- [dspi_clock_polarity_t](#) [cpol](#)
Clock polarity.
- [dspi_clock_phase_t](#) [cpha](#)
Clock phase.
- [dspi_shift_direction_t](#) [direction](#)
MSB or LSB data shift direction.
- uint32_t [pcsToSckDelayInNanoSec](#)
PCS to SCK delay time in nanoseconds; setting to 0 sets the minimum delay.
- uint32_t [lastSckToPcsDelayInNanoSec](#)
The last SCK to PCS delay time in nanoseconds; setting to 0 sets the minimum delay.
- uint32_t [betweenTransferDelayInNanoSec](#)
After the SCK delay time in nanoseconds; setting to 0 sets the minimum delay.

11.2.3.2.0.16 Field Documentation

11.2.3.2.0.16.1 `uint32_t dspi_master_ctar_config_t::baudRate`

11.2.3.2.0.16.2 `uint32_t dspi_master_ctar_config_t::bitsPerFrame`

11.2.3.2.0.16.3 `dspi_clock_polarity_t dspi_master_ctar_config_t::cpol`

11.2.3.2.0.16.4 `dspi_clock_phase_t dspi_master_ctar_config_t::cpha`

11.2.3.2.0.16.5 `dspi_shift_direction_t dspi_master_ctar_config_t::direction`

11.2.3.2.0.16.6 `uint32_t dspi_master_ctar_config_t::pcsToSckDelayInNanoSec`

It also sets the boundary value if out of range.

11.2.3.2.0.16.7 `uint32_t dspi_master_ctar_config_t::lastSckToPcsDelayInNanoSec`

It also sets the boundary value if out of range.

11.2.3.2.0.16.8 `uint32_t dspi_master_ctar_config_t::betweenTransferDelayInNanoSec`

It also sets the boundary value if out of range.

11.2.3.3 struct dspi_master_config_t

Data Fields

- `dspi_ctar_selection_t whichCtar`
The desired CTAR to use.
- `dspi_master_ctar_config_t ctarConfig`
Set the ctarConfig to the desired CTAR.
- `dspi_which_pcs_t whichPcs`
The desired Peripheral Chip Select (pcs).
- `dspi_pcs_polarity_config_t pcsActiveHighOrLow`
The desired PCS active high or low.
- `bool enableContinuousSCK`
CONT_SCKE, continuous SCK enable.
- `bool enableRxFifoOverWrite`
ROOE, receive FIFO overflow overwrite enable.
- `bool enableModifiedTimingFormat`
Enables a modified transfer format to be used if true.
- `dspi_master_sample_point_t samplePoint`
Controls when the module master samples SIN in the Modified Transfer Format.

DSPI Driver

11.2.3.3.0.17 Field Documentation

11.2.3.3.0.17.1 `dspi_ctar_selection_t dspi_master_config_t::whichCtar`

11.2.3.3.0.17.2 `dspi_master_ctar_config_t dspi_master_config_t::ctarConfig`

11.2.3.3.0.17.3 `dspi_which_pcs_t dspi_master_config_t::whichPcs`

11.2.3.3.0.17.4 `dspi_pcs_polarity_config_t dspi_master_config_t::pcsActiveHighOrLow`

11.2.3.3.0.17.5 `bool dspi_master_config_t::enableContinuousSCK`

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

11.2.3.3.0.17.6 `bool dspi_master_config_t::enableRxFifoOverWrite`

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

11.2.3.3.0.17.7 `bool dspi_master_config_t::enableModifiedTimingFormat`

11.2.3.3.0.17.8 `dspi_master_sample_point_t dspi_master_config_t::samplePoint`

It's valid only when CPHA=0.

11.2.3.4 struct `dspi_slave_ctar_config_t`

Data Fields

- `uint32_t bitsPerFrame`
Bits per frame, minimum 4, maximum 16.
- `dspi_clock_polarity_t cpol`
Clock polarity.
- `dspi_clock_phase_t cpha`
Clock phase.

11.2.3.4.0.18 Field Documentation

11.2.3.4.0.18.1 `uint32_t dspi_slave_ctar_config_t::bitsPerFrame`

11.2.3.4.0.18.2 `dspi_clock_polarity_t dspi_slave_ctar_config_t::cpol`

11.2.3.4.0.18.3 `dspi_clock_phase_t dspi_slave_ctar_config_t::cpha`

Slave only supports MSB and does not support LSB.

11.2.3.5 struct dsp_slave_config_t

Data Fields

- [dsp_ctar_selection_t](#) `whichCtar`
The desired CTAR to use.
- [dsp_slave_ctar_config_t](#) `ctarConfig`
Set the ctarConfig to the desired CTAR.
- `bool` [enableContinuousSCK](#)
CONT_SCKE, continuous SCK enable.
- `bool` [enableRxFifoOverWrite](#)
ROOE, receive FIFO overflow overwrite enable.
- `bool` [enableModifiedTimingFormat](#)
Enables a modified transfer format to be used if true.
- [dsp_master_sample_point_t](#) `samplePoint`
Controls when the module master samples SIN in the Modified Transfer Format.

11.2.3.5.0.19 Field Documentation

11.2.3.5.0.19.1 `dsp_ctar_selection_t dsp_slave_config_t::whichCtar`

11.2.3.5.0.19.2 `dsp_slave_ctar_config_t dsp_slave_config_t::ctarConfig`

11.2.3.5.0.19.3 `bool dsp_slave_config_t::enableContinuousSCK`

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

11.2.3.5.0.19.4 `bool dsp_slave_config_t::enableRxFifoOverWrite`

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

11.2.3.5.0.19.5 `bool dsp_slave_config_t::enableModifiedTimingFormat`

11.2.3.5.0.19.6 `dsp_master_sample_point_t dsp_slave_config_t::samplePoint`

It's valid only when CPHA=0.

11.2.3.6 struct dsp_transfer_t

Data Fields

- `uint8_t *` [txData](#)
Send buffer.
- `uint8_t *` [rxData](#)
Receive buffer.
- `volatile size_t` [dataSize](#)
Transfer bytes.
- `uint32_t` [configFlags](#)
Transfer transfer configuration flags; set from `_dsp_transfer_config_flag_for_master` if the transfer is

DSPI Driver

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

11.2.3.6.0.20 Field Documentation

11.2.3.6.0.20.1 `uint8_t* dspi_transfer_t::txData`

11.2.3.6.0.20.2 `uint8_t* dspi_transfer_t::rxData`

11.2.3.6.0.20.3 `volatile size_t dspi_transfer_t::dataSize`

11.2.3.6.0.20.4 `uint32_t dspi_transfer_t::configFlags`

11.2.3.7 struct `_dspi_master_handle`

Forward declaration of the `_dspi_master_handle` typedefs.

Data Fields

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

11.2.3.7.0.21 Field Documentation

- 11.2.3.7.0.21.1 `uint32_t dspi_master_handle_t::bitsPerFrame`
 - 11.2.3.7.0.21.2 `volatile uint32_t dspi_master_handle_t::command`
 - 11.2.3.7.0.21.3 `volatile uint32_t dspi_master_handle_t::lastCommand`
 - 11.2.3.7.0.21.4 `uint8_t dspi_master_handle_t::fifoSize`
 - 11.2.3.7.0.21.5 `volatile bool dspi_master_handle_t::isPcsActiveAfterTransfer`
 - 11.2.3.7.0.21.6 `volatile bool dspi_master_handle_t::isThereExtraByte`
 - 11.2.3.7.0.21.7 `uint8_t* volatile dspi_master_handle_t::txData`
 - 11.2.3.7.0.21.8 `uint8_t* volatile dspi_master_handle_t::rxData`
 - 11.2.3.7.0.21.9 `volatile size_t dspi_master_handle_t::remainingSendByteCount`
 - 11.2.3.7.0.21.10 `volatile size_t dspi_master_handle_t::remainingReceiveByteCount`
 - 11.2.3.7.0.21.11 `volatile uint8_t dspi_master_handle_t::state`
 - 11.2.3.7.0.21.12 `dspi_master_transfer_callback_t dspi_master_handle_t::callback`
 - 11.2.3.7.0.21.13 `void* dspi_master_handle_t::userData`
- ### 11.2.3.8 struct _dspi_slave_handle

Forward declaration of the [_dspi_slave_handle](#) typedefs.

Data Fields

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

DSPI Driver

- volatile uint32_t `errorCount`
Error count for slave transfer.
- `dspi_slave_transfer_callback_t` `callback`
Completion callback.
- void * `userData`
Callback user data.

11.2.3.8.0.22 Field Documentation

- 11.2.3.8.0.22.1 uint32_t `dspi_slave_handle_t::bitsPerFrame`
- 11.2.3.8.0.22.2 volatile bool `dspi_slave_handle_t::isThereExtraByte`
- 11.2.3.8.0.22.3 uint8_t* volatile `dspi_slave_handle_t::txData`
- 11.2.3.8.0.22.4 uint8_t* volatile `dspi_slave_handle_t::rxData`
- 11.2.3.8.0.22.5 volatile size_t `dspi_slave_handle_t::remainingSendByteCount`
- 11.2.3.8.0.22.6 volatile size_t `dspi_slave_handle_t::remainingReceiveByteCount`
- 11.2.3.8.0.22.7 volatile uint8_t `dspi_slave_handle_t::state`
- 11.2.3.8.0.22.8 volatile uint32_t `dspi_slave_handle_t::errorCount`
- 11.2.3.8.0.22.9 `dspi_slave_transfer_callback_t` `dspi_slave_handle_t::callback`
- 11.2.3.8.0.22.10 void* `dspi_slave_handle_t::userData`

11.2.4 Macro Definition Documentation

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

11.2.4.2 `#define DSPI_DUMMY_DATA (0x00U)`

Dummy data used for Tx if there is no txData.

11.2.4.3 #define DSPI_MASTER_CTAR_SHIFT (0U)

11.2.4.4 #define DSPI_MASTER_CTAR_MASK (0x0FU)

11.2.4.5 #define DSPI_MASTER_PCS_SHIFT (4U)

11.2.4.6 #define DSPI_MASTER_PCS_MASK (0xF0U)

11.2.4.7 #define DSPI_SLAVE_CTAR_SHIFT (0U)

11.2.4.8 #define DSPI_SLAVE_CTAR_MASK (0x07U)

11.2.5 Typedef Documentation

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

DSPI Driver

Parameters

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

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

Parameters

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

11.2.6 Enumeration Type Documentation

11.2.6.1 enum _dspi_status

Enumerator

kStatus_DSPI_Busy DSPI transfer is busy.
kStatus_DSPI_Error DSPI driver error.
kStatus_DSPI_Idle DSPI is idle.
kStatus_DSPI_OutOfRange DSPI transfer out of range.

11.2.6.2 enum _dspi_flags

Enumerator

kDSPI_TxCompleteFlag Transfer Complete Flag.
kDSPI_EndOfQueueFlag End of Queue Flag.
kDSPI_TxFifoUnderflowFlag Transmit FIFO Underflow Flag.
kDSPI_TxFifoFillRequestFlag Transmit FIFO Fill Flag.
kDSPI_RxFifoOverflowFlag Receive FIFO Overflow Flag.
kDSPI_RxFifoDrainRequestFlag Receive FIFO Drain Flag.
kDSPI_TxAndRxStatusFlag The module is in Stopped/Running state.
kDSPI_AllStatusFlag All statuses above.

11.2.6.3 enum _dspi_interrupt_enable

Enumerator

kDSPI_TxCompleteInterruptEnable TCF interrupt enable.
kDSPI_EndOfQueueInterruptEnable EOQF interrupt enable.
kDSPI_TxFifoUnderflowInterruptEnable TFUF interrupt enable.
kDSPI_TxFifoFillRequestInterruptEnable TFFF interrupt enable, DMA disable.
kDSPI_RxFifoOverflowInterruptEnable RFOF interrupt enable.
kDSPI_RxFifoDrainRequestInterruptEnable RFDF interrupt enable, DMA disable.
kDSPI_AllInterruptEnable All above interrupts enable.

11.2.6.4 enum _dspi_dma_enable

Enumerator

kDSPI_TxDmaEnable TFFF flag generates DMA requests. No Tx interrupt request.
kDSPI_RxDmaEnable RFDF flag generates DMA requests. No Rx interrupt request.

11.2.6.5 enum dspi_master_slave_mode_t

Enumerator

kDSPI_Master DSPI peripheral operates in master mode.
kDSPI_Slave DSPI peripheral operates in slave mode.

11.2.6.6 enum dspi_master_sample_point_t

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

Enumerator

kDSPI_SckToSin0Clock 0 system clocks between SCK edge and SIN sample.
kDSPI_SckToSin1Clock 1 system clock between SCK edge and SIN sample.
kDSPI_SckToSin2Clock 2 system clocks between SCK edge and SIN sample.

11.2.6.7 enum dspi_which_pcs_t

Enumerator

kDSPI_Pcs0 Pcs[0].
kDSPI_Pcs1 Pcs[1].
kDSPI_Pcs2 Pcs[2].

DSPI Driver

kDSPI_Pcs3 Pcs[3].

kDSPI_Pcs4 Pcs[4].

kDSPI_Pcs5 Pcs[5].

11.2.6.8 enum dspi_pcs_polarity_config_t

Enumerator

kDSPI_PcsActiveHigh Pcs Active High (idles low).

kDSPI_PcsActiveLow Pcs Active Low (idles high).

11.2.6.9 enum _dspi_pcs_polarity

Enumerator

kDSPI_Pcs0ActiveLow Pcs0 Active Low (idles high).

kDSPI_Pcs1ActiveLow Pcs1 Active Low (idles high).

kDSPI_Pcs2ActiveLow Pcs2 Active Low (idles high).

kDSPI_Pcs3ActiveLow Pcs3 Active Low (idles high).

kDSPI_Pcs4ActiveLow Pcs4 Active Low (idles high).

kDSPI_Pcs5ActiveLow Pcs5 Active Low (idles high).

kDSPI_PcsAllActiveLow Pcs0 to Pcs5 Active Low (idles high).

11.2.6.10 enum dspi_clock_polarity_t

Enumerator

kDSPI_ClockPolarityActiveHigh CPOL=0. Active-high DSPI clock (idles low).

kDSPI_ClockPolarityActiveLow CPOL=1. Active-low DSPI clock (idles high).

11.2.6.11 enum dspi_clock_phase_t

Enumerator

kDSPI_ClockPhaseFirstEdge CPHA=0. Data is captured on the leading edge of the SCK and changed on the following edge.

kDSPI_ClockPhaseSecondEdge CPHA=1. Data is changed on the leading edge of the SCK and captured on the following edge.

11.2.6.12 enum dspi_shift_direction_t

Enumerator

kDSPI_MsbFirst Data transfers start with most significant bit.

kDSPI_LsbFirst Data transfers start with least significant bit. Shifting out of LSB is not supported for slave

11.2.6.13 enum dspi_delay_type_t

Enumerator

kDSPI_PcsToSck Pcs-to-SCK delay.

kDSPI_LastSckToPcs The last SCK edge to Pcs delay.

kDSPI_BetweenTransfer Delay between transfers.

11.2.6.14 enum dspi_ctar_selection_t

Enumerator

kDSPI_Ctar0 CTAR0 selection option for master or slave mode; note that CTAR0 and CTAR0_SLAVE are the same register address.

kDSPI_Ctar1 CTAR1 selection option for master mode only.

kDSPI_Ctar2 CTAR2 selection option for master mode only; note that some devices do not support CTAR2.

kDSPI_Ctar3 CTAR3 selection option for master mode only; note that some devices do not support CTAR3.

kDSPI_Ctar4 CTAR4 selection option for master mode only; note that some devices do not support CTAR4.

kDSPI_Ctar5 CTAR5 selection option for master mode only; note that some devices do not support CTAR5.

kDSPI_Ctar6 CTAR6 selection option for master mode only; note that some devices do not support CTAR6.

kDSPI_Ctar7 CTAR7 selection option for master mode only; note that some devices do not support CTAR7.

11.2.6.15 enum _dspi_transfer_config_flag_for_master

Enumerator

kDSPI_MasterCtar0 DSPI master transfer use CTAR0 setting.

kDSPI_MasterCtar1 DSPI master transfer use CTAR1 setting.

kDSPI_MasterCtar2 DSPI master transfer use CTAR2 setting.

DSPI Driver

kDSPI_MasterCtar3 DSPI master transfer use CTAR3 setting.
kDSPI_MasterCtar4 DSPI master transfer use CTAR4 setting.
kDSPI_MasterCtar5 DSPI master transfer use CTAR5 setting.
kDSPI_MasterCtar6 DSPI master transfer use CTAR6 setting.
kDSPI_MasterCtar7 DSPI master transfer use CTAR7 setting.
kDSPI_MasterPcs0 DSPI master transfer use PCS0 signal.
kDSPI_MasterPcs1 DSPI master transfer use PCS1 signal.
kDSPI_MasterPcs2 DSPI master transfer use PCS2 signal.
kDSPI_MasterPcs3 DSPI master transfer use PCS3 signal.
kDSPI_MasterPcs4 DSPI master transfer use PCS4 signal.
kDSPI_MasterPcs5 DSPI master transfer use PCS5 signal.
kDSPI_MasterPcsContinuous Indicates whether the PCS signal is continuous.
kDSPI_MasterActiveAfterTransfer Indicates whether the PCS signal is active after the last frame transfer.

11.2.6.16 enum _dspi_transfer_config_flag_for_slave

Enumerator

kDSPI_SlaveCtar0 DSPI slave transfer use CTAR0 setting. DSPI slave can only use PCS0.

11.2.6.17 enum _dspi_transfer_state

Enumerator

kDSPI_Idle Nothing in the transmitter/receiver.
kDSPI_Busy Transfer queue is not finished.
kDSPI_Error Transfer error.

11.2.7 Function Documentation

11.2.7.1 void DSPI_MasterInit (SPI_Type * base, const dspi_master_config_t * masterConfig, uint32_t srcClock_Hz)

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

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

```

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

```

Parameters

<i>base</i>	DSPI peripheral address.
<i>masterConfig</i>	Pointer to the structure dspi_master_config_t .
<i>srcClock_Hz</i>	Module source input clock in Hertz.

11.2.7.2 void DSPI_MasterGetDefaultConfig (dspi_master_config_t * *masterConfig*)

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

```

*   dspi_master_config_t masterConfig;
*   DSPI_MasterGetDefaultConfig(&masterConfig);
*

```

Parameters

<i>masterConfig</i>	pointer to dspi_master_config_t structure
---------------------	---

11.2.7.3 void DSPI_SlaveInit (SPI_Type * *base*, const dspi_slave_config_t * *slaveConfig*)

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

```

*   dspi_slave_config_t slaveConfig;
*   slaveConfig->whichCtar                = kDSPI_Ctar0;
*   slaveConfig->ctarConfig.bitsPerFrame  = 8;
*   slaveConfig->ctarConfig.cpol          =
    kDSPI_ClockPolarityActiveHigh;
*   slaveConfig->ctarConfig.cpha          =
    kDSPI_ClockPhaseFirstEdge;
*   slaveConfig->enableContinuousSCK      = false;

```

DSPI Driver

```
* slaveConfig->enableRxFifoOverWrite    = false;
* slaveConfig->enableModifiedTimingFormat = false;
* slaveConfig->samplePoint              = kDSPI_SckToSin0Clock;
*   DSPI_SlaveInit(base, &slaveConfig);
*
```

Parameters

<i>base</i>	DSPI peripheral address.
<i>slaveConfig</i>	Pointer to the structure dspi_master_config_t .

11.2.7.4 void DSPI_SlaveGetDefaultConfig (dspi_slave_config_t * *slaveConfig*)

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

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

Parameters

<i>slaveConfig</i>	Pointer to the dspi_slave_config_t structure.
--------------------	---

11.2.7.5 void DSPI_Deinit (SPI_Type * *base*)

Call this API to disable the DSPI clock.

Parameters

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

11.2.7.6 static void DSPI_Enable (SPI_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

11.2.7.7 static uint32_t DSPI_GetStatusFlags (SPI_Type * *base*) [inline], [static]

Parameters

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

Returns

DSPI status (in SR register).

11.2.7.8 static void DSPI_ClearStatusFlags (SPI_Type * *base*, uint32_t *statusFlags*) [inline], [static]

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

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

Parameters

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

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

11.2.7.9 void DSPI_EnableInterrupts (SPI_Type * *base*, uint32_t *mask*)

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

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

DSPI Driver

Parameters

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

11.2.7.10 `static void DSPI_DisableInterrupts (SPI_Type * base, uint32_t mask) [inline], [static]`

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

Parameters

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

11.2.7.11 `static void DSPI_EnableDMA (SPI_Type * base, uint32_t mask) [inline], [static]`

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

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

Parameters

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

11.2.7.12 `static void DSPI_DisableDMA (SPI_Type * base, uint32_t mask) [inline], [static]`

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

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

Parameters

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

11.2.7.13 **static uint32_t DSPI_MasterGetTxRegisterAddress (SPI_Type * *base*) [inline], [static]**

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

Parameters

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

Returns

The DSPI master PUSHHR data register address.

11.2.7.14 **static uint32_t DSPI_SlaveGetTxRegisterAddress (SPI_Type * *base*) [inline], [static]**

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

Parameters

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

Returns

The DSPI slave PUSHHR data register address.

11.2.7.15 **static uint32_t DSPI_GetRxRegisterAddress (SPI_Type * *base*) [inline], [static]**

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

DSPI Driver

Parameters

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

Returns

The DSPI POPR data register address.

**11.2.7.16 static void DSPI_SetMasterSlaveMode (SPI_Type * *base*,
dspi_master_slave_mode_t *mode*) [inline], [static]**

Parameters

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

11.2.7.17 static bool DSPI_IsMaster (SPI_Type * *base*) [inline], [static]

Parameters

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

Returns

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

11.2.7.18 static void DSPI_StartTransfer (SPI_Type * *base*) [inline], [static]

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

Parameters

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

11.2.7.19 static void DSPI_StopTransfer (SPI_Type * *base*) [inline], [static]

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

Parameters

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

11.2.7.20 static void DSPI_SetFifoEnable (SPI_Type * *base*, bool *enableTxFifo*, bool *enableRxFifo*) [inline], [static]

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

Parameters

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

11.2.7.21 static void DSPI_FlushFifo (SPI_Type * *base*, bool *flushTxFifo*, bool *flushRxFifo*) [inline], [static]

Parameters

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

11.2.7.22 static void DSPI_SetAllPcsPolarity (SPI_Type * *base*, uint32_t *mask*) [inline], [static]

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

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

Parameters

DSPI Driver

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

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

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

Parameters

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

Returns

The actual calculated baud rate

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

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

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

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

Parameters

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

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

11.2.7.25 `uint32_t DSPI_MasterSetDelayTimes (SPI_Type * base, dsapi_ctar_selection_t whichCtar, dsapi_delay_type_t whichDelay, uint32_t srcClock_Hz, uint32_t delayTimeInNanoSec)`

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

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

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

Parameters

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

Returns

The actual calculated delay value.

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

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

DSPI Driver

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

```
* dspi_command_data_config_t commandConfig;  
* commandConfig.isPcsContinuous = true;  
* commandConfig.whichCtar = kDSPICTar0;  
* commandConfig.whichPcs = kDSPIPcs0;  
* commandConfig.clearTransferCount = false;  
* commandConfig.isEndOfQueue = false;  
* DSPI_MasterWriteData(base, &commandConfig, dataWord);
```

Parameters

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

11.2.7.27 void DSPI_GetDefaultDataCommandConfig (dspi_command_data_config_t * *command*)

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

```
* dspi_command_data_config_t command;  
* DSPI_GetDefaultDataCommandConfig(&command);  
*
```

Parameters

<i>command</i>	Pointer to the dspi_command_data_config_t structure.
----------------	--

11.2.7.28 void DSPI_MasterWriteDataBlocking (SPI_Type * *base*, dspi_command_data_config_t * *command*, uint16_t *data*)

In master mode, the 16-bit data is appended to the 16-bit command info. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the desired Clock and Transfer Attributes register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). This is an example.

```
* dspi_command_config_t commandConfig;  
* commandConfig.isPcsContinuous = true;  
* commandConfig.whichCtar = kDSPICTar0;
```

```

* commandConfig.whichPcs = kDSPIPCs1;
* commandConfig.clearTransferCount = false;
* commandConfig.isEndOfQueue = false;
* DSPI_MasterWriteDataBlocking(base, &commandConfig, dataWord);
*

```

Note that this function does not return until after the transmit is complete. Also note that the DSPI must be enabled and running to transmit data (MCR[MDIS] & [HALT] = 0). Because the SPI is a synchronous protocol, the received data is available when the transmit completes.

Parameters

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

11.2.7.29 static uint32_t DSPI_MasterGetFormattedCommand (dspi_command_data_config_t * *command*) [inline], [static]

This function allows the caller to pass in the data command structure and returns the command word formatted according to the DSPI PUSHHR register bit field placement. The user can then "OR" the returned command word with the desired data to send and use the function DSPI_HAL_WriteCommandDataMastermode or DSPI_HAL_WriteCommandDataMastermodeBlocking to write the entire 32-bit command data word to the PUSHHR. This helps improve performance in cases where the command structure is constant. For example, the user calls this function before starting a transfer to generate the command word. When they are ready to transmit the data, they OR this formatted command word with the desired data to transmit. This process increases transmit performance when compared to calling send functions, such as DSPI_HAL_WriteDataMastermode, which format the command word each time a data word is to be sent.

Parameters

<i>command</i>	Pointer to the command structure.
----------------	-----------------------------------

Returns

The command word formatted to the PUSHHR data register bit field.

11.2.7.30 void DSPI_MasterWriteCommandDataBlocking (SPI_Type * *base*, uint32_t *data*)

In this function, the user must append the 16-bit data to the 16-bit command information and then provide the total 32-bit word as the data to send. The command portion provides characteristics of the data, such as the optional continuous chip select operation between transfers, the desired Clock and Transfer Attributes

DSPI Driver

register to use for the associated SPI frame, the desired PCS signal to use for the data transfer, whether the current transfer is the last in the queue, and whether to clear the transfer count (normally needed when sending the first frame of a data packet). The user is responsible for appending this command with the data to send. This is an example:

```
* dataWord = <16-bit command> | <16-bit data>;
* DSPI_MasterWriteCommandDataBlocking(base, dataWord);
*
```

Note that this function does not return until after the transmit is complete. Also note that the DSPI must be enabled and running to transmit data (MCR[MDIS] & [HALT] = 0). Because the SPI is a synchronous protocol, the received data is available when the transmit completes.

For a blocking polling transfer, see methods below. Option 1: uint32_t command_to_send = DSPI_MasterGetFormattedCommand(&command); uint32_t data0 = command_to_send | data_need_to_send_0; uint32_t data1 = command_to_send | data_need_to_send_1; uint32_t data2 = command_to_send | data_need_to_send_2;

DSPI_MasterWriteCommandDataBlocking(base,data0); DSPI_MasterWriteCommandDataBlocking(base,data1); DSPI_MasterWriteCommandDataBlocking(base,data2);

Option 2: DSPI_MasterWriteDataBlocking(base,&command,data_need_to_send_0); DSPI_MasterWriteDataBlocking(base,&command,data_need_to_send_1); DSPI_MasterWriteDataBlocking(base,&command,data_need_to_send_2);

Parameters

<i>base</i>	DSPI peripheral address.
<i>data</i>	The data word (command and data combined) to be sent.

11.2.7.31 static void DSPI_SlaveWriteData (SPI_Type * *base*, uint32_t *data*) [inline], [static]

In slave mode, up to 16-bit words may be written.

Parameters

<i>base</i>	DSPI peripheral address.
<i>data</i>	The data to send.

11.2.7.32 void DSPI_SlaveWriteDataBlocking (SPI_Type * *base*, uint32_t *data*)

In slave mode, up to 16-bit words may be written. The function first clears the transmit complete flag, writes data into data register, and finally waits until the data is transmitted.

Parameters

<i>base</i>	DSPI peripheral address.
<i>data</i>	The data to send.

11.2.7.33 static uint32_t DSPI_ReadData (SPI_Type * *base*) [inline], [static]

Parameters

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

Returns

The data from the read data buffer.

11.2.7.34 void DSPI_MasterTransferCreateHandle (SPI_Type * *base*, dspi_master_handle_t * *handle*, dspi_master_transfer_callback_t *callback*, void * *userData*)

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

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	DSPI handle pointer to dspi_master_handle_t.
<i>callback</i>	DSPI callback.
<i>userData</i>	Callback function parameter.

11.2.7.35 status_t DSPI_MasterTransferBlocking (SPI_Type * *base*, dspi_transfer_t * *transfer*)

This function transfers data using polling. This is a blocking function, which does not return until all transfers have been completed.

DSPI Driver

Parameters

<i>base</i>	DSPI peripheral base address.
<i>transfer</i>	Pointer to the dspi_transfer_t structure.

Returns

status of `status_t`.

11.2.7.36 `status_t DSPI_MasterTransferNonBlocking (SPI_Type * base, dspi_master_handle_t * handle, dspi_transfer_t * transfer)`

This function transfers data using interrupts. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	Pointer to the dspi_transfer_t structure.

Returns

status of `status_t`.

11.2.7.37 `status_t DSPI_MasterTransferGetCount (SPI_Type * base, dspi_master_handle_t * handle, size_t * count)`

This function gets the master transfer count.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state.
<i>count</i>	The number of bytes transferred by using the non-blocking transaction.

Returns

status of `status_t`.

11.2.7.38 void DSPI_MasterTransferAbort (SPI_Type * *base*, dspi_master_handle_t * *handle*)

This function aborts a transfer using an interrupt.

DSPI Driver

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state.

11.2.7.39 void DSPI_MasterTransferHandleIRQ (SPI_Type * *base*, dspi_master_handle_t * *handle*)

This function processes the DSPI transmit and receive IRQ.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state.

11.2.7.40 void DSPI_SlaveTransferCreateHandle (SPI_Type * *base*, dspi_slave_handle_t * *handle*, dspi_slave_transfer_callback_t *callback*, void * *userData*)

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

Parameters

<i>handle</i>	DSPI handle pointer to the <code>dspi_slave_handle_t</code> .
<i>base</i>	DSPI peripheral base address.
<i>callback</i>	DSPI callback.
<i>userData</i>	Callback function parameter.

11.2.7.41 status_t DSPI_SlaveTransferNonBlocking (SPI_Type * *base*, dspi_slave_handle_t * *handle*, dspi_transfer_t * *transfer*)

This function transfers data using an interrupt. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_slave_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	Pointer to the dspi_transfer_t structure.

Returns

status of `status_t`.

11.2.7.42 **status_t DSPI_SlaveTransferGetCount (SPI_Type * *base*, dspi_slave_handle_t * *handle*, size_t * *count*)**

This function gets the slave transfer count.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_master_handle_t</code> structure which stores the transfer state.
<i>count</i>	The number of bytes transferred by using the non-blocking transaction.

Returns

status of `status_t`.

11.2.7.43 **void DSPI_SlaveTransferAbort (SPI_Type * *base*, dspi_slave_handle_t * *handle*)**

This function aborts a transfer using an interrupt.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_slave_handle_t</code> structure which stores the transfer state.

11.2.7.44 **void DSPI_SlaveTransferHandleIRQ (SPI_Type * *base*, dspi_slave_handle_t * *handle*)**

This function processes the DSPI transmit and receive IRQ.

DSPI Driver

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	Pointer to the <code>dspi_slave_handle_t</code> structure which stores the transfer state.

11.3 DSPI DMA Driver

11.3.1 Overview

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

Data Structures

- struct [dspi_master_dma_handle_t](#)
DSPI master DMA transfer handle structure used for transactional API. [More...](#)
- struct [dspi_slave_dma_handle_t](#)
DSPI slave DMA transfer handle structure used for transactional API. [More...](#)

Typedefs

- typedef void(* [dspi_master_dma_transfer_callback_t](#))(SPI_Type *base, dspi_master_dma_handle_t *handle, status_t status, void *userData)
Completion callback function pointer type.
- typedef void(* [dspi_slave_dma_transfer_callback_t](#))(SPI_Type *base, dspi_slave_dma_handle_t *handle, status_t status, void *userData)
Completion callback function pointer type.

Functions

- void [DSPI_MasterTransferCreateHandleDMA](#) (SPI_Type *base, dspi_master_dma_handle_t *handle, [dspi_master_dma_transfer_callback_t](#) callback, void *userData, dma_handle_t *dmaRxRegToRxDataHandle, dma_handle_t *dmaTxDataToIntermediaryHandle, dma_handle_t *dmaIntermediaryToTxRegHandle)
Initializes the DSPI master DMA handle.
- status_t [DSPI_MasterTransferDMA](#) (SPI_Type *base, dspi_master_dma_handle_t *handle, [dspi_transfer_t](#) *transfer)
DSPI master transfers data using DMA.
- void [DSPI_MasterTransferAbortDMA](#) (SPI_Type *base, dspi_master_dma_handle_t *handle)
DSPI master aborts a transfer which is using DMA.
- status_t [DSPI_MasterTransferGetCountDMA](#) (SPI_Type *base, dspi_master_dma_handle_t *handle, size_t *count)
Gets the master DMA transfer remaining bytes.
- void [DSPI_SlaveTransferCreateHandleDMA](#) (SPI_Type *base, dspi_slave_dma_handle_t *handle, [dspi_slave_dma_transfer_callback_t](#) callback, void *userData, dma_handle_t *dmaRxRegToRxDataHandle, dma_handle_t *dmaTxDataToTxRegHandle)
Initializes the DSPI slave DMA handle.
- status_t [DSPI_SlaveTransferDMA](#) (SPI_Type *base, dspi_slave_dma_handle_t *handle, [dspi_transfer_t](#) *transfer)
DSPI slave transfers data using DMA.

DSPI DMA Driver

- void [DSPI_SlaveTransferAbortDMA](#) (SPI_Type *base, dsp_slave_dma_handle_t *handle)
DSPI slave aborts a transfer which is using DMA.
- status_t [DSPI_SlaveTransferGetCountDMA](#) (SPI_Type *base, dsp_slave_dma_handle_t *handle, size_t *count)
Gets the slave DMA transfer remaining bytes.

11.3.2 Data Structure Documentation

11.3.2.1 struct _dsp_master_dma_handle

Forward declaration of the DSPI DMA master handle typedefs.

Data Fields

- uint32_t [bitsPerFrame](#)
The desired number of bits per frame.
- volatile uint32_t [command](#)
The desired data command.
- volatile uint32_t [lastCommand](#)
The desired last data command.
- uint8_t [fifoSize](#)
FIFO dataSize.
- volatile bool [isPcsActiveAfterTransfer](#)
Indicates whether the PCS signal keeps active after the last frame transfer.
- volatile bool [isThereExtraByte](#)
Indicates whether there is an extra byte.
- uint8_t *volatile [txData](#)
Send buffer.
- uint8_t *volatile [rxData](#)
Receive buffer.
- volatile size_t [remainingSendByteCount](#)
A number of bytes remaining to send.
- volatile size_t [remainingReceiveByteCount](#)
A number of bytes remaining to receive.
- size_t [totalByteCount](#)
A number of transfer bytes.
- uint32_t [rxBuffIfNull](#)
Used if there is not rxData for DMA purpose.
- uint32_t [txBuffIfNull](#)
Used if there is not txData for DMA purpose.
- volatile uint8_t [state](#)
DSPI transfer state, see [_dsp_transfer_state](#).
- [dsp_master_dma_transfer_callback_t](#) [callback](#)
Completion callback.
- void * [userData](#)
Callback user data.
- dma_handle_t * [dmaRxRegToRxDataHandle](#)
dma_handle_t handle point used for RxReg to RxData buff

- `dma_handle_t * dmaTxDataToIntermediaryHandle`
dma_handle_t handle point used for TxData to Intermediary
- `dma_handle_t * dmaIntermediaryToTxRegHandle`
dma_handle_t handle point used for Intermediary to TxReg

11.3.2.1.0.23 Field Documentation

- 11.3.2.1.0.23.1 `uint32_t dspi_master_dma_handle_t::bitsPerFrame`
- 11.3.2.1.0.23.2 `volatile uint32_t dspi_master_dma_handle_t::command`
- 11.3.2.1.0.23.3 `volatile uint32_t dspi_master_dma_handle_t::lastCommand`
- 11.3.2.1.0.23.4 `uint8_t dspi_master_dma_handle_t::fifoSize`
- 11.3.2.1.0.23.5 `volatile bool dspi_master_dma_handle_t::isPcsActiveAfterTransfer`
- 11.3.2.1.0.23.6 `volatile bool dspi_master_dma_handle_t::isThereExtraByte`
- 11.3.2.1.0.23.7 `uint8_t* volatile dspi_master_dma_handle_t::txData`
- 11.3.2.1.0.23.8 `uint8_t* volatile dspi_master_dma_handle_t::rxData`
- 11.3.2.1.0.23.9 `volatile size_t dspi_master_dma_handle_t::remainingSendByteCount`
- 11.3.2.1.0.23.10 `volatile size_t dspi_master_dma_handle_t::remainingReceiveByteCount`
- 11.3.2.1.0.23.11 `uint32_t dspi_master_dma_handle_t::rxBuffIfNull`
- 11.3.2.1.0.23.12 `uint32_t dspi_master_dma_handle_t::txBuffIfNull`
- 11.3.2.1.0.23.13 `volatile uint8_t dspi_master_dma_handle_t::state`
- 11.3.2.1.0.23.14 `dspi_master_dma_transfer_callback_t dspi_master_dma_handle_t::callback`
- 11.3.2.1.0.23.15 `void* dspi_master_dma_handle_t::userData`

11.3.2.2 struct _dspi_slave_dma_handle

Forward declaration of the DSPI DMA slave handle typedefs.

Data Fields

- `uint32_t bitsPerFrame`
Desired number of bits per frame.
- `volatile bool isThereExtraByte`
Indicates whether there is an extra byte.
- `uint8_t *volatile txData`
A send buffer.
- `uint8_t *volatile rxData`

DSPI DMA Driver

- *A receive buffer.*
volatile size_t [remainingSendByteCount](#)
- *A number of bytes remaining to send.*
volatile size_t [remainingReceiveByteCount](#)
- *A number of bytes remaining to receive.*
size_t [totalByteCount](#)
- *A number of transfer bytes.*
uint32_t [rxBuffIfNull](#)
- *Used if there is not rxData for DMA purpose.*
uint32_t [txBuffIfNull](#)
- *Used if there is not txData for DMA purpose.*
uint32_t [txLastData](#)
- *Used if there is an extra byte when 16 bits per frame for DMA purpose.*
volatile uint8_t [state](#)
- *DSPI transfer state.*
uint32_t [errorCount](#)
- *Error count for the slave transfer.*
[dsp_slave_dma_transfer_callback_t](#) [callback](#)
- *Completion callback.*
void * [userData](#)
- *Callback user data.*
dma_handle_t * [dmaRxRegToRxDataHandle](#)
dma_handle_t handle point used for RxReg to RxData buff
- *dma_handle_t handle point used for TxData to TxReg*
dma_handle_t * [dmaTxDataToTxRegHandle](#)

11.3.2.2.0.24 Field Documentation

- 11.3.2.2.0.24.1** `uint32_t dspi_slave_dma_handle_t::bitsPerFrame`
- 11.3.2.2.0.24.2** `volatile bool dspi_slave_dma_handle_t::isThereExtraByte`
- 11.3.2.2.0.24.3** `uint8_t* volatile dspi_slave_dma_handle_t::txData`
- 11.3.2.2.0.24.4** `uint8_t* volatile dspi_slave_dma_handle_t::rxData`
- 11.3.2.2.0.24.5** `volatile size_t dspi_slave_dma_handle_t::remainingSendByteCount`
- 11.3.2.2.0.24.6** `volatile size_t dspi_slave_dma_handle_t::remainingReceiveByteCount`
- 11.3.2.2.0.24.7** `uint32_t dspi_slave_dma_handle_t::rxBuffIfNull`
- 11.3.2.2.0.24.8** `uint32_t dspi_slave_dma_handle_t::txBuffIfNull`
- 11.3.2.2.0.24.9** `uint32_t dspi_slave_dma_handle_t::txLastData`
- 11.3.2.2.0.24.10** `volatile uint8_t dspi_slave_dma_handle_t::state`
- 11.3.2.2.0.24.11** `uint32_t dspi_slave_dma_handle_t::errorCount`
- 11.3.2.2.0.24.12** `dspi_slave_dma_transfer_callback_t dspi_slave_dma_handle_t::callback`
- 11.3.2.2.0.24.13** `void* dspi_slave_dma_handle_t::userData`

11.3.3 Typedef Documentation

- 11.3.3.1** `typedef void(* dspi_master_dma_transfer_callback_t)(SPI_Type *base, dspi_master_dma_handle_t *handle, status_t status, void *userData)`

DSPI DMA Driver

Parameters

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

11.3.3.2 `typedef void(* dspi_slave_dma_transfer_callback_t)(SPI_Type *base, dspi_slave_dma_handle_t *handle, status_t status, void *userData)`

Parameters

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

11.3.4 Function Documentation

11.3.4.1 `void DSPI_MasterTransferCreateHandleDMA (SPI_Type * base, dspi_master_dma_handle_t * handle, dspi_master_dma_transfer_callback_t callback, void * userData, dma_handle_t * dmaRxRegToRxDataHandle, dma_handle_t * dmaTxDataToIntermediaryHandle, dma_handle_t * dmaIntermediaryToTxRegHandle)`

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

Note that DSPI DMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx is the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for dmaRxRegToRxDataHandle and Tx DMAMUX source for dmaIntermediaryToTxRegHandle. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for dmaRxRegToRxDataHandle.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	DSPI handle pointer to dspi_master_dma_handle_t.
<i>callback</i>	DSPI callback.
<i>userData</i>	A callback function parameter.
<i>dmaRxRegTo-RxDataHandle</i>	dmaRxRegToRxDataHandle pointer to dma_handle_t.
<i>dmaTxDataTo-Intermediary-Handle</i>	dmaTxDataToIntermediaryHandle pointer to dma_handle_t.
<i>dma-Intermediary-ToTxReg-Handle</i>	dmaIntermediaryToTxRegHandle pointer to dma_handle_t.

11.3.4.2 **status_t DSPI_MasterTransferDMA (SPI_Type * *base*, dspi_master_dma_handle_t * *handle*, dspi_transfer_t * *transfer*)**

This function transfers data using DMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note that the master DMA transfer does not support the transfer_size of 1 when the bitsPerFrame is greater than 8.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the dspi_master_dma_handle_t structure which stores the transfer state.
<i>transfer</i>	A pointer to the dspi_transfer_t structure.

Returns

status of status_t.

11.3.4.3 **void DSPI_MasterTransferAbortDMA (SPI_Type * *base*, dspi_master_dma_handle_t * *handle*)**

This function aborts a transfer which is using DMA.

DSPI DMA Driver

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_master_dma_handle_t</code> structure which stores the transfer state.

11.3.4.4 **status_t DSPI_MasterTransferGetCountDMA (SPI_Type * *base*, dspi_master_dma_handle_t * *handle*, size_t * *count*)**

This function gets the master DMA transfer remaining bytes.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_master_dma_handle_t</code> structure which stores the transfer state.
<i>count</i>	A number of bytes transferred by the non-blocking transaction.

Returns

status of `status_t`.

11.3.4.5 **void DSPI_SlaveTransferCreateHandleDMA (SPI_Type * *base*, dspi_slave_dma_handle_t * *handle*, dspi_slave_dma_transfer_callback_t *callback*, void * *userData*, dma_handle_t * *dmaRxRegToRxDataHandle*, dma_handle_t * *dmaTxDataToTxRegHandle*)**

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

Note that DSPI DMA has a separated (Rx and Tx as two sources) or shared (Rx and Tx is the same source) DMA request source. (1) For a separated DMA request source, enable and set the Rx DMAMUX source for `dmaRxRegToRxDataHandle` and Tx DMAMUX source for `dmaTxDataToTxRegHandle`. (2) For a shared DMA request source, enable and set the Rx/Rx DMAMUX source for `dmaRxRegToRxDataHandle`.

Parameters

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

<i>handle</i>	DSPI handle pointer to <code>dspi_slave_dma_handle_t</code> .
<i>callback</i>	DSPI callback.
<i>userData</i>	A callback function parameter.
<i>dmaRxRegTo-RxDataHandle</i>	<code>dmaRxRegToRxDataHandle</code> pointer to <code>dma_handle_t</code> .
<i>dmaTxDataTo-TxRegHandle</i>	<code>dmaTxDataToTxRegHandle</code> pointer to <code>dma_handle_t</code> .

11.3.4.6 **status_t DSPI_SlaveTransferDMA (SPI_Type * *base*, dspi_slave_dma_handle_t * *handle*, dspi_transfer_t * *transfer*)**

This function transfers data using DMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Note that the slave DMA transfer does not support the `transfer_size` of 1 when the `bitsPerFrame` is greater than eight.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_slave_dma_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	A pointer to the dspi_transfer_t structure.

Returns

status of `status_t`.

11.3.4.7 **void DSPI_SlaveTransferAbortDMA (SPI_Type * *base*, dspi_slave_dma_handle_t * *handle*)**

This function aborts a transfer which is using DMA.

Parameters

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

DSPI DMA Driver

<i>handle</i>	A pointer to the <code>dspi_slave_dma_handle_t</code> structure which stores the transfer state.
---------------	--

11.3.4.8 `status_t DSPI_SlaveTransferGetCountDMA (SPI_Type * base, dspi_slave_dma_handle_t * handle, size_t * count)`

This function gets the slave DMA transfer remaining bytes.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_slave_dma_handle_t</code> structure which stores the transfer state.
<i>count</i>	A number of bytes transferred by the non-blocking transaction.

Returns

status of `status_t`.

11.4 DSPI eDMA Driver

11.4.1 Overview

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

Data Structures

- struct [dspi_master_edma_handle_t](#)
DSPI master eDMA transfer handle structure used for the transactional API. [More...](#)
- struct [dspi_slave_edma_handle_t](#)
DSPI slave eDMA transfer handle structure used for the transactional API. [More...](#)

Typedefs

- typedef void(* [dspi_master_edma_transfer_callback_t](#))(SPI_Type *base, dspi_master_edma_handle_t *handle, status_t status, void *userData)
Completion callback function pointer type.
- typedef void(* [dspi_slave_edma_transfer_callback_t](#))(SPI_Type *base, dspi_slave_edma_handle_t *handle, status_t status, void *userData)
Completion callback function pointer type.

Functions

- void [DSPI_MasterTransferCreateHandleEDMA](#) (SPI_Type *base, dspi_master_edma_handle_t *handle, [dspi_master_edma_transfer_callback_t](#) callback, void *userData, [edma_handle_t](#) *edmaRxRegToRxDataHandle, [edma_handle_t](#) *edmaTxDataToIntermediaryHandle, [edma_handle_t](#) *edmaIntermediaryToTxRegHandle)
Initializes the DSPI master eDMA handle.
- status_t [DSPI_MasterTransferEDMA](#) (SPI_Type *base, dspi_master_edma_handle_t *handle, [dspi_transfer_t](#) *transfer)
DSPI master transfer data using eDMA.
- void [DSPI_MasterTransferAbortEDMA](#) (SPI_Type *base, dspi_master_edma_handle_t *handle)
DSPI master aborts a transfer which is using eDMA.
- status_t [DSPI_MasterTransferGetCountEDMA](#) (SPI_Type *base, dspi_master_edma_handle_t *handle, size_t *count)
Gets the master eDMA transfer count.
- void [DSPI_SlaveTransferCreateHandleEDMA](#) (SPI_Type *base, dspi_slave_edma_handle_t *handle, [dspi_slave_edma_transfer_callback_t](#) callback, void *userData, [edma_handle_t](#) *edmaRxRegToRxDataHandle, [edma_handle_t](#) *edmaTxDataToTxRegHandle)
Initializes the DSPI slave eDMA handle.
- status_t [DSPI_SlaveTransferEDMA](#) (SPI_Type *base, dspi_slave_edma_handle_t *handle, [dspi_transfer_t](#) *transfer)
DSPI slave transfer data using eDMA.

DSPI eDMA Driver

- void [DSPI_SlaveTransferAbortEDMA](#) (SPI_Type *base, dsp_slave_edma_handle_t *handle)
DSPI slave aborts a transfer which is using eDMA.
- status_t [DSPI_SlaveTransferGetCountEDMA](#) (SPI_Type *base, dsp_slave_edma_handle_t *handle, size_t *count)
Gets the slave eDMA transfer count.

11.4.2 Data Structure Documentation

11.4.2.1 struct _dsp_master_edma_handle

Forward declaration of the DSPI eDMA master handle typedefs.

Data Fields

- uint32_t [bitsPerFrame](#)
The desired number of bits per frame.
- volatile uint32_t [command](#)
The desired data command.
- volatile uint32_t [lastCommand](#)
The desired last data command.
- uint8_t [fifoSize](#)
FIFO dataSize.
- volatile bool [isPcsActiveAfterTransfer](#)
Indicates whether the PCS signal keeps active after the last frame transfer.
- uint8_t [nbytes](#)
eDMA minor byte transfer count initially configured.
- volatile uint8_t [state](#)
DSPI transfer state , _dsp_transfer_state.
- uint8_t *volatile [txData](#)
Send buffer.
- uint8_t *volatile [rxData](#)
Receive buffer.
- volatile size_t [remainingSendByteCount](#)
A number of bytes remaining to send.
- volatile size_t [remainingReceiveByteCount](#)
A number of bytes remaining to receive.
- size_t [totalByteCount](#)
A number of transfer bytes.
- uint32_t [rxBuffIfNull](#)
Used if there is not rxData for DMA purpose.
- uint32_t [txBuffIfNull](#)
Used if there is not txData for DMA purpose.
- [dsp_master_edma_transfer_callback_t](#) [callback](#)
Completion callback.
- void * [userData](#)
Callback user data.
- [edma_handle_t](#) * [edmaRxRegToRxDataHandle](#)
edma_handle_t handle point used for RxReg to RxData buff

- `edma_handle_t * edmaTxDataToIntermediaryHandle`
edma_handle_t handle point used for TxData to Intermediary
- `edma_handle_t * edmaIntermediaryToTxRegHandle`
edma_handle_t handle point used for Intermediary to TxReg
- `edma_tcd_t dspiSoftwareTCD [2]`
SoftwareTCD , internal used.

11.4.2.1.0.25 Field Documentation

- 11.4.2.1.0.25.1 `uint32_t dspi_master_edma_handle_t::bitsPerFrame`
- 11.4.2.1.0.25.2 `volatile uint32_t dspi_master_edma_handle_t::command`
- 11.4.2.1.0.25.3 `volatile uint32_t dspi_master_edma_handle_t::lastCommand`
- 11.4.2.1.0.25.4 `uint8_t dspi_master_edma_handle_t::fifoSize`
- 11.4.2.1.0.25.5 `volatile bool dspi_master_edma_handle_t::isPcsActiveAfterTransfer`
- 11.4.2.1.0.25.6 `uint8_t dspi_master_edma_handle_t::nbytes`
- 11.4.2.1.0.25.7 `volatile uint8_t dspi_master_edma_handle_t::state`
- 11.4.2.1.0.25.8 `uint8_t* volatile dspi_master_edma_handle_t::txData`
- 11.4.2.1.0.25.9 `uint8_t* volatile dspi_master_edma_handle_t::rxData`
- 11.4.2.1.0.25.10 `volatile size_t dspi_master_edma_handle_t::remainingSendByteCount`
- 11.4.2.1.0.25.11 `volatile size_t dspi_master_edma_handle_t::remainingReceiveByteCount`
- 11.4.2.1.0.25.12 `uint32_t dspi_master_edma_handle_t::rxBuffIfNull`
- 11.4.2.1.0.25.13 `uint32_t dspi_master_edma_handle_t::txBuffIfNull`
- 11.4.2.1.0.25.14 `dspi_master_edma_transfer_callback_t dspi_master_edma_handle_t::callback`
- 11.4.2.1.0.25.15 `void* dspi_master_edma_handle_t::userData`

11.4.2.2 struct _dspi_slave_edma_handle

Forward declaration of the DSPI eDMA slave handle typedefs.

Data Fields

- `uint32_t bitsPerFrame`
The desired number of bits per frame.
- `uint8_t *volatile txData`
Send buffer.
- `uint8_t *volatile rxData`

DSPI eDMA Driver

- *Receive buffer.*
volatile size_t [remainingSendByteCount](#)
A number of bytes remaining to send.
- volatile size_t [remainingReceiveByteCount](#)
A number of bytes remaining to receive.
- size_t [totalByteCount](#)
A number of transfer bytes.
- uint32_t [rxBuffIfNull](#)
Used if there is not rxData for DMA purpose.
- uint32_t [txBuffIfNull](#)
Used if there is not txData for DMA purpose.
- uint32_t [txLastData](#)
Used if there is an extra byte when 16bits per frame for DMA purpose.
- uint8_t [nbytes](#)
eDMA minor byte transfer count initially configured.
- volatile uint8_t [state](#)
DSPI transfer state.
- [dsp_slave_edma_transfer_callback_t](#) [callback](#)
Completion callback.
- void * [userData](#)
Callback user data.
- [edma_handle_t](#) * [edmaRxRegToRxDataHandle](#)
edma_handle_t handle point used for RxReg to RxData buff
- [edma_handle_t](#) * [edmaTxDataToTxRegHandle](#)
edma_handle_t handle point used for TxData to TxReg

11.4.2.2.0.26 Field Documentation

- 11.4.2.2.0.26.1** `uint32_t dspi_slave_edma_handle_t::bitsPerFrame`
- 11.4.2.2.0.26.2** `uint8_t* volatile dspi_slave_edma_handle_t::txData`
- 11.4.2.2.0.26.3** `uint8_t* volatile dspi_slave_edma_handle_t::rxData`
- 11.4.2.2.0.26.4** `volatile size_t dspi_slave_edma_handle_t::remainingSendByteCount`
- 11.4.2.2.0.26.5** `volatile size_t dspi_slave_edma_handle_t::remainingReceiveByteCount`
- 11.4.2.2.0.26.6** `uint32_t dspi_slave_edma_handle_t::rxBuffIfNull`
- 11.4.2.2.0.26.7** `uint32_t dspi_slave_edma_handle_t::txBuffIfNull`
- 11.4.2.2.0.26.8** `uint32_t dspi_slave_edma_handle_t::txLastData`
- 11.4.2.2.0.26.9** `uint8_t dspi_slave_edma_handle_t::nbytes`
- 11.4.2.2.0.26.10** `volatile uint8_t dspi_slave_edma_handle_t::state`
- 11.4.2.2.0.26.11** `dspi_slave_edma_transfer_callback_t dspi_slave_edma_handle_t::callback`
- 11.4.2.2.0.26.12** `void* dspi_slave_edma_handle_t::userData`

11.4.3 Typedef Documentation

- 11.4.3.1** `typedef void(* dspi_master_edma_transfer_callback_t)(SPI_Type *base, dspi_master_edma_handle_t *handle, status_t status, void *userData)`

DSPI eDMA Driver

Parameters

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

11.4.3.2 typedef void(* dspi_slave_edma_transfer_callback_t)(SPI_Type *base, dspi_slave_edma_handle_t *handle, status_t status, void *userData)

Parameters

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

11.4.4 Function Documentation

11.4.4.1 void DSPI_MasterTransferCreateHandleEDMA (SPI_Type * base, dspi_master_edma_handle_t * handle, dspi_master_edma_transfer_callback_t callback, void * userData, edma_handle_t * edmaRxRegToRxDataHandle, edma_handle_t * edmaTxDataToIntermediaryHandle, edma_handle_t * edmaIntermediaryToTxRegHandle)

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

Note that DSPI eDMA has separated (RX and TX as two sources) or shared (RX and TX are the same source) DMA request source. (1) For the separated DMA request source, enable and set the RX DMAMUX source for edmaRxRegToRxDataHandle and TX DMAMUX source for edmaIntermediaryToTxRegHandle. (2) For the shared DMA request source, enable and set the RX/RX DMAMUX source for the edmaRxRegToRxDataHandle.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	DSPI handle pointer to <code>dspi_master_edma_handle_t</code> .
<i>callback</i>	DSPI callback.
<i>userData</i>	A callback function parameter.
<i>edmaRxRegTo-RxDataHandle</i>	<code>edmaRxRegToRxDataHandle</code> pointer to edma_handle_t .
<i>edmaTxData-To-Intermediary-Handle</i>	<code>edmaTxDataToIntermediaryHandle</code> pointer to edma_handle_t .
<i>edma-Intermediary-ToTxReg-Handle</i>	<code>edmaIntermediaryToTxRegHandle</code> pointer to edma_handle_t .

11.4.4.2 `status_t DSPI_MasterTransferEDMA (SPI_Type * base, dspi-
_master_edma_handle_t * handle, dspi_transfer_t * transfer
)`

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_master_edma_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	A pointer to the dspi_transfer_t structure.

Returns

status of `status_t`.

11.4.4.3 `void DSPI_MasterTransferAbortEDMA (SPI_Type * base,
dsapi_master_edma_handle_t * handle)`

This function aborts a transfer which is using eDMA.

DSPI eDMA Driver

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_master_edma_handle_t</code> structure which stores the transfer state.

11.4.4.4 **status_t DSPI_MasterTransferGetCountEDMA (SPI_Type * *base*, dspi_master_edma_handle_t * *handle*, size_t * *count*)**

This function gets the master eDMA transfer count.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_master_edma_handle_t</code> structure which stores the transfer state.
<i>count</i>	A number of bytes transferred by the non-blocking transaction.

Returns

status of `status_t`.

11.4.4.5 **void DSPI_SlaveTransferCreateHandleEDMA (SPI_Type * *base*, dspi_slave_edma_handle_t * *handle*, dspi_slave_edma_transfer_callback_t *callback*, void * *userData*, edma_handle_t * *edmaRxRegToRxDataHandle*, edma_handle_t * *edmaTxDataToTxRegHandle*)**

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

Note that DSPI eDMA has separated (RN and TX in 2 sources) or shared (RX and TX are the same source) DMA request source. (1)For the separated DMA request source, enable and set the RX DMAMUX source for `edmaRxRegToRxDataHandle` and TX DMAMUX source for `edmaTxDataToTxRegHandle`. (2)For the shared DMA request source, enable and set the RX/RX DMAMUX source for the `edmaRxRegToRxDataHandle`.

Parameters

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

<i>handle</i>	DSPI handle pointer to <code>dspi_slave_edma_handle_t</code> .
<i>callback</i>	DSPI callback.
<i>userData</i>	A callback function parameter.
<i>edmaRxRegTo-RxDataHandle</i>	<code>edmaRxRegToRxDataHandle</code> pointer to edma_handle_t .
<i>edmaTxData-ToTxReg-Handle</i>	<code>edmaTxDataToTxRegHandle</code> pointer to edma_handle_t .

11.4.4.6 **status_t DSPI_SlaveTransferEDMA (SPI_Type * *base*, dspi_slave_edma_handle_t * *handle*, dspi_transfer_t * *transfer*)**

This function transfers data using eDMA. This is a non-blocking function, which returns right away. When all data is transferred, the callback function is called. Note that the slave eDMA transfer doesn't support `transfer_size` is 1 when the `bitsPerFrame` is greater than eight.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_slave_edma_handle_t</code> structure which stores the transfer state.
<i>transfer</i>	A pointer to the dspi_transfer_t structure.

Returns

status of `status_t`.

11.4.4.7 **void DSPI_SlaveTransferAbortEDMA (SPI_Type * *base*, dspi_slave_edma_handle_t * *handle*)**

This function aborts a transfer which is using eDMA.

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_slave_edma_handle_t</code> structure which stores the transfer state.

11.4.4.8 **status_t DSPI_SlaveTransferGetCountEDMA (SPI_Type * *base*, dspi_slave_edma_handle_t * *handle*, size_t * *count*)**

This function gets the slave eDMA transfer count.

DSPI eDMA Driver

Parameters

<i>base</i>	DSPI peripheral base address.
<i>handle</i>	A pointer to the <code>dspi_slave_edma_handle_t</code> structure which stores the transfer state.
<i>count</i>	A number of bytes transferred so far by the non-blocking transaction.

Returns

status of `status_t`.

11.5 DSPI FreeRTOS Driver

11.5.1 Overview

DSPI RTOS Operation

- status_t **DSPI_RTOS_Init** (dspi_rtos_handle_t *handle, SPI_Type *base, const dspi_master_config_t *masterConfig, uint32_t srcClock_Hz)
Initializes the DSPI.
- status_t **DSPI_RTOS_Deinit** (dspi_rtos_handle_t *handle)
Deinitializes the DSPI.
- status_t **DSPI_RTOS_Transfer** (dspi_rtos_handle_t *handle, dspi_transfer_t *transfer)
Performs the SPI transfer.

11.5.2 Function Documentation

11.5.2.1 status_t DSPI_RTOS_Init (dspi_rtos_handle_t * handle, SPI_Type * base, const dspi_master_config_t * masterConfig, uint32_t srcClock_Hz)

This function initializes the DSPI module and the related RTOS context.

Parameters

<i>handle</i>	The RTOS DSPI handle, the pointer to an allocated space for RTOS context.
<i>base</i>	The pointer base address of the DSPI instance to initialize.
<i>masterConfig</i>	A configuration structure to set-up the DSPI in master mode.
<i>srcClock_Hz</i>	A frequency of the input clock of the DSPI module.

Returns

status of the operation.

11.5.2.2 status_t DSPI_RTOS_Deinit (dspi_rtos_handle_t * handle)

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

Parameters

DSPI FreeRTOS Driver

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

11.5.2.3 **status_t DSPI_RTOS_Transfer (dspi_rtos_handle_t * *handle*, dspi_transfer_t * *transfer*)**

This function performs the SPI transfer according to the data given in the transfer structure.

Parameters

<i>handle</i>	The RTOS DSPI handle.
<i>transfer</i>	A structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 12

eDMA: Enhanced Direct Memory Access (eDMA) Controller Driver

12.1 Overview

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

12.2 Typical use case

12.2.1 eDMA Operation

```
edma_transfer_config_t transferConfig;
edma_config_t userConfig;
uint32_t transferDone = false;

EDMA_GetDefaultConfig(&userConfig);
EDMA_Init(DMA0, &userConfig);
EDMA_CreateHandle(&g_EDMA_Handle, DMA0, channel);
EDMA_SetCallback(&g_EDMA_Handle, EDMA_Callback, &transferDone);
EDMA_PrepareTransfer(&transferConfig, srcAddr, srcWidth, destAddr, destWidth,
                    bytesEachRequest, transferBytes, kEDMA_MemoryToMemory);
EDMA_SubmitTransfer(&g_EDMA_Handle, &transferConfig, true);
EDMA_StartTransfer(&g_EDMA_Handle);
/* Waits for the eDMA transfer to finish */
while (transferDone != true);
```

Data Structures

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

Macros

- #define [DMA_DCHPRI_INDEX](#)(channel) (((channel) & ~0x03U) | (3 - ((channel)&0x03U)))
Compute the offset unit from DCHPRI3.
- #define [DMA_DCHPRIn](#)(base, channel) ((volatile uint8_t *)&(base->DCHPRI3))[[DMA_DCHPRI_INDEX](#)(channel)]
Get the pointer of DCHPRIn.

Typical use case

Typedefs

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

Enumerations

- enum [edma_transfer_size_t](#) {
 [kEDMA_TransferSize1Bytes](#) = 0x0U,
 [kEDMA_TransferSize2Bytes](#) = 0x1U,
 [kEDMA_TransferSize4Bytes](#) = 0x2U,
 [kEDMA_TransferSize16Bytes](#) = 0x4U,
 [kEDMA_TransferSize32Bytes](#) = 0x5U }
eDMA transfer configuration
- enum [edma_modulo_t](#) {

```

kEDMA_ModuloDisable = 0x0U,
kEDMA_Modulo2bytes,
kEDMA_Modulo4bytes,
kEDMA_Modulo8bytes,
kEDMA_Modulo16bytes,
kEDMA_Modulo32bytes,
kEDMA_Modulo64bytes,
kEDMA_Modulo128bytes,
kEDMA_Modulo256bytes,
kEDMA_Modulo512bytes,
kEDMA_Modulo1Kbytes,
kEDMA_Modulo2Kbytes,
kEDMA_Modulo4Kbytes,
kEDMA_Modulo8Kbytes,
kEDMA_Modulo16Kbytes,
kEDMA_Modulo32Kbytes,
kEDMA_Modulo64Kbytes,
kEDMA_Modulo128Kbytes,
kEDMA_Modulo256Kbytes,
kEDMA_Modulo512Kbytes,
kEDMA_Modulo1Mbytes,
kEDMA_Modulo2Mbytes,
kEDMA_Modulo4Mbytes,
kEDMA_Modulo8Mbytes,
kEDMA_Modulo16Mbytes,
kEDMA_Modulo32Mbytes,
kEDMA_Modulo64Mbytes,
kEDMA_Modulo128Mbytes,
kEDMA_Modulo256Mbytes,
kEDMA_Modulo512Mbytes,
kEDMA_Modulo1Gbytes,
kEDMA_Modulo2Gbytes }
    eDMA modulo configuration
• enum edma_bandwidth_t {
    kEDMA_BandwidthStallNone = 0x0U,
    kEDMA_BandwidthStall4Cycle = 0x2U,
    kEDMA_BandwidthStall8Cycle = 0x3U }
    Bandwidth control.
• enum edma_channel_link_type_t {
    kEDMA_LinkNone = 0x0U,
    kEDMA_MinorLink,
    kEDMA_MajorLink }
    Channel link type.
• enum _edma_channel_status_flags {

```

Typical use case

```
kEDMA_DoneFlag = 0x1U,  
kEDMA_ErrorFlag = 0x2U,  
kEDMA_InterruptFlag = 0x4U }  
    eDMA channel status flags.  
• enum _edma_error_status_flags {  
    kEDMA_DestinationBusErrorFlag = DMA_ES_DBE_MASK,  
    kEDMA_SourceBusErrorFlag = DMA_ES_SBE_MASK,  
    kEDMA_ScatterGatherErrorFlag = DMA_ES_SGE_MASK,  
    kEDMA_NbytesErrorFlag = DMA_ES_NCE_MASK,  
    kEDMA_DestinationOffsetErrorFlag = DMA_ES_DOE_MASK,  
    kEDMA_DestinationAddressErrorFlag = DMA_ES_DAE_MASK,  
    kEDMA_SourceOffsetErrorFlag = DMA_ES_SOE_MASK,  
    kEDMA_SourceAddressErrorFlag = DMA_ES_SAE_MASK,  
    kEDMA_ErrorChannelFlag = DMA_ES_ERRCHN_MASK,  
    kEDMA_ChannelPriorityErrorFlag = DMA_ES_CPE_MASK,  
    kEDMA_TransferCanceledFlag = DMA_ES_ECX_MASK,  
    kEDMA_ValidFlag = DMA_ES_VLD_MASK }  
    eDMA channel error status flags.  
• enum edma_interrupt_enable_t {  
    kEDMA_ErrorInterruptEnable = 0x1U,  
    kEDMA_MajorInterruptEnable = DMA_CSR_INTMAJOR_MASK,  
    kEDMA_HalfInterruptEnable = DMA_CSR_INTHALF_MASK }  
    eDMA interrupt source  
• enum edma_transfer_type_t {  
    kEDMA_MemoryToMemory = 0x0U,  
    kEDMA_PeripheralToMemory,  
    kEDMA_MemoryToPeripheral }  
    eDMA transfer type  
• enum _edma_transfer_status {  
    kStatus_EDMA_QueueFull = MAKE_STATUS(kStatusGroup_EDMA, 0),  
    kStatus_EDMA_Busy = MAKE_STATUS(kStatusGroup_EDMA, 1) }  
    eDMA transfer status
```

Driver version

- #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))
 eDMA driver version

eDMA initialization and de-initialization

- void EDMA_Init (DMA_Type *base, const edma_config_t *config)
 Initializes the eDMA peripheral.
- void EDMA_Deinit (DMA_Type *base)
 Deinitializes the eDMA peripheral.
- void EDMA_GetDefaultConfig (edma_config_t *config)
 Gets the eDMA default configuration structure.

eDMA Channel Operation

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

eDMA TCD Operation

- void [EDMA_TcdReset](#) ([edma_tcd_t](#) *tcd)
Sets all fields to default values for the TCD structure.
- void [EDMA_TcdSetTransferConfig](#) ([edma_tcd_t](#) *tcd, const [edma_transfer_config_t](#) *config, [edma_tcd_t](#) *nextTcd)
Configures the eDMA TCD transfer attribute.
- void [EDMA_TcdSetMinorOffsetConfig](#) ([edma_tcd_t](#) *tcd, const [edma_minor_offset_config_t](#) *config)
Configures the eDMA TCD minor offset feature.
- void [EDMA_TcdSetChannelLink](#) ([edma_tcd_t](#) *tcd, [edma_channel_link_type_t](#) type, uint32_t linkedChannel)
Sets the channel link for the eDMA TCD.
- static void [EDMA_TcdSetBandWidth](#) ([edma_tcd_t](#) *tcd, [edma_bandwidth_t](#) bandWidth)
Sets the bandwidth for the eDMA TCD.
- void [EDMA_TcdSetModulo](#) ([edma_tcd_t](#) *tcd, [edma_modulo_t](#) srcModulo, [edma_modulo_t](#) destModulo)
Sets the source modulo and the destination modulo for the eDMA TCD.
- static void [EDMA_TcdEnableAutoStopRequest](#) ([edma_tcd_t](#) *tcd, bool enable)
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.

Typical use case

- void [EDMA_TcdDisableInterrupts](#) (edma_tcd_t *tcd, uint32_t mask)
Disables the interrupt source for the eDMA TCD.

eDMA Channel Transfer Operation

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

eDMA Channel Status Operation

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

eDMA Transactional Operation

- void [EDMA_CreateHandle](#) (edma_handle_t *handle, DMA_Type *base, uint32_t channel)
Creates the eDMA handle.
- void [EDMA_InstallTCDMemory](#) (edma_handle_t *handle, edma_tcd_t *tcdPool, uint32_t tcdSize)
Installs the TCDs memory pool into the eDMA handle.
- void [EDMA_SetCallback](#) (edma_handle_t *handle, edma_callback callback, void *userData)
Installs a callback function for the eDMA transfer.
- void [EDMA_PrepareTransfer](#) (edma_transfer_config_t *config, void *srcAddr, uint32_t srcWidth, void *destAddr, uint32_t destWidth, uint32_t bytesEachRequest, uint32_t transferBytes, edma_transfer_type_t type)
Prepares the eDMA transfer structure.
- status_t [EDMA_SubmitTransfer](#) (edma_handle_t *handle, const edma_transfer_config_t *config)
Submits the eDMA transfer request.
- void [EDMA_StartTransfer](#) (edma_handle_t *handle)
eDMA starts transfer.
- void [EDMA_StopTransfer](#) (edma_handle_t *handle)
eDMA stops transfer.
- void [EDMA_AbortTransfer](#) (edma_handle_t *handle)
eDMA aborts transfer.
- void [EDMA_HandleIRQ](#) (edma_handle_t *handle)
eDMA IRQ handler for the current major loop transfer completion.

12.3 Data Structure Documentation

12.3.1 struct edma_config_t

Data Fields

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

12.3.1.0.0.27 Field Documentation

12.3.1.0.0.27.1 bool edma_config_t::enableContinuousLinkMode

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

12.3.1.0.0.27.2 bool edma_config_t::enableHaltOnError

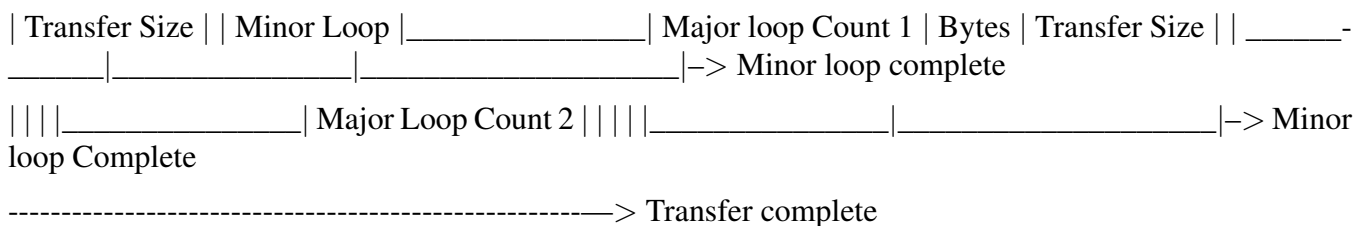
Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

12.3.1.0.0.27.3 bool edma_config_t::enableDebugMode

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

12.3.2 struct edma_transfer_config_t

This structure configures the source/destination transfer attribute. This figure shows the eDMA's transfer model:



Data Fields

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

12.3.2.0.0.28 Field Documentation

12.3.2.0.0.28.1 `uint32_t edma_transfer_config_t::srcAddr`

12.3.2.0.0.28.2 `uint32_t edma_transfer_config_t::destAddr`

12.3.2.0.0.28.3 `edma_transfer_size_t edma_transfer_config_t::srcTransferSize`

12.3.2.0.0.28.4 `edma_transfer_size_t edma_transfer_config_t::destTransferSize`

12.3.2.0.0.28.5 `int16_t edma_transfer_config_t::srcOffset`

12.3.2.0.0.28.6 `int16_t edma_transfer_config_t::destOffset`

12.3.2.0.0.28.7 `uint32_t edma_transfer_config_t::majorLoopCounts`

12.3.3 `struct edma_channel_Preemption_config_t`

Data Fields

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

12.3.4 struct edma_minor_offset_config_t

Data Fields

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

12.3.4.0.0.29 Field Documentation

12.3.4.0.0.29.1 bool edma_minor_offset_config_t::enableSrcMinorOffset

12.3.4.0.0.29.2 bool edma_minor_offset_config_t::enableDestMinorOffset

12.3.4.0.0.29.3 uint32_t edma_minor_offset_config_t::minorOffset

12.3.5 struct edma_tcd_t

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

Data Fields

- __IO uint32_t [SADDR](#)
SADDR register, used to save source address.
- __IO uint16_t [SOFF](#)
SOFF register, save offset bytes every transfer.
- __IO uint16_t [ATTR](#)
ATTR register, source/destination transfer size and modulo.
- __IO uint32_t [NBYTES](#)
Nbytes register, minor loop length in bytes.
- __IO uint32_t [SLAST](#)
SLAST register.
- __IO uint32_t [DADDR](#)
DADDR register, used for destination address.
- __IO uint16_t [DOFF](#)
DOFF register, used for destination offset.
- __IO uint16_t [CITER](#)
CITER register, current minor loop numbers, for unfinished minor loop.
- __IO uint32_t [DLAST_SGA](#)
DLASTSGA register, next stcd address used in scatter-gather mode.
- __IO uint16_t [CSR](#)
CSR register, for TCD control status.
- __IO uint16_t [BITER](#)
BITER register, begin minor loop count.

Data Structure Documentation

12.3.5.0.0.30 Field Documentation

12.3.5.0.0.30.1 `__IO uint16_t edma_tcd_t::CITER`

12.3.5.0.0.30.2 `__IO uint16_t edma_tcd_t::BITER`

12.3.6 struct edma_handle_t

Data Fields

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

12.3.6.0.0.31 Field Documentation

12.3.6.0.0.31.1 `edma_callback edma_handle_t::callback`

12.3.6.0.0.31.2 `void* edma_handle_t::userData`

12.3.6.0.0.31.3 `DMA_Type* edma_handle_t::base`

12.3.6.0.0.31.4 `edma_tcd_t* edma_handle_t::tcdPool`

12.3.6.0.0.31.5 `uint8_t edma_handle_t::channel`

12.3.6.0.0.31.6 `volatile int8_t edma_handle_t::header`

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

12.3.6.0.0.31.7 `volatile int8_t edma_handle_t::tail`

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

12.3.6.0.0.31.8 volatile int8_t edma_handle_t::tcdUsed

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

12.3.6.0.0.31.9 volatile int8_t edma_handle_t::tcdSize**12.3.6.0.0.31.10 uint8_t edma_handle_t::flags****12.4 Macro Definition Documentation****12.4.1 #define FSL_EDMA_DRIVER_VERSION (MAKE_VERSION(2, 1, 1))**

Version 2.1.1.

12.5 Typedef Documentation**12.5.1 typedef void(* edma_callback)(struct _edma_handle *handle, void *userData, bool transferDone, uint32_t tcds)****12.6 Enumeration Type Documentation****12.6.1 enum edma_transfer_size_t**

Enumerator

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

12.6.2 enum edma_modulo_t

Enumerator

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

Enumeration Type Documentation

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 enum _edma_channel_status_flags

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 enum _edma_error_status_flags

Enumerator

kEDMA_DestinationBusErrorFlag Bus error on destination address.

kEDMA_SourceBusErrorFlag Bus error on the source address.

kEDMA_ScatterGatherErrorFlag Error on the Scatter/Gather address, not 32byte aligned.

kEDMA_NbytesErrorFlag NBYTES/CITER configuration error.

kEDMA_DestinationOffsetErrorFlag Destination offset not aligned with destination size.

kEDMA_DestinationAddressErrorFlag Destination address not aligned with destination size.

kEDMA_SourceOffsetErrorFlag Source offset not aligned with source size.

kEDMA_SourceAddressErrorFlag Source address not aligned with source size.

kEDMA_ErrorChannelFlag Error channel number of the cancelled channel number.

kEDMA_ChannelPriorityErrorFlag Channel priority is not unique.

kEDMA_TransferCanceledFlag Transfer cancelled.

kEDMA_ValidFlag No error occurred, this bit is 0. Otherwise, it is 1.

12.6.7 enum edma_interrupt_enable_t

Enumerator

kEDMA_ErrorInterruptEnable Enable interrupt while channel error occurs.

kEDMA_MajorInterruptEnable Enable interrupt while major count exhausted.

kEDMA_HalfInterruptEnable Enable interrupt while major count to half value.

12.6.8 enum edma_transfer_type_t

Enumerator

kEDMA_MemoryToMemory Transfer from memory to memory.

kEDMA_PeripheralToMemory Transfer from peripheral to memory.

kEDMA_MemoryToPeripheral Transfer from memory to peripheral.

12.6.9 enum _edma_transfer_status

Enumerator

kStatus_EDMA_QueueFull TCD queue is full.

kStatus_EDMA_Busy Channel is busy and can't handle the transfer request.

12.7 Function Documentation

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

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

Parameters

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

Note

This function enables the minor loop map feature.

12.7.2 void EDMA_Deinit (DMA_Type * *base*)

This function gates the eDMA clock.

Parameters

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

12.7.3 void EDMA_GetDefaultConfig (edma_config_t * *config*)

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

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

Parameters

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

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

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

Function Documentation

Parameters

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

Note

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

This function enables the auto stop request feature.

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

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

Example:

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

Parameters

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

Note

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

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

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

Parameters

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

12.7.7 static void EDMA_SetChannelPreemptionConfig (DMA_Type * *base*, uint32_t *channel*, const edma_channel_Preemption_config_t * *config*) [inline], [static]

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

Parameters

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

12.7.8 void EDMA_SetChannelLink (DMA_Type * *base*, uint32_t *channel*, edma_channel_link_type_t *type*, uint32_t *linkedChannel*)

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

Parameters

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

Function Documentation

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

Note

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

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

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

Parameters

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

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

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

Parameters

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

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

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

Function Documentation

<i>mask</i>	The mask of the interrupt source to be set. Use the defined <code>edma_interrupt_enable_t</code> type.
-------------	--

12.7.14 void EDMA_TcdReset (edma_tcd_t * *tcd*)

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

Parameters

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

Note

This function enables the auto stop request feature.

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

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

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

Parameters

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

Note

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

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

12.7.16 `void EDMA_TcdSetMinorOffsetConfig (edma_tcd_t * 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.

Function Documentation

Parameters

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

12.7.17 void EDMA_TcdSetChannelLink (edma_tcd_t * *tcd*, edma_channel_link_type_t *type*, uint32_t *linkedChannel*)

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

Note

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

Parameters

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

12.7.18 static void EDMA_TcdSetBandWidth (edma_tcd_t * *tcd*, edma_bandwidth_t *bandWidth*) [inline], [static]

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

Parameters

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

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

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

Parameters

<i>tcd</i>	A pointer to the TCD structure.
<i>srcModulo</i>	A source modulo value.
<i>destModulo</i>	A destination modulo value.

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

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

Parameters

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

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

Parameters

Function Documentation

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

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

Parameters

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

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

This function enables the hardware channel request.

Parameters

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

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

This function disables the hardware channel request.

Parameters

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

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

This function starts a minor loop transfer.

Parameters

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

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

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

Parameters

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

Returns

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

Note

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

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

Function Documentation

Parameters

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

Returns

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

12.7.28 `uint32_t EDMA_GetChannelStatusFlags (DMA_Type * base, uint32_t channel)`

Parameters

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

Returns

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

12.7.29 `void EDMA_ClearChannelStatusFlags (DMA_Type * base, uint32_t channel, uint32_t mask)`

Parameters

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

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

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

Parameters

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

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

This function is called after the EDMA_CreateHandle to use scatter/gather feature.

Parameters

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

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

This callback is called in the eDMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

Parameters

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

12.7.33 void EDMA_PrepareTransfer (edma_transfer_config_t * *config*, void * *srcAddr*, uint32_t *srcWidth*, void * *destAddr*, uint32_t *destWidth*, uint32_t *bytesEachRequest*, uint32_t *transferBytes*, edma_transfer_type_t *type*)

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

Function Documentation

Parameters

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

Note

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

12.7.34 `status_t EDMA_SubmitTransfer (edma_handle_t * handle, const edma_transfer_config_t * config)`

This function submits the eDMA transfer request according to the transfer configuration structure. If submitting the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

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

Return values

<i>kStatus_EDMA_Success</i>	It means submit transfer request succeed.
<i>kStatus_EDMA_Queue-Full</i>	It means TCD queue is full. Submit transfer request is not allowed.

<i>kStatus_EDMA_Busy</i>	It means the given channel is busy, need to submit request later.
--------------------------	---

12.7.35 void EDMA_StartTransfer (edma_handle_t * *handle*)

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

Parameters

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

12.7.36 void EDMA_StopTransfer (edma_handle_t * *handle*)

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

Parameters

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

12.7.37 void EDMA_AbortTransfer (edma_handle_t * *handle*)

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

Parameters

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

12.7.38 void EDMA_HandleIRQ (edma_handle_t * *handle*)

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

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

Function Documentation

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

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

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

Parameters

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

Chapter 13

EWM: External Watchdog Monitor Driver

13.1 Overview

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

13.2 Typical use case

```
ewm_config_t config;
EWM_GetDefaultConfig(&config);
config.enableInterrupt = true;
config.compareLowValue = 0U;
config.compareHighValue = 0xAAU;
NVIC_EnableIRQ(WDOG_EWM_IRQn);
EWM_Init(base, &config);
```

Data Structures

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

Enumerations

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

Driver version

- #define `FSL_EWM_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 1)`)
EWM driver version 2.0.1.

EWM initialization and de-initialization

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

EWM functional Operation

- static void `EWM_EnableInterrupts` (`EWM_Type *base`, `uint32_t mask`)
Enables the EWM interrupt.
- static void `EWM_DisableInterrupts` (`EWM_Type *base`, `uint32_t mask`)

Enumeration Type Documentation

- 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 [compareLowValue](#)
Compare low-register value.
- uint8_t [compareHighValue](#)
Compare high-register value.

13.4 Macro Definition Documentation

13.4.1 #define FSL_EWM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

13.5 Enumeration Type Documentation

13.5.1 enum _ewm_interrupt_enable_t

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

Enumerator

kEWM_InterruptEnable Enable the EWM to generate an interrupt.

13.5.2 enum _ewm_status_flags_t

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

Enumerator

kEWM_RunningFlag Running flag, set when EWM is enabled.

13.6 Function Documentation

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

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

This is an example.

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

Parameters

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

13.6.2 void EWM_Deinit (EWM_Type * *base*)

This function is used to shut down the EWM.

Parameters

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

13.6.3 void EWM_GetDefaultConfig (ewm_config_t * *config*)

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

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

Function Documentation

Parameters

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

See Also

[ewm_config_t](#)

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

This function enables the EWM interrupt.

Parameters

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

C90TFS Flash Driver

14.1 Overview

The flash provides the C90TFS Flash driver of MCUXpresso SDK devices with the C90TFS Flash module inside. The flash driver provides general APIs to handle specific operations on C90TFS/FTFx Flash module. The user can use those APIs directly in the application. In addition, it provides internal functions called by the driver. Although these functions are not meant to be called from the user's application directly, the APIs can still be used.

Data Structures

- struct [flash_execute_in_ram_function_config_t](#)
Flash execute-in-RAM function information. [More...](#)
- struct [flash_swap_state_config_t](#)
Flash Swap information. [More...](#)
- struct [flash_swap_ifr_field_config_t](#)
Flash Swap IFR fields. [More...](#)
- union [flash_swap_ifr_field_data_t](#)
Flash Swap IFR field data. [More...](#)
- union [pflash_protection_status_low_t](#)
PFlash protection status - low 32bit. [More...](#)
- struct [pflash_protection_status_t](#)
PFlash protection status - full. [More...](#)
- struct [flash_prefetch_speculation_status_t](#)
Flash prefetch speculation status. [More...](#)
- struct [flash_protection_config_t](#)
Active flash protection information for the current operation. [More...](#)
- struct [flash_access_config_t](#)
Active flash Execute-Only access information for the current operation. [More...](#)
- struct [flash_operation_config_t](#)
Active flash information for the current operation. [More...](#)
- struct [flash_config_t](#)
Flash driver state information. [More...](#)

Typedefs

- typedef void(* [flash_callback_t](#))(void)
A callback type used for the Pflash block.

Enumerations

- enum [flash_margin_value_t](#) {
 [kFLASH_MarginValueNormal](#),
 [kFLASH_MarginValueUser](#),
 [kFLASH_MarginValueFactory](#),

Overview

`kFLASH_MarginValueInvalid` }

Enumeration for supported flash margin levels.

- enum `flash_security_state_t` {
 `kFLASH_SecurityStateNotSecure`,
 `kFLASH_SecurityStateBackdoorEnabled`,
 `kFLASH_SecurityStateBackdoorDisabled` }

Enumeration for the three possible flash security states.

- enum `flash_protection_state_t` {
 `kFLASH_ProtectionStateUnprotected`,
 `kFLASH_ProtectionStateProtected`,
 `kFLASH_ProtectionStateMixed` }

Enumeration for the three possible flash protection levels.

- enum `flash_execute_only_access_state_t` {
 `kFLASH_AccessStateUnLimited`,
 `kFLASH_AccessStateExecuteOnly`,
 `kFLASH_AccessStateMixed` }

Enumeration for the three possible flash execute access levels.

- enum `flash_property_tag_t` {
 `kFLASH_PropertyPflashSectorSize` = 0x00U,
 `kFLASH_PropertyPflashTotalSize` = 0x01U,
 `kFLASH_PropertyPflashBlockSize` = 0x02U,
 `kFLASH_PropertyPflashBlockCount` = 0x03U,
 `kFLASH_PropertyPflashBlockBaseAddr` = 0x04U,
 `kFLASH_PropertyPflashFacSupport` = 0x05U,
 `kFLASH_PropertyPflashAccessSegmentSize` = 0x06U,
 `kFLASH_PropertyPflashAccessSegmentCount` = 0x07U,
 `kFLASH_PropertyFlexRamBlockBaseAddr` = 0x08U,
 `kFLASH_PropertyFlexRamTotalSize` = 0x09U,
 `kFLASH_PropertyDflashSectorSize` = 0x10U,
 `kFLASH_PropertyDflashTotalSize` = 0x11U,
 `kFLASH_PropertyDflashBlockSize` = 0x12U,
 `kFLASH_PropertyDflashBlockCount` = 0x13U,
 `kFLASH_PropertyDflashBlockBaseAddr` = 0x14U,
 `kFLASH_PropertyEepromTotalSize` = 0x15U,
 `kFLASH_PropertyFlashMemoryIndex` = 0x20U,
 `kFLASH_PropertyFlashCacheControllerIndex` = 0x21U }

Enumeration for various flash properties.

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

Constants for execute-in-RAM flash function.

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

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

- enum `_flash_read_resource_range` {


```

kFLASH_ResourceRangePflashIfrSizeInBytes = 256U,
kFLASH_ResourceRangeVersionIdSizeInBytes = 8U,
kFLASH_ResourceRangeVersionIdStart = 0x00U,
kFLASH_ResourceRangeVersionIdEnd = 0x07U ,
kFLASH_ResourceRangePflashSwapIfrEnd,
kFLASH_ResourceRangeDflashIfrStart = 0x800000U,
kFLASH_ResourceRangeDflashIfrEnd = 0x8003FFU }

```

Enumeration for the range of special-purpose flash resource.

- enum `_k3_flash_read_once_index` {
`kFLASH_RecordIndexSwapAddr` = 0xA1U,
`kFLASH_RecordIndexSwapEnable` = 0xA2U,
`kFLASH_RecordIndexSwapDisable` = 0xA3U }
Enumeration for the index of read/program once record.
- enum `flash_flexram_function_option_t` {
`kFLASH_FlexramFunctionOptionAvailableAsRam` = 0xFFU,
`kFLASH_FlexramFunctionOptionAvailableForEeprom` = 0x00U }
Enumeration for the two possible options of set FlexRAM function command.

- enum `_flash_acceleration_ram_property`
Enumeration for acceleration RAM property.
- enum `flash_swap_function_option_t` {
`kFLASH_SwapFunctionOptionEnable` = 0x00U,
`kFLASH_SwapFunctionOptionDisable` = 0x01U }
Enumeration for the possible options of Swap function.
- enum `flash_swap_control_option_t` {
`kFLASH_SwapControlOptionInitializeSystem` = 0x01U,
`kFLASH_SwapControlOptionSetInUpdateState` = 0x02U,
`kFLASH_SwapControlOptionSetInCompleteState` = 0x04U,
`kFLASH_SwapControlOptionReportStatus` = 0x08U,
`kFLASH_SwapControlOptionDisableSystem` = 0x10U }
Enumeration for the possible options of Swap control commands.

- enum `flash_swap_state_t` {
`kFLASH_SwapStateUninitialized` = 0x00U,
`kFLASH_SwapStateReady` = 0x01U,
`kFLASH_SwapStateUpdate` = 0x02U,
`kFLASH_SwapStateUpdateErased` = 0x03U,
`kFLASH_SwapStateComplete` = 0x04U,
`kFLASH_SwapStateDisabled` = 0x05U }
Enumeration for the possible flash Swap status.
- enum `flash_swap_block_status_t` {
`kFLASH_SwapBlockStatusLowerHalfProgramBlocksAtZero`,
`kFLASH_SwapBlockStatusUpperHalfProgramBlocksAtZero` }
Enumeration for the possible flash Swap block status

- enum `flash_partition_flexram_load_option_t` {
`kFLASH_PartitionFlexramLoadOptionLoadedWithValidEepromData`,
`kFLASH_PartitionFlexramLoadOptionNotLoaded` = 0x01U }
Enumeration for the FlexRAM load during reset option.

- enum `flash_memory_index_t` {

Overview

```
kFLASH_MemoryIndexPrimaryFlash = 0x00U,  
kFLASH_MemoryIndexSecondaryFlash = 0x01U }
```

Enumeration for the flash memory index.

- enum `flash_cache_controller_index_t` {
 `kFLASH_CacheControllerIndexForCore0` = 0x00U,
 `kFLASH_CacheControllerIndexForCore1` = 0x01U }

Enumeration for the flash cache controller index.

- enum `flash_prefetch_speculation_option_t`
 Enumeration for the two possible options of flash prefetch speculation.
- enum `flash_cache_clear_process_t` {
 `kFLASH_CacheClearProcessPre` = 0x00U,
 `kFLASH_CacheClearProcessPost` = 0x01U }

Flash cache clear process code.

Flash version

- enum `_flash_driver_version_constants` {
 `kFLASH_DriverVersionName` = 'F',
 `kFLASH_DriverVersionMajor` = 2,
 `kFLASH_DriverVersionMinor` = 3,
 `kFLASH_DriverVersionBugfix` = 1 }
 Flash driver version for ROM.
- #define `MAKE_VERSION`(major, minor, bugfix) (((major) << 16) | ((minor) << 8) | (bugfix))
 Constructs the version number for drivers.
- #define `FSL_FLASH_DRIVER_VERSION` (`MAKE_VERSION`(2, 3, 1))
 Flash driver version for SDK.

Flash configuration

- #define `FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT` 1
 Indicates whether to support FlexNVM in the Flash driver.
- #define `FLASH_SSD_IS_FLEXNVM_ENABLED` (`FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT` && `FSL_FEATURE_FLASH_HAS_FLEX_NVM`)
 Indicates whether the FlexNVM is enabled in the Flash driver.
- #define `FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT` 1
 Indicates whether to support Secondary flash in the Flash driver.
- #define `FLASH_SSD_IS_SECONDARY_FLASH_ENABLED` (0)
 Indicates whether the secondary flash is supported in the Flash driver.
- #define `FLASH_DRIVER_IS_FLASH_RESIDENT` 1
 Flash driver location.
- #define `FLASH_DRIVER_IS_EXPORTED` 0
 Flash Driver Export option.

Flash status

- enum `_flash_status` {
`kStatus_FLASH_Success` = MAKE_STATUS(kStatusGroupGeneric, 0),
`kStatus_FLASH_InvalidArgument` = MAKE_STATUS(kStatusGroupGeneric, 4),
`kStatus_FLASH_SizeError` = MAKE_STATUS(kStatusGroupFlashDriver, 0),
`kStatus_FLASH_AlignmentError`,
`kStatus_FLASH_AddressError` = MAKE_STATUS(kStatusGroupFlashDriver, 2),
`kStatus_FLASH_AccessError`,
`kStatus_FLASH_ProtectionViolation`,
`kStatus_FLASH_CommandFailure`,
`kStatus_FLASH_UnknownProperty` = MAKE_STATUS(kStatusGroupFlashDriver, 6),
`kStatus_FLASH_EraseKeyError` = MAKE_STATUS(kStatusGroupFlashDriver, 7),
`kStatus_FLASH_RegionExecuteOnly`,
`kStatus_FLASH_ExecuteInRamFunctionNotReady`,
`kStatus_FLASH_PartitionStatusUpdateFailure`,
`kStatus_FLASH_SetFlexramAsEepromError`,
`kStatus_FLASH_RecoverFlexramAsRamError`,
`kStatus_FLASH_SetFlexramAsRamError` = MAKE_STATUS(kStatusGroupFlashDriver, 13),
`kStatus_FLASH_RecoverFlexramAsEepromError`,
`kStatus_FLASH_CommandNotSupported` = MAKE_STATUS(kStatusGroupFlashDriver, 15),
`kStatus_FLASH_SwapSystemNotInUninitialized`,
`kStatus_FLASH_SwapIndicatorAddressError`,
`kStatus_FLASH_ReadOnlyProperty` = MAKE_STATUS(kStatusGroupFlashDriver, 18),
`kStatus_FLASH_InvalidPropertyValue`,
`kStatus_FLASH_InvalidSpeculationOption` }
Flash driver status codes.
- #define `kStatusGroupGeneric` 0
Flash driver status group.
- #define `kStatusGroupFlashDriver` 1
- #define `MAKE_STATUS`(group, code) (((group)*100) + (code)))
Constructs a status code value from a group and a code number.

Flash API key

- enum `_flash_driver_api_keys` { `kFLASH_ApiEraseKey` = FOUR_CHAR_CODE('k', 'f', 'e', 'k') }
Enumeration for Flash driver API keys.
- #define `FOUR_CHAR_CODE`(a, b, c, d) (((d) << 24) | ((c) << 16) | ((b) << 8) | ((a)))
Constructs the four character code for the Flash driver API key.

Initialization

- status_t `FLASH_Init` (flash_config_t *config)
Initializes the global flash properties structure members.
- status_t `FLASH_SetCallback` (flash_config_t *config, flash_callback_t callback)
Sets the desired flash callback function.
- status_t `FLASH_PrepareExecuteInRamFunctions` (flash_config_t *config)
Prepares flash execute-in-RAM functions.

Overview

Erasing

- status_t [FLASH_EraseAll](#) (flash_config_t *config, uint32_t key)
Erases entire flash.
- status_t [FLASH_Erase](#) (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)
Erases the flash sectors encompassed by parameters passed into function.
- status_t [FLASH_EraseAllExecuteOnlySegments](#) (flash_config_t *config, uint32_t key)
Erases the entire flash, including protected sectors.

Programming

- status_t [FLASH_Program](#) (flash_config_t *config, uint32_t start, uint32_t *src, uint32_t lengthInBytes)
Programs flash with data at locations passed in through parameters.
- status_t [FLASH_ProgramOnce](#) (flash_config_t *config, uint32_t index, uint32_t *src, uint32_t lengthInBytes)
Programs Program Once Field through parameters.

Reading

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

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

Parameters

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

Return values

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

<i>kStatus_FLASH_Address-Error</i>	Address is out of range.
<i>kStatus_FLASH_Set-FlexramAsRamError</i>	Failed to set flexram as RAM.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_-CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_Recover-FlexramAsEepromError</i>	Failed to recover FlexRAM as EEPROM.

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

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

Parameters

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

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Address-Error</i>	Address is out of range.

Overview

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

- status_t **FLASH_ReadOnce** (flash_config_t *config, uint32_t index, uint32_t *dst, uint32_t lengthInBytes)
Reads the resource with data at locations passed in through parameters.

Security

- status_t **FLASH_GetSecurityState** (flash_config_t *config, flash_security_state_t *state)
Returns the security state via the pointer passed into the function.
- status_t **FLASH_SecurityBypass** (flash_config_t *config, const uint8_t *backdoorKey)
Allows users to bypass security with a backdoor key.

Verification

- status_t **FLASH_VerifyEraseAll** (flash_config_t *config, flash_margin_value_t margin)
Verifies erasure of the entire flash at a specified margin level.
- status_t **FLASH_VerifyErase** (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_margin_value_t margin)
Verifies an erasure of the desired flash area at a specified margin level.
- status_t **FLASH_VerifyProgram** (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint32_t *expectedData, flash_margin_value_t margin, uint32_t *failedAddress, uint32_t *failedData)
Verifies programming of the desired flash area at a specified margin level.
- status_t **FLASH_VerifyEraseAllExecuteOnlySegments** (flash_config_t *config, flash_margin_value_t margin)
Verifies whether the program flash execute-only segments have been erased to the specified read margin level.

Protection

- status_t **FLASH_IsProtected** (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_protection_state_t *protection_state)
Returns the protection state of the desired flash area via the pointer passed into the function.
- status_t **FLASH_IsExecuteOnly** (flash_config_t *config, uint32_t start, uint32_t lengthInBytes, flash_execute_only_access_state_t *access_state)
Returns the access state of the desired flash area via the pointer passed into the function.

Properties

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

Returns the desired flash property.

- status_t **FLASH_SetProperty** (flash_config_t *config, flash_property_tag_t whichProperty, uint32_t value)

Sets the desired flash property.

Flash Protection Utilities

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

Parameters

<i>config</i>	Pointer to storage for the driver runtime state.
<i>option</i>	The option used to set FlexRAM load behavior during reset.
<i>eeepromData-SizeCode</i>	Determines the amount of FlexRAM used in each of the available EEPROM subsystems.
<i>flexnvm-PartitionCode</i>	Specifies how to split the FlexNVM block between data flash memory and EEPROM backup memory supporting EEPROM functions.

Return values

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

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

Sets the PFlash Protection to the intended protection status.

- status_t **FLASH_PflashGetProtection** (flash_config_t *config, pflash_protection_status_t *protectStatus)

Gets the PFlash protection status.

14.2 Data Structure Documentation

14.2.1 struct flash_execute_in_ram_function_config_t

Data Fields

- uint32_t [activeFunctionCount](#)
Number of available execute-in-RAM functions.
- uint32_t * [flashRunCommand](#)
Execute-in-RAM function: flash_run_command.
- uint32_t * [flashCommonBitOperation](#)
Execute-in-RAM function: flash_common_bit_operation.

14.2.1.0.0.32 Field Documentation

14.2.1.0.0.32.1 uint32_t flash_execute_in_ram_function_config_t::activeFunctionCount

14.2.1.0.0.32.2 uint32_t* flash_execute_in_ram_function_config_t::flashRunCommand

14.2.1.0.0.32.3 uint32_t* flash_execute_in_ram_function_config_t::flashCommonBitOperation

14.2.2 struct flash_swap_state_config_t

Data Fields

- [flash_swap_state_t](#) flashSwapState
The current Swap system status.
- [flash_swap_block_status_t](#) currentSwapBlockStatus
The current Swap block status.
- [flash_swap_block_status_t](#) nextSwapBlockStatus
The next Swap block status.

14.2.2.0.0.33 Field Documentation

14.2.2.0.0.33.1 flash_swap_state_t flash_swap_state_config_t::flashSwapState

14.2.2.0.0.33.2 flash_swap_block_status_t flash_swap_state_config_t::currentSwapBlockStatus

14.2.2.0.0.33.3 flash_swap_block_status_t flash_swap_state_config_t::nextSwapBlockStatus

14.2.3 struct flash_swap_ifr_field_config_t

Data Fields

- uint16_t [swapIndicatorAddress](#)
A Swap indicator address field.
- uint16_t [swapEnableWord](#)
A Swap enable word field.
- uint8_t [reserved0](#) [4]

A reserved field.

14.2.3.0.0.34 Field Documentation

14.2.3.0.0.34.1 uint16_t flash_swap_ifr_field_config_t::swapIndicatorAddress

14.2.3.0.0.34.2 uint16_t flash_swap_ifr_field_config_t::swapEnableWord

14.2.3.0.0.34.3 uint8_t flash_swap_ifr_field_config_t::reserved0[4]

14.2.4 union flash_swap_ifr_field_data_t

Data Fields

- uint32_t [flashSwapIfrData](#) [2]
A flash Swap IFR field data .
- [flash_swap_ifr_field_config_t](#) [flashSwapIfrField](#)
A flash Swap IFR field structure.

14.2.4.0.0.35 Field Documentation

14.2.4.0.0.35.1 uint32_t flash_swap_ifr_field_data_t::flashSwapIfrData[2]

14.2.4.0.0.35.2 flash_swap_ifr_field_config_t flash_swap_ifr_field_data_t::flashSwapIfrField

14.2.5 union pflash_protection_status_low_t

Data Fields

- uint32_t [protl32b](#)
PROT[31:0] .
- uint8_t [protsl](#)
PROTS[7:0] .
- uint8_t [protsh](#)
PROTS[15:8] .

Data Structure Documentation

14.2.5.0.0.36 Field Documentation

14.2.5.0.0.36.1 `uint32_t pflash_protection_status_low_t::protl32b`

14.2.5.0.0.36.2 `uint8_t pflash_protection_status_low_t::protsl`

14.2.5.0.0.36.3 `uint8_t pflash_protection_status_low_t::protsh`

14.2.6 struct `pflash_protection_status_t`

Data Fields

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

14.2.6.0.0.37 Field Documentation

14.2.6.0.0.37.1 `pflash_protection_status_low_t pflash_protection_status_t::valueLow32b`

14.2.7 struct `flash_prefetch_speculation_status_t`

Data Fields

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

14.2.7.0.0.38 Field Documentation

14.2.7.0.0.38.1 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::instructionOption`

14.2.7.0.0.38.2 `flash_prefetch_speculation_option_t flash_prefetch_speculation_status_t::dataOption`

14.2.8 struct `flash_protection_config_t`

Data Fields

- `uint32_t regionBase`
Base address of flash protection region.
- `uint32_t regionSize`
size of flash protection region.
- `uint32_t regionCount`
flash protection region count.

14.2.8.0.0.39 Field Documentation**14.2.8.0.0.39.1** uint32_t flash_protection_config_t::regionBase**14.2.8.0.0.39.2** uint32_t flash_protection_config_t::regionSize**14.2.8.0.0.39.3** uint32_t flash_protection_config_t::regionCount**14.2.9 struct flash_access_config_t****Data Fields**

- uint32_t [SegmentBase](#)
Base address of flash Execute-Only segment.
- uint32_t [SegmentSize](#)
size of flash Execute-Only segment.
- uint32_t [SegmentCount](#)
flash Execute-Only segment count.

14.2.9.0.0.40 Field Documentation**14.2.9.0.0.40.1** uint32_t flash_access_config_t::SegmentBase**14.2.9.0.0.40.2** uint32_t flash_access_config_t::SegmentSize**14.2.9.0.0.40.3** uint32_t flash_access_config_t::SegmentCount**14.2.10 struct flash_operation_config_t****Data Fields**

- uint32_t [convertedAddress](#)
A converted address for the current flash type.
- uint32_t [activeSectorSize](#)
A sector size of the current flash type.
- uint32_t [activeBlockSize](#)
A block size of the current flash type.
- uint32_t [blockWriteUnitSize](#)
The write unit size.
- uint32_t [sectorCmdAddressAligment](#)
An erase sector command address alignment.
- uint32_t [partCmdAddressAligment](#)
A program/verify part command address alignment.
- 32_t [resourceCmdAddressAligment](#)
A read resource command address alignment.
- uint32_t [checkCmdAddressAligment](#)
A program check command address alignment.

Data Structure Documentation

14.2.10.0.0.41 Field Documentation

14.2.10.0.0.41.1 uint32_t flash_operation_config_t::convertedAddress

14.2.10.0.0.41.2 uint32_t flash_operation_config_t::activeSectorSize

14.2.10.0.0.41.3 uint32_t flash_operation_config_t::activeBlockSize

14.2.10.0.0.41.4 uint32_t flash_operation_config_t::blockWriteUnitSize

14.2.10.0.0.41.5 uint32_t flash_operation_config_t::sectorCmdAddressAligment

14.2.10.0.0.41.6 uint32_t flash_operation_config_t::partCmdAddressAligment

14.2.10.0.0.41.7 uint32_t flash_operation_config_t::resourceCmdAddressAligment

14.2.10.0.0.41.8 uint32_t flash_operation_config_t::checkCmdAddressAligment

14.2.11 struct flash_config_t

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

Data Fields

- uint32_t [PFlashBlockBase](#)
A base address of the first PFlash block.
- uint32_t [PFlashTotalSize](#)
The size of the combined PFlash block.
- uint8_t [PFlashBlockCount](#)
A number of PFlash blocks.
- uint8_t [FlashMemoryIndex](#)
0 - primary flash; 1 - secondary flash
- uint8_t [FlashCacheControllerIndex](#)
0 - Controller for core 0; 1 - Controller for core 1
- uint8_t [Reserved0](#)
Reserved field 0.
- uint32_t [PFlashSectorSize](#)
The size in bytes of a sector of PFlash.
- [flash_callback_t](#) [PFlashCallback](#)
The callback function for the flash API.
- uint32_t [PFlashAccessSegmentSize](#)
A size in bytes of an access segment of PFlash.
- uint32_t [PFlashAccessSegmentCount](#)
A number of PFlash access segments.
- uint32_t * [flashExecuteInRamFunctionInfo](#)
An information structure of the flash execute-in-RAM function.
- uint32_t [FlexRAMBlockBase](#)
For the FlexNVM device, this is the base address of the FlexRAM.

- `uint32_t FlexRAMTotalSize`
For the FlexNVM device, this is the size of the FlexRAM.
- `uint32_t DFlashBlockBase`
For the FlexNVM device, this is the base address of the D-Flash memory (FlexNVM memory)
- `uint32_t DFlashTotalSize`
For the FlexNVM device, this is the total size of the FlexNVM memory;.
- `uint32_t EEpromTotalSize`
For the FlexNVM device, this is the size in bytes of the EEPROM area which was partitioned from FlexRAM.

14.2.11.0.0.42 Field Documentation

14.2.11.0.0.42.1 `uint32_t flash_config_t::PFlashTotalSize`

14.2.11.0.0.42.2 `uint8_t flash_config_t::PFlashBlockCount`

14.2.11.0.0.42.3 `uint32_t flash_config_t::PFlashSectorSize`

14.2.11.0.0.42.4 `flash_callback_t flash_config_t::PFlashCallback`

14.2.11.0.0.42.5 `uint32_t flash_config_t::PFlashAccessSegmentSize`

14.2.11.0.0.42.6 `uint32_t flash_config_t::PFlashAccessSegmentCount`

14.2.11.0.0.42.7 `uint32_t* flash_config_t::flashExecuteInRamFunctionInfo`

14.2.11.0.0.42.8 `uint32_t flash_config_t::FlexRAMBlockBase`

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

14.2.11.0.0.42.9 `uint32_t flash_config_t::FlexRAMTotalSize`

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

14.2.11.0.0.42.10 `uint32_t flash_config_t::DFlashBlockBase`

For the non-FlexNVM device, this field is unused

14.2.11.0.0.42.11 `uint32_t flash_config_t::DFlashTotalSize`

For the non-FlexNVM device, this field is unused

14.2.11.0.0.42.12 `uint32_t flash_config_t::EEpromTotalSize`

For the non-FlexNVM device, this field is unused

Enumeration Type Documentation

14.3 Macro Definition Documentation

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

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

Version 2.3.1.

14.3.3 #define FLASH_SSD_CONFIG_ENABLE_FLEXNVM_SUPPORT 1

Enables the FlexNVM support by default.

14.3.4 #define FLASH_SSD_CONFIG_ENABLE_SECONDARY_FLASH_SUPPORT 1

Enables the secondary flash support by default.

14.3.5 #define FLASH_DRIVER_IS_FLASH_RESIDENT 1

Used for the flash resident application.

14.3.6 #define FLASH_DRIVER_IS_EXPORTED 0

Used for the KSDK application.

14.3.7 #define kStatusGroupGeneric 0

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

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

14.4 Enumeration Type Documentation

14.4.1 enum _flash_driver_version_constants

Enumerator

kFLASH_DriverVersionName Flash driver version name.

kFLASH_DriverVersionMajor Major flash driver version.
kFLASH_DriverVersionMinor Minor flash driver version.
kFLASH_DriverVersionBugfix Bugfix for flash driver version.

14.4.2 enum _flash_status

Enumerator

kStatus_FLASH_Success API is executed successfully.
kStatus_FLASH_InvalidArgument Invalid argument.
kStatus_FLASH_SizeError Error size.
kStatus_FLASH_AlignmentError Parameter is not aligned with the specified baseline.
kStatus_FLASH_AddressError Address is out of range.
kStatus_FLASH_AccessError Invalid instruction codes and out-of bound addresses.
kStatus_FLASH_ProtectionViolation The program/erase operation is requested to execute on protected areas.
kStatus_FLASH_CommandFailure Run-time error during command execution.
kStatus_FLASH_UnknownProperty Unknown property.
kStatus_FLASH_EraseKeyError API erase key is invalid.
kStatus_FLASH_RegionExecuteOnly The current region is execute-only.
kStatus_FLASH_ExecuteInRamFunctionNotReady Execute-in-RAM function is not available.
kStatus_FLASH_PartitionStatusUpdateFailure Failed to update partition status.
kStatus_FLASH_SetFlexramAsEepromError Failed to set FlexRAM as EEPROM.
kStatus_FLASH_RecoverFlexramAsRamError Failed to recover FlexRAM as RAM.
kStatus_FLASH_SetFlexramAsRamError Failed to set FlexRAM as RAM.
kStatus_FLASH_RecoverFlexramAsEepromError Failed to recover FlexRAM as EEPROM.
kStatus_FLASH_CommandNotSupported Flash API is not supported.
kStatus_FLASH_SwapSystemNotInUninitialized Swap system is not in an uninitialized state.
kStatus_FLASH_SwapIndicatorAddressError The swap indicator address is invalid.
kStatus_FLASH_ReadOnlyProperty The flash property is read-only.
kStatus_FLASH_InvalidPropertyValue The flash property value is out of range.
kStatus_FLASH_InvalidSpeculationOption The option of flash prefetch speculation is invalid.

14.4.3 enum _flash_driver_api_keys

Note

The resulting value is built with a byte order such that the string being readable in expected order when viewed in a hex editor, if the value is treated as a 32-bit little endian value.

Enumerator

kFLASH_ApiEraseKey Key value used to validate all flash erase APIs.

Enumeration Type Documentation

14.4.4 enum flash_margin_value_t

Enumerator

kFLASH_MarginValueNormal Use the 'normal' read level for 1s.

kFLASH_MarginValueUser Apply the 'User' margin to the normal read-1 level.

kFLASH_MarginValueFactory Apply the 'Factory' margin to the normal read-1 level.

kFLASH_MarginValueInvalid Not real margin level, Used to determine the range of valid margin level.

14.4.5 enum flash_security_state_t

Enumerator

kFLASH_SecurityStateNotSecure Flash is not secure.

kFLASH_SecurityStateBackdoorEnabled Flash backdoor is enabled.

kFLASH_SecurityStateBackdoorDisabled Flash backdoor is disabled.

14.4.6 enum flash_protection_state_t

Enumerator

kFLASH_ProtectionStateUnprotected Flash region is not protected.

kFLASH_ProtectionStateProtected Flash region is protected.

kFLASH_ProtectionStateMixed Flash is mixed with protected and unprotected region.

14.4.7 enum flash_execute_only_access_state_t

Enumerator

kFLASH_AccessStateUnLimited Flash region is unlimited.

kFLASH_AccessStateExecuteOnly Flash region is execute only.

kFLASH_AccessStateMixed Flash is mixed with unlimited and execute only region.

14.4.8 enum flash_property_tag_t

Enumerator

kFLASH_PropertyPflashSectorSize Pflash sector size property.

kFLASH_PropertyPflashTotalSize Pflash total size property.

kFLASH_PropertyPflashBlockSize Pflash block size property.
kFLASH_PropertyPflashBlockCount Pflash block count property.
kFLASH_PropertyPflashBlockBaseAddr Pflash block base address property.
kFLASH_PropertyPflashFacSupport Pflash fac support property.
kFLASH_PropertyPflashAccessSegmentSize Pflash access segment size property.
kFLASH_PropertyPflashAccessSegmentCount Pflash access segment count property.
kFLASH_PropertyFlexRamBlockBaseAddr FlexRam block base address property.
kFLASH_PropertyFlexRamTotalSize FlexRam total size property.
kFLASH_PropertyDflashSectorSize Dflash sector size property.
kFLASH_PropertyDflashTotalSize Dflash total size property.
kFLASH_PropertyDflashBlockSize Dflash block size property.
kFLASH_PropertyDflashBlockCount Dflash block count property.
kFLASH_PropertyDflashBlockBaseAddr Dflash block base address property.
kFLASH_PropertyEepromTotalSize EEPROM total size property.
kFLASH_PropertyFlashMemoryIndex Flash memory index property.
kFLASH_PropertyFlashCacheControllerIndex Flash cache controller index property.

14.4.9 enum _flash_execute_in_ram_function_constants

Enumerator

kFLASH_ExecuteInRamFunctionMaxSizeInWords The maximum size of execute-in-RAM function.
kFLASH_ExecuteInRamFunctionTotalNum Total number of execute-in-RAM functions.

14.4.10 enum flash_read_resource_option_t

Enumerator

kFLASH_ResourceOptionFlashIfr Select code for Program flash 0 IFR, Program flash swap 0 IFR, Data flash 0 IFR.
kFLASH_ResourceOptionVersionId Select code for the version ID.

14.4.11 enum _flash_read_resource_range

Enumerator

kFLASH_ResourceRangePflashIfrSizeInBytes Pflash IFR size in byte.
kFLASH_ResourceRangeVersionIdSizeInBytes Version ID IFR size in byte.
kFLASH_ResourceRangeVersionIdStart Version ID IFR start address.
kFLASH_ResourceRangeVersionIdEnd Version ID IFR end address.

Enumeration Type Documentation

kFLASH_ResourceRangePflashSwapIfrEnd Pflash swap IFR end address.

kFLASH_ResourceRangeDflashIfrStart Dflash IFR start address.

kFLASH_ResourceRangeDflashIfrEnd Dflash IFR end address.

14.4.12 enum _k3_flash_read_once_index

Enumerator

kFLASH_RecordIndexSwapAddr Index of Swap indicator address.

kFLASH_RecordIndexSwapEnable Index of Swap system enable.

kFLASH_RecordIndexSwapDisable Index of Swap system disable.

14.4.13 enum flash_flexram_function_option_t

Enumerator

kFLASH_FlexramFunctionOptionAvailableAsRam An option used to make FlexRAM available as RAM.

kFLASH_FlexramFunctionOptionAvailableForEeprom An option used to make FlexRAM available for EEPROM.

14.4.14 enum flash_swap_function_option_t

Enumerator

kFLASH_SwapFunctionOptionEnable An option used to enable the Swap function.

kFLASH_SwapFunctionOptionDisable An option used to disable the Swap function.

14.4.15 enum flash_swap_control_option_t

Enumerator

kFLASH_SwapControlOptionIntializeSystem An option used to initialize the Swap system.

kFLASH_SwapControlOptionSetInUpdateState An option used to set the Swap in an update state.

kFLASH_SwapControlOptionSetInCompleteState An option used to set the Swap in a complete state.

kFLASH_SwapControlOptionReportStatus An option used to report the Swap status.

kFLASH_SwapControlOptionDisableSystem An option used to disable the Swap status.

14.4.16 enum flash_swap_state_t

Enumerator

kFLASH_SwapStateUninitialized Flash Swap system is in an uninitialized state.

kFLASH_SwapStateReady Flash Swap system is in a ready state.

kFLASH_SwapStateUpdate Flash Swap system is in an update state.

kFLASH_SwapStateUpdateErased Flash Swap system is in an updateErased state.

kFLASH_SwapStateComplete Flash Swap system is in a complete state.

kFLASH_SwapStateDisabled Flash Swap system is in a disabled state.

14.4.17 enum flash_swap_block_status_t

Enumerator

kFLASH_SwapBlockStatusLowerHalfProgramBlocksAtZero Swap block status is that lower half program block at zero.

kFLASH_SwapBlockStatusUpperHalfProgramBlocksAtZero Swap block status is that upper half program block at zero.

14.4.18 enum flash_partition_flexram_load_option_t

Enumerator

kFLASH_PartitionFlexramLoadOptionLoadedWithValidEepromData FlexRAM is loaded with valid EEPROM data during reset sequence.

kFLASH_PartitionFlexramLoadOptionNotLoaded FlexRAM is not loaded during reset sequence.

14.4.19 enum flash_memory_index_t

Enumerator

kFLASH_MemoryIndexPrimaryFlash Current flash memory is primary flash.

kFLASH_MemoryIndexSecondaryFlash Current flash memory is secondary flash.

14.4.20 enum flash_cache_controller_index_t

Enumerator

kFLASH_CacheControllerIndexForCore0 Current flash cache controller is for core 0.

kFLASH_CacheControllerIndexForCore1 Current flash cache controller is for core 1.

Function Documentation

14.4.21 enum flash_cache_clear_process_t

Enumerator

kFLASH_CacheClearProcessPre Pre flash cache clear process.

kFLASH_CacheClearProcessPost Post flash cache clear process.

14.5 Function Documentation

14.5.1 status_t FLASH_Init (flash_config_t * config)

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

Parameters

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

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH-PartitionStatusUpdate-Failure</i>	Failed to update the partition status.

14.5.2 status_t FLASH_SetCallback (flash_config_t * config, flash_callback_t callback)

Parameters

<i>config</i>	Pointer to the storage for the driver runtime state.
<i>callback</i>	A callback function to be stored in the driver.

Return values

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

14.5.3 status_t FLASH_PrepareExecuteInRamFunctions (flash_config_t * config)

Parameters

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

Return values

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

14.5.4 status_t FLASH_EraseAll (flash_config_t * config, uint32_t key)

Parameters

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

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Erase-KeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.

Function Documentation

<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH-CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH-PartitionStatusUpdate-Failure</i>	Failed to update the partition status.

14.5.5 status_t FLASH_Erase (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, uint32_t *key*)

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

<i>config</i>	The pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be erased. The start address does not need to be sector-aligned but must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words) to be erased. Must be word-aligned.
<i>key</i>	The value used to validate all flash erase APIs.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH-AlignmentError</i>	The parameter is not aligned with the specified baseline.
<i>kStatus_FLASH_Address-Error</i>	The address is out of range.

<i>kStatus_FLASH_Erase-KeyError</i>	The API erase key is invalid.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH_-CommandFailure</i>	Run-time error during the command execution.

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

Parameters

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

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH_Erase-KeyError</i>	API erase key is invalid.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH_-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.

Function Documentation

<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.
<i>kStatus_FLASH_PartitionStatusUpdateFailure</i>	Failed to update the partition status.

Erases all program flash execute-only segments defined by the FXACC registers.

Parameters

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

Return values

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

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

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

Parameters

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

Return values

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

14.5.8 **status_t FLASH_ProgramOnce (flash_config_t * *config*, uint32_t *index*, uint32_t * *src*, uint32_t *lengthInBytes*)**

This function programs the Program Once Field with the desired data for a given flash area as determined by the index and length.

Parameters

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

Function Documentation

<i>index</i>	The index indicating which area of the Program Once Field to be programmed.
<i>src</i>	A pointer to the source buffer of data that is to be programmed into the Program Once Field.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

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

14.5.9 **status_t FLASH_ReadOnce (flash_config_t * *config*, uint32_t *index*, uint32_t * *dst*, uint32_t *lengthInBytes*)**

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

Parameters

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

<i>option</i>	The resource option which indicates which area should be read back.
---------------	---

Return values

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

Reads the Program Once Field through parameters.

This function reads the read once feild with given index and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>index</i>	The index indicating the area of program once field to be read.
<i>dst</i>	A pointer to the destination buffer of data that is used to store data to be read.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be programmed. Must be word-aligned.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_ExecuteInRamFunctionNotReady</i>	Execute-in-RAM function is not available.

Function Documentation

<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH-CommandFailure</i>	Run-time error during the command execution.

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

This function retrieves the current flash security status, including the security enabling state and the back-door key enabling state.

Parameters

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

Return values

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

14.5.11 **status_t FLASH_SecurityBypass (flash_config_t * *config*, const uint8_t * *backdoorKey*)**

If the MCU is in secured state, this function unsecures the MCU by comparing the provided backdoor key with ones in the flash configuration field.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>backdoorKey</i>	A pointer to the user buffer containing the backdoor key.

Return values

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

14.5.12 **status_t FLASH_VerifyEraseAll (flash_config_t * *config*, flash_margin_value_t *margin*)**

This function checks whether the flash is erased to the specified read margin level.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>margin</i>	Read margin choice.

Return values

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

Function Documentation

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

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

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

Parameters

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

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH-AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_Address-Error</i>	Address is out of range.
<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.

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

14.5.14 **status_t FLASH_VerifyProgram (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, const uint32_t * *expectedData*, flash_margin_value_t *margin*, uint32_t * *failedAddress*, uint32_t * *failedData*)**

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be verified. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be verified. Must be word-aligned.
<i>expectedData</i>	A pointer to the expected data that is to be verified against.
<i>margin</i>	Read margin choice.
<i>failedAddress</i>	A pointer to the returned failing address.
<i>failedData</i>	A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_Invalid-Argument</i>	An invalid argument is provided.
<i>kStatus_FLASH-AlignmentError</i>	Parameter is not aligned with specified baseline.
<i>kStatus_FLASH_Address-Error</i>	Address is out of range.

Function Documentation

<i>kStatus_FLASH_Execute-InRamFunctionNotReady</i>	Execute-in-RAM function is not available.
<i>kStatus_FLASH_Access-Error</i>	Invalid instruction codes and out-of bounds addresses.
<i>kStatus_FLASH-ProtectionViolation</i>	The program/erase operation is requested to execute on protected areas.
<i>kStatus_FLASH-CommandFailure</i>	Run-time error during the command execution.

14.5.15 **status_t FLASH_VerifyEraseAllExecuteOnlySegments (flash_config_t * config, flash_margin_value_t margin)**

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>margin</i>	Read margin choice.

Return values

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

14.5.16 **status_t FLASH_IsProtected (flash_config_t * config, uint32_t start, uint32_t lengthInBytes, flash_protection_state_t * protection_state)**

This function retrieves the current flash protect status for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be checked. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words) to be checked. Must be word-aligned.
<i>protection_state</i>	A pointer to the value returned for the current protection status code for the desired flash area.

Return values

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

14.5.17 **status_t FLASH_IsExecuteOnly (flash_config_t * *config*, uint32_t *start*, uint32_t *lengthInBytes*, flash_execute_only_access_state_t * *access_state*)**

This function retrieves the current flash access status for a given flash area as determined by the start address and length.

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>start</i>	The start address of the desired flash memory to be checked. Must be word-aligned.
<i>lengthInBytes</i>	The length, given in bytes (not words or long-words), to be checked. Must be word-aligned.
<i>access_state</i>	A pointer to the value returned for the current access status code for the desired flash area.

Return values

Function Documentation

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_AlignmentError</i>	The parameter is not aligned to the specified baseline.
<i>kStatus_FLASH_AddressError</i>	The address is out of range.

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

Parameters

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

Return values

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

14.5.19 **status_t FLASH_SetProperty (flash_config_t * *config*, flash_property_tag_t *whichProperty*, uint32_t *value*)**

Parameters

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

<i>value</i>	A to set for the desired flash property.
--------------	--

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_UnknownProperty</i>	An unknown property tag.
<i>kStatus_FLASH_InvalidPropertyValue</i>	An invalid property value.
<i>kStatus_FLASH_ReadOnlyProperty</i>	An read-only property tag.

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

Parameters

<i>config</i>	A pointer to storage for the driver runtime state.
<i>protectStatus</i>	The expected protect status to set to the PFlash protection register. Each bit is corresponding to protection of 1/32(64) of the total PFlash. The least significant bit is corresponding to the lowest address area of PFlash. The most significant bit is corresponding to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

<i>kStatus_FLASH_Success</i>	API was executed successfully.
<i>kStatus_FLASH_InvalidArgument</i>	An invalid argument is provided.
<i>kStatus_FLASH_CommandFailure</i>	Run-time error during command execution.

14.5.21 **status_t FLASH_PflashGetProtection (flash_config_t * *config*, pflash_protection_status_t * *protectStatus*)**

Function Documentation

Parameters

<i>config</i>	A pointer to the storage for the driver runtime state.
<i>protectStatus</i>	Protect status returned by the PFlash IP. Each bit is corresponding to the protection of 1/32(64) of the total PFlash. The least significant bit corresponds to the lowest address area of the PFlash. The most significant bit corresponds to the highest address area of PFlash. There are two possible cases as shown below: 0: this area is protected. 1: this area is unprotected.

Return values

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

Chapter 15

FTM: FlexTimer Driver

15.1 Overview

The MCUXpresso SDK provides a driver for the FlexTimer Module (FTM) of MCUXpresso SDK devices.

15.2 Function groups

The FTM driver supports the generation of PWM signals, input capture, dual edge capture, output compare, and quadrature decoder modes. The driver also supports configuring each of the FTM fault inputs.

15.2.1 Initialization and deinitialization

The function [FTM_Init\(\)](#) initializes the FTM with specified configurations. The function [FTM_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the FTM for the requested register update mode for registers with buffers. It also sets up the FTM's fault operation mode and FTM behavior in the BDM mode.

The function [FTM_Deinit\(\)](#) disables the FTM counter and turns off the module clock.

15.2.2 PWM Operations

The function [FTM_SetupPwm\(\)](#) sets up FTM channels for the PWM output. The function sets up the PWM signal properties for multiple channels. Each channel has its own duty cycle and level-mode specified. However, the same PWM period and PWM mode is applied to all channels requesting the PWM output. The signal duty cycle is provided as a percentage of the PWM period. Its value should be between 0 and 100 0=inactive signal (0% duty cycle) and 100=always active signal (100% duty cycle).

The function [FTM_UpdatePwmDutycycle\(\)](#) updates the PWM signal duty cycle of a particular FTM channel.

The function [FTM_UpdateChnlEdgeLevelSelect\(\)](#) updates the level select bits of a particular FTM channel. This can be used to disable the PWM output when making changes to the PWM signal.

15.2.3 Input capture operations

The function [FTM_SetupInputCapture\(\)](#) sets up an FTM channel for the input capture. The user can specify the capture edge and a filter value to be used when processing the input signal.

The function [FTM_SetupDualEdgeCapture\(\)](#) can be used to measure the pulse width of a signal. A channel pair is used during capture with the input signal coming through a channel n. The user can specify whether

Register Update

to use one-shot or continuous capture, the capture edge for each channel, and any filter value to be used when processing the input signal.

15.2.4 Output compare operations

The function `FTM_SetupOutputCompare()` sets up an FTM channel for the output comparison. The user can specify the channel output on a successful comparison and a comparison value.

15.2.5 Quad decode

The function `FTM_SetupQuadDecode()` sets up FTM channels 0 and 1 for quad decoding. The user can specify the quad decoding mode, polarity, and filter properties for each input signal.

15.2.6 Fault operation

The function `FTM_SetupFault()` sets up the properties for each fault. The user can specify the fault polarity and whether to use a filter on a fault input. The overall fault filter value and fault control mode are set up during initialization.

15.3 Register Update

Some of the FTM registers have buffers. The driver supports various methods to update these registers with the content of the register buffer. The registers can be updated using the PWM synchronized loading or an intermediate point loading. The update mechanism for register with buffers can be specified through the following fields available in the configuration structure.

```
uint32_t pwmSyncMode;  
uint32_t reloadPoints;
```

Multiple PWM synchronization update modes can be used by providing an OR'ed list of options available in the enumeration `ftm_pwm_sync_method_t` to the `pwmSyncMode` field.

When using an intermediate reload points, the PWM synchronization is not required. Multiple reload points can be used by providing an OR'ed list of options available in the enumeration `ftm_reload_point_t` to the `reloadPoints` field.

The driver initialization function sets up the appropriate bits in the FTM module based on the register update options selected.

If software PWM synchronization is used, the below function can be used to initiate a software trigger.

```
FTM_SetSoftwareTrigger(FTM0, true)
```

15.4 Typical use case

15.4.1 PWM output

Output a PWM signal on two FTM channels with different duty cycles. Periodically update the PWM signal duty cycle.

```
int main(void)
{
    bool brightnessUp = true; /* Indicates whether LEDs are brighter or dimmer. */
    ftm_config_t ftmInfo;
    uint8_t updatedDutycycle = 0U;
    ftm_chnl_pwm_signal_param_t ftmParam[2];

    /* Configures the FTM parameters with frequency 24 kHz */
    ftmParam[0].chnlNumber = (ftm_chnl_t)BOARD_FIRST_FTM_CHANNEL;
    ftmParam[0].level = kFTM_LowTrue;
    ftmParam[0].dutyCyclePercent = 0U;
    ftmParam[0].firstEdgeDelayPercent = 0U;

    ftmParam[1].chnlNumber = (ftm_chnl_t)BOARD_SECOND_FTM_CHANNEL;
    ftmParam[1].level = kFTM_LowTrue;
    ftmParam[1].dutyCyclePercent = 0U;
    ftmParam[1].firstEdgeDelayPercent = 0U;

    FTM_GetDefaultConfig(&ftmInfo);

    /* Initializes the FTM module. */
    FTM_Init(BOARD_FTM_BASEADDR, &ftmInfo);

    FTM_SetupPwm(BOARD_FTM_BASEADDR, ftmParam, 2U,
        kFTM_EdgeAlignedPwm, 24000U, FTM_SOURCE_CLOCK);
    FTM_StartTimer(BOARD_FTM_BASEADDR, kFTM_SystemClock);

    while (1)
    {
        /* Delays to check whether the LED brightness has changed. */
        delay();

        if (brightnessUp)
        {
            /* Increases the duty cycle until it reaches a limited value. */
            if (++updatedDutycycle == 100U)
            {
                brightnessUp = false;
            }
        }
        else
        {
            /* Decreases the duty cycle until it reaches a limited value. */
            if (--updatedDutycycle == 0U)
            {
                brightnessUp = true;
            }
        }

        /* Starts the PWM mode with an updated duty cycle. */
        FTM_UpdatePwmDutycycle(BOARD_FTM_BASEADDR, (
            ftm_chnl_t)BOARD_FIRST_FTM_CHANNEL, kFTM_EdgeAlignedPwm,
            updatedDutycycle);
        FTM_UpdatePwmDutycycle(BOARD_FTM_BASEADDR, (
            ftm_chnl_t)BOARD_SECOND_FTM_CHANNEL, kFTM_EdgeAlignedPwm,
            updatedDutycycle);

        /* Software trigger to update registers. */
        FTM_SetSoftwareTrigger(BOARD_FTM_BASEADDR, true);
    }
}
```

Typical use case

Data Structures

- struct `ftm_chnl_pwm_signal_param_t`
Options to configure a FTM channel's PWM signal. [More...](#)
- struct `ftm_dual_edge_capture_param_t`
FlexTimer dual edge capture parameters. [More...](#)
- struct `ftm_phase_params_t`
FlexTimer quadrature decode phase parameters. [More...](#)
- struct `ftm_fault_param_t`
Structure is used to hold the parameters to configure a FTM fault. [More...](#)
- struct `ftm_config_t`
FTM configuration structure. [More...](#)

Enumerations

- enum `ftm_chnl_t` {
 `kFTM_Chnl_0` = 0U,
 `kFTM_Chnl_1`,
 `kFTM_Chnl_2`,
 `kFTM_Chnl_3`,
 `kFTM_Chnl_4`,
 `kFTM_Chnl_5`,
 `kFTM_Chnl_6`,
 `kFTM_Chnl_7` }
List of FTM channels.
- enum `ftm_fault_input_t` {
 `kFTM_Fault_0` = 0U,
 `kFTM_Fault_1`,
 `kFTM_Fault_2`,
 `kFTM_Fault_3` }
List of FTM faults.
- enum `ftm_pwm_mode_t` {
 `kFTM_EdgeAlignedPwm` = 0U,
 `kFTM_CenterAlignedPwm`,
 `kFTM_CombinedPwm` }
FTM PWM operation modes.
- enum `ftm_pwm_level_select_t` {
 `kFTM_NoPwmSignal` = 0U,
 `kFTM_LowTrue`,
 `kFTM_HighTrue` }
FTM PWM output pulse mode: high-true, low-true or no output.
- enum `ftm_output_compare_mode_t` {
 `kFTM_NoOutputSignal` = (1U << FTM_CnSC_MSA_SHIFT),
 `kFTM_ToggleOnMatch` = ((1U << FTM_CnSC_MSA_SHIFT) | (1U << FTM_CnSC_ELSA_SHIFT)),
 `kFTM_ClearOnMatch` = ((1U << FTM_CnSC_MSA_SHIFT) | (2U << FTM_CnSC_ELSA_SHIFT)),
 `kFTM_SetOnMatch` = ((1U << FTM_CnSC_MSA_SHIFT) | (3U << FTM_CnSC_ELSA_SHIFT))

T)) }

FlexTimer output compare mode.

- enum `ftm_input_capture_edge_t` {
`kFTM_RisingEdge` = (1U << FTM_CnSC_ELSA_SHIFT),
`kFTM_FallingEdge` = (2U << FTM_CnSC_ELSA_SHIFT),
`kFTM_RiseAndFallEdge` = (3U << FTM_CnSC_ELSA_SHIFT) }

FlexTimer input capture edge.

- enum `ftm_dual_edge_capture_mode_t` {
`kFTM_OneShot` = 0U,
`kFTM_Continuous` = (1U << FTM_CnSC_MSA_SHIFT) }

FlexTimer dual edge capture modes.

- enum `ftm_quad_decode_mode_t` {
`kFTM_QuadPhaseEncode` = 0U,
`kFTM_QuadCountAndDir` }

FlexTimer quadrature decode modes.

- enum `ftm_phase_polarity_t` {
`kFTM_QuadPhaseNormal` = 0U,
`kFTM_QuadPhaseInvert` }

FlexTimer quadrature phase polarities.

- enum `ftm_deadtime_prescale_t` {
`kFTM_Deadtime_Prescale_1` = 1U,
`kFTM_Deadtime_Prescale_4`,
`kFTM_Deadtime_Prescale_16` }

FlexTimer pre-scaler factor for the dead time insertion.

- enum `ftm_clock_source_t` {
`kFTM_SystemClock` = 1U,
`kFTM_FixedClock`,
`kFTM_ExternalClock` }

FlexTimer clock source selection.

- enum `ftm_clock_prescale_t` {
`kFTM_Prescale_Divide_1` = 0U,
`kFTM_Prescale_Divide_2`,
`kFTM_Prescale_Divide_4`,
`kFTM_Prescale_Divide_8`,
`kFTM_Prescale_Divide_16`,
`kFTM_Prescale_Divide_32`,
`kFTM_Prescale_Divide_64`,
`kFTM_Prescale_Divide_128` }

FlexTimer pre-scaler factor selection for the clock source.

- enum `ftm_bdm_mode_t` {
`kFTM_BdmMode_0` = 0U,
`kFTM_BdmMode_1`,
`kFTM_BdmMode_2`,
`kFTM_BdmMode_3` }

Options for the FlexTimer behaviour in BDM Mode.

- enum `ftm_fault_mode_t` {

Typical use case

```
kFTM_Fault_Disable = 0U,  
kFTM_Fault_EvenChnls,  
kFTM_Fault_AllChnlsMan,  
kFTM_Fault_AllChnlsAuto }
```

Options for the FTM fault control mode.

- enum `ftm_external_trigger_t` {
 `kFTM_Chnl0Trigger` = (1U << 4),
 `kFTM_Chnl1Trigger` = (1U << 5),
 `kFTM_Chnl2Trigger` = (1U << 0),
 `kFTM_Chnl3Trigger` = (1U << 1),
 `kFTM_Chnl4Trigger` = (1U << 2),
 `kFTM_Chnl5Trigger` = (1U << 3),
 `kFTM_Chnl6Trigger`,
 `kFTM_Chnl7Trigger`,
 `kFTM_InitTrigger` = (1U << 6),
 `kFTM_ReloadInitTrigger` = (1U << 7) }

FTM external trigger options.

- enum `ftm_pwm_sync_method_t` {
 `kFTM_SoftwareTrigger` = FTM_SYNC_SWSYNC_MASK,
 `kFTM_HardwareTrigger_0` = FTM_SYNC_TRIG0_MASK,
 `kFTM_HardwareTrigger_1` = FTM_SYNC_TRIG1_MASK,
 `kFTM_HardwareTrigger_2` = FTM_SYNC_TRIG2_MASK }

FlexTimer PWM sync options to update registers with buffer.

- enum `ftm_reload_point_t` {
 `kFTM_Chnl0Match` = (1U << 0),
 `kFTM_Chnl1Match` = (1U << 1),
 `kFTM_Chnl2Match` = (1U << 2),
 `kFTM_Chnl3Match` = (1U << 3),
 `kFTM_Chnl4Match` = (1U << 4),
 `kFTM_Chnl5Match` = (1U << 5),
 `kFTM_Chnl6Match` = (1U << 6),
 `kFTM_Chnl7Match` = (1U << 7),
 `kFTM_CntMax` = (1U << 8),
 `kFTM_CntMin` = (1U << 9),
 `kFTM_HalfCycMatch` = (1U << 10) }

FTM options available as loading point for register reload.

- enum `ftm_interrupt_enable_t` {

```

kFTM_Chnl0InterruptEnable = (1U << 0),
kFTM_Chnl1InterruptEnable = (1U << 1),
kFTM_Chnl2InterruptEnable = (1U << 2),
kFTM_Chnl3InterruptEnable = (1U << 3),
kFTM_Chnl4InterruptEnable = (1U << 4),
kFTM_Chnl5InterruptEnable = (1U << 5),
kFTM_Chnl6InterruptEnable = (1U << 6),
kFTM_Chnl7InterruptEnable = (1U << 7),
kFTM_FaultInterruptEnable = (1U << 8),
kFTM_TimeOverflowInterruptEnable = (1U << 9),
kFTM_ReloadInterruptEnable = (1U << 10) }

```

List of FTM interrupts.

- enum `ftm_status_flags_t` {
`kFTM_Chnl0Flag` = (1U << 0),
`kFTM_Chnl1Flag` = (1U << 1),
`kFTM_Chnl2Flag` = (1U << 2),
`kFTM_Chnl3Flag` = (1U << 3),
`kFTM_Chnl4Flag` = (1U << 4),
`kFTM_Chnl5Flag` = (1U << 5),
`kFTM_Chnl6Flag` = (1U << 6),
`kFTM_Chnl7Flag` = (1U << 7),
`kFTM_FaultFlag` = (1U << 8),
`kFTM_TimeOverflowFlag` = (1U << 9),
`kFTM_ChnlTriggerFlag` = (1U << 10),
`kFTM_ReloadFlag` = (1U << 11) }
- *List of FTM flags.*
- enum `_ftm_quad_decoder_flags` {
`kFTM_QuadDecoderCountingIncreaseFlag` = `FTM_QDCTRL_QUADIR_MASK`,
`kFTM_QuadDecoderCountingOverflowOnTopFlag` = `FTM_QDCTRL_TOFDIR_MASK` }

List of FTM Quad Decoder flags.

Functions

- void `FTM_SetupFault` (FTM_Type *base, `ftm_fault_input_t` faultNumber, const `ftm_fault_param_t` *faultParams)
Sets up the working of the FTM fault protection.
- static void `FTM_SetGlobalTimeBaseOutputEnable` (FTM_Type *base, bool enable)
Enables or disables the FTM global time base signal generation to other FTMs.
- static void `FTM_SetOutputMask` (FTM_Type *base, `ftm_chnl_t` chnlNumber, bool mask)
Sets the FTM peripheral timer channel output mask.
- static void `FTM_SetSoftwareTrigger` (FTM_Type *base, bool enable)
Enables or disables the FTM software trigger for PWM synchronization.
- static void `FTM_SetWriteProtection` (FTM_Type *base, bool enable)
Enables or disables the FTM write protection.

Driver version

- #define `FSL_FTM_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 2))

Typical use case

Version 2.0.2.

Initialization and deinitialization

- status_t [FTM_Init](#) (FTM_Type *base, const [ftm_config_t](#) *config)
Ungates the FTM clock and configures the peripheral for basic operation.
- void [FTM_Deinit](#) (FTM_Type *base)
Gates the FTM clock.
- void [FTM_GetDefaultConfig](#) ([ftm_config_t](#) *config)
Fills in the FTM configuration structure with the default settings.

Channel mode operations

- status_t [FTM_SetupPwm](#) (FTM_Type *base, const [ftm_chnl_pwm_signal_param_t](#) *chnlParams, uint8_t numOfChnls, [ftm_pwm_mode_t](#) mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz)
Configures the PWM signal parameters.
- void [FTM_UpdatePwmDutycycle](#) (FTM_Type *base, [ftm_chnl_t](#) chnlNumber, [ftm_pwm_mode_t](#) currentPwmMode, uint8_t dutyCyclePercent)
Updates the duty cycle of an active PWM signal.
- void [FTM_UpdateChnlEdgeLevelSelect](#) (FTM_Type *base, [ftm_chnl_t](#) chnlNumber, uint8_t level)
Updates the edge level selection for a channel.
- void [FTM_SetupInputCapture](#) (FTM_Type *base, [ftm_chnl_t](#) chnlNumber, [ftm_input_capture_edge_t](#) captureMode, uint32_t filterValue)
Enables capturing an input signal on the channel using the function parameters.
- void [FTM_SetupOutputCompare](#) (FTM_Type *base, [ftm_chnl_t](#) chnlNumber, [ftm_output_compare_mode_t](#) compareMode, uint32_t compareValue)
Configures the FTM to generate timed pulses.
- void [FTM_SetupDualEdgeCapture](#) (FTM_Type *base, [ftm_chnl_t](#) chnlPairNumber, const [ftm_dual_edge_capture_param_t](#) *edgeParam, uint32_t filterValue)
Configures the dual edge capture mode of the FTM.

Interrupt Interface

- void [FTM_EnableInterrupts](#) (FTM_Type *base, uint32_t mask)
Enables the selected FTM interrupts.
- void [FTM_DisableInterrupts](#) (FTM_Type *base, uint32_t mask)
Disables the selected FTM interrupts.
- uint32_t [FTM_GetEnabledInterrupts](#) (FTM_Type *base)
Gets the enabled FTM interrupts.

Status Interface

- uint32_t [FTM_GetStatusFlags](#) (FTM_Type *base)
Gets the FTM status flags.
- void [FTM_ClearStatusFlags](#) (FTM_Type *base, uint32_t mask)
Clears the FTM status flags.

Read and write the timer period

- static void [FTM_SetTimerPeriod](#) (FTM_Type *base, uint32_t ticks)

- *Sets the timer period in units of ticks.*
- static uint32_t [FTM_GetCurrentTimerCount](#) (FTM_Type *base)
Reads the current timer counting value.

Timer Start and Stop

- static void [FTM_StartTimer](#) (FTM_Type *base, [ftm_clock_source_t](#) clockSource)
Starts the FTM counter.
- static void [FTM_StopTimer](#) (FTM_Type *base)
Stops the FTM counter.

Software output control

- static void [FTM_SetSoftwareCtrlEnable](#) (FTM_Type *base, [ftm_chnl_t](#) chnlNumber, bool value)
Enables or disables the channel software output control.
- static void [FTM_SetSoftwareCtrlVal](#) (FTM_Type *base, [ftm_chnl_t](#) chnlNumber, bool value)
Sets the channel software output control value.

Channel pair operations

- static void [FTM_SetFaultControlEnable](#) (FTM_Type *base, [ftm_chnl_t](#) chnlPairNumber, bool value)
This function enables/disables the fault control in a channel pair.
- static void [FTM_SetDeadTimeEnable](#) (FTM_Type *base, [ftm_chnl_t](#) chnlPairNumber, bool value)
This function enables/disables the dead time insertion in a channel pair.
- static void [FTM_SetComplementaryEnable](#) (FTM_Type *base, [ftm_chnl_t](#) chnlPairNumber, bool value)
This function enables/disables complementary mode in a channel pair.
- static void [FTM_SetInvertEnable](#) (FTM_Type *base, [ftm_chnl_t](#) chnlPairNumber, bool value)
This function enables/disables inverting control in a channel pair.

Quad Decoder

- void [FTM_SetupQuadDecode](#) (FTM_Type *base, const [ftm_phase_params_t](#) *phaseAParams, const [ftm_phase_params_t](#) *phaseBParams, [ftm_quad_decode_mode_t](#) quadMode)
Configures the parameters and activates the quadrature decoder mode.
- static uint32_t [FTM_GetQuadDecoderFlags](#) (FTM_Type *base)
Gets the FTM Quad Decoder flags.
- static void [FTM_SetQuadDecoderModuloValue](#) (FTM_Type *base, uint32_t startValue, uint32_t overValue)
Sets the modulo values for Quad Decoder.
- static uint32_t [FTM_GetQuadDecoderCounterValue](#) (FTM_Type *base)
Gets the current Quad Decoder counter value.
- static void [FTM_ClearQuadDecoderCounterValue](#) (FTM_Type *base)
Clears the current Quad Decoder counter value.

15.5 Data Structure Documentation

15.5.1 struct ftm_chnl_pwm_signal_param_t

Data Fields

- [ftm_chnl_t chnlNumber](#)
The channel/channel pair number.
- [ftm_pwm_level_select_t level](#)
PWM output active level select.
- [uint8_t dutyCyclePercent](#)
PWM pulse width, value should be between 0 to 100 0 = inactive signal(0% duty cycle)...
- [uint8_t firstEdgeDelayPercent](#)
Used only in combined PWM mode to generate an asymmetrical PWM.

15.5.1.0.0.43 Field Documentation

15.5.1.0.0.43.1 ftm_chnl_t ftm_chnl_pwm_signal_param_t::chnlNumber

In combined mode, this represents the channel pair number.

15.5.1.0.0.43.2 ftm_pwm_level_select_t ftm_chnl_pwm_signal_param_t::level

15.5.1.0.0.43.3 uint8_t ftm_chnl_pwm_signal_param_t::dutyCyclePercent

100 = always active signal (100% duty cycle).

15.5.1.0.0.43.4 uint8_t ftm_chnl_pwm_signal_param_t::firstEdgeDelayPercent

Specifies the delay to the first edge in a PWM period. If unsure leave as 0; Should be specified as a percentage of the PWM period

15.5.2 struct ftm_dual_edge_capture_param_t

Data Fields

- [ftm_dual_edge_capture_mode_t mode](#)
Dual Edge Capture mode.
- [ftm_input_capture_edge_t currChanEdgeMode](#)
Input capture edge select for channel n.
- [ftm_input_capture_edge_t nextChanEdgeMode](#)
Input capture edge select for channel n+1.

15.5.3 struct ftm_phase_params_t

Data Fields

- bool [enablePhaseFilter](#)
True: enable phase filter; false: disable filter.
- uint32_t [phaseFilterVal](#)
Filter value, used only if phase filter is enabled.
- [ftm_phase_polarity_t](#) [phasePolarity](#)
Phase polarity.

15.5.4 struct ftm_fault_param_t

Data Fields

- bool [enableFaultInput](#)
True: Fault input is enabled; false: Fault input is disabled.
- bool [faultLevel](#)
True: Fault polarity is active low; in other words, '0' indicates a fault; False: Fault polarity is active high.
- bool [useFaultFilter](#)
True: Use the filtered fault signal; False: Use the direct path from fault input.

15.5.5 struct ftm_config_t

This structure holds the configuration settings for the FTM peripheral. To initialize this structure to reasonable defaults, call the [FTM_GetDefaultConfig\(\)](#) function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

Data Fields

- [ftm_clock_prescale_t](#) [prescale](#)
FTM clock prescale value.
- [ftm_bdm_mode_t](#) [bdmMode](#)
FTM behavior in BDM mode.
- uint32_t [pwmSyncMode](#)
Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration [ftm_pwm_sync_method_t](#).
- uint32_t [reloadPoints](#)
FTM reload points; When using this, the PWM synchronization is not required.
- [ftm_fault_mode_t](#) [faultMode](#)
FTM fault control mode.
- uint8_t [faultFilterValue](#)
Fault input filter value.

Enumeration Type Documentation

- [ftm_deadtime_prescale_t](#) `deadTimePrescale`
The dead time prescalar value.
- [uint32_t](#) `deadTimeValue`
The dead time value `deadTimeValue`'s available range is 0-1023 when register has `DTVALEX`, otherwise its available range is 0-63.
- [uint32_t](#) `extTriggers`
External triggers to enable.
- [uint8_t](#) `chnlInitState`
Defines the initialization value of the channels in `OUTINT` register.
- [uint8_t](#) `chnlPolarity`
Defines the output polarity of the channels in `POL` register.
- [bool](#) `useGlobalTimeBase`
True: Use of an external global time base is enabled; False: disabled.

15.5.5.0.0.44 Field Documentation

15.5.5.0.0.44.1 [uint32_t](#) `ftm_config_t::pwmSyncMode`

15.5.5.0.0.44.2 [uint32_t](#) `ftm_config_t::reloadPoints`

Multiple reload points can be used by providing an OR'ed list of options available in enumeration [ftm_reload_point_t](#).

15.5.5.0.0.44.3 [uint32_t](#) `ftm_config_t::deadTimeValue`

15.5.5.0.0.44.4 [uint32_t](#) `ftm_config_t::extTriggers`

Multiple trigger sources can be enabled by providing an OR'ed list of options available in enumeration [ftm_external_trigger_t](#).

15.6 Enumeration Type Documentation

15.6.1 [enum](#) `ftm_chnl_t`

Note

Actual number of available channels is SoC dependent

Enumerator

- `kFTM_Chnl_0`* FTM channel number 0.
- `kFTM_Chnl_1`* FTM channel number 1.
- `kFTM_Chnl_2`* FTM channel number 2.
- `kFTM_Chnl_3`* FTM channel number 3.
- `kFTM_Chnl_4`* FTM channel number 4.
- `kFTM_Chnl_5`* FTM channel number 5.
- `kFTM_Chnl_6`* FTM channel number 6.
- `kFTM_Chnl_7`* FTM channel number 7.

15.6.2 enum ftm_fault_input_t

Enumerator

kFTM_Fault_0 FTM fault 0 input pin.
kFTM_Fault_1 FTM fault 1 input pin.
kFTM_Fault_2 FTM fault 2 input pin.
kFTM_Fault_3 FTM fault 3 input pin.

15.6.3 enum ftm_pwm_mode_t

Enumerator

kFTM_EdgeAlignedPwm Edge-aligned PWM.
kFTM_CenterAlignedPwm Center-aligned PWM.
kFTM_CombinedPwm Combined PWM.

15.6.4 enum ftm_pwm_level_select_t

Enumerator

kFTM_NoPwmSignal No PWM output on pin.
kFTM_LowTrue Low true pulses.
kFTM_HighTrue High true pulses.

15.6.5 enum ftm_output_compare_mode_t

Enumerator

kFTM_NoOutputSignal No channel output when counter reaches CnV.
kFTM_ToggleOnMatch Toggle output.
kFTM_ClearOnMatch Clear output.
kFTM_SetOnMatch Set output.

15.6.6 enum ftm_input_capture_edge_t

Enumerator

kFTM_RisingEdge Capture on rising edge only.
kFTM_FallingEdge Capture on falling edge only.
kFTM_RiseAndFallEdge Capture on rising or falling edge.

Enumeration Type Documentation

15.6.7 enum ftm_dual_edge_capture_mode_t

Enumerator

kFTM_OneShot One-shot capture mode.
kFTM_Continuous Continuous capture mode.

15.6.8 enum ftm_quad_decode_mode_t

Enumerator

kFTM_QuadPhaseEncode Phase A and Phase B encoding mode.
kFTM_QuadCountAndDir Count and direction encoding mode.

15.6.9 enum ftm_phase_polarity_t

Enumerator

kFTM_QuadPhaseNormal Phase input signal is not inverted.
kFTM_QuadPhaseInvert Phase input signal is inverted.

15.6.10 enum ftm_deadtime_prescale_t

Enumerator

kFTM_Deadtime_Prescale_1 Divide by 1.
kFTM_Deadtime_Prescale_4 Divide by 4.
kFTM_Deadtime_Prescale_16 Divide by 16.

15.6.11 enum ftm_clock_source_t

Enumerator

kFTM_SystemClock System clock selected.
kFTM_FixedClock Fixed frequency clock.
kFTM_ExternalClock External clock.

15.6.12 enum ftm_clock_prescale_t

Enumerator

kFTM_Prescale_Divide_1 Divide by 1.
kFTM_Prescale_Divide_2 Divide by 2.
kFTM_Prescale_Divide_4 Divide by 4.
kFTM_Prescale_Divide_8 Divide by 8.
kFTM_Prescale_Divide_16 Divide by 16.
kFTM_Prescale_Divide_32 Divide by 32.
kFTM_Prescale_Divide_64 Divide by 64.
kFTM_Prescale_Divide_128 Divide by 128.

15.6.13 enum ftm_bdm_mode_t

Enumerator

kFTM_BdmMode_0 FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
kFTM_BdmMode_1 FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
kFTM_BdmMode_2 FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers.
kFTM_BdmMode_3 FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode.

15.6.14 enum ftm_fault_mode_t

Enumerator

kFTM_Fault_Disable Fault control is disabled for all channels.
kFTM_Fault_EvenChnls Enabled for even channels only(0,2,4,6) with manual fault clearing.
kFTM_Fault_AllChnlsMan Enabled for all channels with manual fault clearing.
kFTM_Fault_AllChnlsAuto Enabled for all channels with automatic fault clearing.

15.6.15 enum ftm_external_trigger_t

Enumeration Type Documentation

Note

Actual available external trigger sources are SoC-specific

Enumerator

- kFTM_Chnl0Trigger*** Generate trigger when counter equals chnl 0 CnV reg.
- kFTM_Chnl1Trigger*** Generate trigger when counter equals chnl 1 CnV reg.
- kFTM_Chnl2Trigger*** Generate trigger when counter equals chnl 2 CnV reg.
- kFTM_Chnl3Trigger*** Generate trigger when counter equals chnl 3 CnV reg.
- kFTM_Chnl4Trigger*** Generate trigger when counter equals chnl 4 CnV reg.
- kFTM_Chnl5Trigger*** Generate trigger when counter equals chnl 5 CnV reg.
- kFTM_Chnl6Trigger*** Available on certain SoC's, generate trigger when counter equals chnl 6 CnV reg.
- kFTM_Chnl7Trigger*** Available on certain SoC's, generate trigger when counter equals chnl 7 CnV reg.
- kFTM_InitTrigger*** Generate Trigger when counter is updated with CNTIN.
- kFTM_ReloadInitTrigger*** Available on certain SoC's, trigger on reload point.

15.6.16 enum ftm_pwm_sync_method_t

Enumerator

- kFTM_SoftwareTrigger*** Software triggers PWM sync.
- kFTM_HardwareTrigger_0*** Hardware trigger 0 causes PWM sync.
- kFTM_HardwareTrigger_1*** Hardware trigger 1 causes PWM sync.
- kFTM_HardwareTrigger_2*** Hardware trigger 2 causes PWM sync.

15.6.17 enum ftm_reload_point_t

Note

Actual available reload points are SoC-specific

Enumerator

- kFTM_Chnl0Match*** Channel 0 match included as a reload point.
- kFTM_Chnl1Match*** Channel 1 match included as a reload point.
- kFTM_Chnl2Match*** Channel 2 match included as a reload point.
- kFTM_Chnl3Match*** Channel 3 match included as a reload point.
- kFTM_Chnl4Match*** Channel 4 match included as a reload point.
- kFTM_Chnl5Match*** Channel 5 match included as a reload point.
- kFTM_Chnl6Match*** Channel 6 match included as a reload point.
- kFTM_Chnl7Match*** Channel 7 match included as a reload point.

kFTM_CntMax Use in up-down count mode only, reload when counter reaches the maximum value.

kFTM_CntMin Use in up-down count mode only, reload when counter reaches the minimum value.

kFTM_HalfCycMatch Available on certain SoC's, half cycle match reload point.

15.6.18 enum ftm_interrupt_enable_t

Note

Actual available interrupts are SoC-specific

Enumerator

kFTM_Chnl0InterruptEnable Channel 0 interrupt.
kFTM_Chnl1InterruptEnable Channel 1 interrupt.
kFTM_Chnl2InterruptEnable Channel 2 interrupt.
kFTM_Chnl3InterruptEnable Channel 3 interrupt.
kFTM_Chnl4InterruptEnable Channel 4 interrupt.
kFTM_Chnl5InterruptEnable Channel 5 interrupt.
kFTM_Chnl6InterruptEnable Channel 6 interrupt.
kFTM_Chnl7InterruptEnable Channel 7 interrupt.
kFTM_FaultInterruptEnable Fault interrupt.
kFTM_TimeOverflowInterruptEnable Time overflow interrupt.
kFTM_ReloadInterruptEnable Reload interrupt; Available only on certain SoC's.

15.6.19 enum ftm_status_flags_t

Note

Actual available flags are SoC-specific

Enumerator

kFTM_Chnl0Flag Channel 0 Flag.
kFTM_Chnl1Flag Channel 1 Flag.
kFTM_Chnl2Flag Channel 2 Flag.
kFTM_Chnl3Flag Channel 3 Flag.
kFTM_Chnl4Flag Channel 4 Flag.
kFTM_Chnl5Flag Channel 5 Flag.
kFTM_Chnl6Flag Channel 6 Flag.
kFTM_Chnl7Flag Channel 7 Flag.
kFTM_FaultFlag Fault Flag.

Function Documentation

kFTM_TimeOverflowFlag Time overflow Flag.

kFTM_ChnlTriggerFlag Channel trigger Flag.

kFTM_ReloadFlag Reload Flag; Available only on certain SoC's.

15.6.20 enum _ftm_quad_decoder_flags

Enumerator

kFTM_QuadDecoderCountingIncreaseFlag Counting direction is increasing (FTM counter increment), or the direction is decreasing.

kFTM_QuadDecoderCountingOverflowOnTopFlag Indicates if the TOF bit was set on the top or the bottom of counting.

15.7 Function Documentation

15.7.1 status_t FTM_Init (FTM_Type * *base*, const ftm_config_t * *config*)

Note

This API should be called at the beginning of the application which is using the FTM driver.

Parameters

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

Returns

kStatus_Success indicates success; Else indicates failure.

15.7.2 void FTM_Deinit (FTM_Type * *base*)

Parameters

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

15.7.3 void FTM_GetDefaultConfig (ftm_config_t * *config*)

The default values are:

```

* config->prescale = kFTM_Prescale_Divide_1;
* config->bdmMode = kFTM_BdmMode_0;
* config->pwmSyncMode = kFTM_SoftwareTrigger;
* config->reloadPoints = 0;
* config->faultMode = kFTM_Fault_Disable;
* config->faultFilterValue = 0;
* config->deadTimePrescale = kFTM_Deadtime_Prescale_1;
* config->deadTimeValue = 0;
* config->extTriggers = 0;
* config->chnlInitState = 0;
* config->chnlPolarity = 0;
* config->useGlobalTimeBase = false;
*

```

Parameters

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

15.7.4 **status_t FTM_SetupPwm (FTM_Type * *base*, const ftm_chnl_pwm_signal_param_t * *chnlParams*, uint8_t *numOfChnls*, ftm_pwm_mode_t *mode*, uint32_t *pwmFreq_Hz*, uint32_t *srcClock_Hz*)**

Call this function to configure the PWM signal period, mode, duty cycle, and edge. Use this function to configure all FTM channels that are used to output a PWM signal.

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlParams</i>	Array of PWM channel parameters to configure the channel(s)
<i>numOfChnls</i>	Number of channels to configure; This should be the size of the array passed in
<i>mode</i>	PWM operation mode, options available in enumeration ftm_pwm_mode_t
<i>pwmFreq_Hz</i>	PWM signal frequency in Hz
<i>srcClock_Hz</i>	FTM counter clock in Hz

Returns

kStatus_Success if the PWM setup was successful kStatus_Error on failure

15.7.5 **void FTM_UpdatePwmDutycycle (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_pwm_mode_t *currentPwmMode*, uint8_t *dutyCyclePercent*)**

Function Documentation

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel/channel pair number. In combined mode, this represents the channel pair number
<i>currentPwm-Mode</i>	The current PWM mode set during PWM setup
<i>dutyCycle-Percent</i>	New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

15.7.6 void FTM_UpdateChnlEdgeLevelSelect (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, uint8_t *level*)

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel number
<i>level</i>	The level to be set to the ELSnB:ELSnA field; Valid values are 00, 01, 10, 11. See the Kinetis SoC reference manual for details about this field.

15.7.7 void FTM_SetupInputCapture (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_input_capture_edge_t *captureMode*, uint32_t *filterValue*)

When the edge specified in the captureMode argument occurs on the channel, the FTM counter is captured into the CnV register. The user has to read the CnV register separately to get this value. The filter function is disabled if the filterVal argument passed in is 0. The filter function is available only for channels 0, 1, 2, 3.

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel number
<i>captureMode</i>	Specifies which edge to capture

<i>filterValue</i>	Filter value, specify 0 to disable filter. Available only for channels 0-3.
--------------------	---

15.7.8 void FTM_SetupOutputCompare (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, ftm_output_compare_mode_t *compareMode*, uint32_t *compareValue*)

When the FTM counter matches the value of compareVal argument (this is written into CnV reg), the channel output is changed based on what is specified in the compareMode argument.

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	The channel number
<i>compareMode</i>	Action to take on the channel output when the compare condition is met
<i>compareValue</i>	Value to be programmed in the CnV register.

15.7.9 void FTM_SetupDualEdgeCapture (FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, const ftm_dual_edge_capture_param_t * *edgeParam*, uint32_t *filterValue*)

This function sets up the dual edge capture mode on a channel pair. The capture edge for the channel pair and the capture mode (one-shot or continuous) is specified in the parameter argument. The filter function is disabled if the filterVal argument passed is zero. The filter function is available only on channels 0 and 2. The user has to read the channel CnV registers separately to get the capture values.

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>edgeParam</i>	Sets up the dual edge capture function
<i>filterValue</i>	Filter value, specify 0 to disable filter. Available only for channel pair 0 and 1.

15.7.10 void FTM_SetupFault (FTM_Type * *base*, ftm_fault_input_t *faultNumber*, const ftm_fault_param_t * *faultParams*)

FTM can have up to 4 fault inputs. This function sets up fault parameters, fault level, and a filter.

Function Documentation

Parameters

<i>base</i>	FTM peripheral base address
<i>faultNumber</i>	FTM fault to configure.
<i>faultParams</i>	Parameters passed in to set up the fault

15.7.11 void FTM_EnableInterrupts (FTM_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	FTM peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration ftm_interrupt_enable_t

15.7.12 void FTM_DisableInterrupts (FTM_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	FTM peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration ftm_interrupt_enable_t

15.7.13 uint32_t FTM_GetEnabledInterrupts (FTM_Type * *base*)

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [ftm_interrupt_enable_t](#)

15.7.14 uint32_t FTM_GetStatusFlags (FTM_Type * *base*)

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration [ftm_status_flags_t](#)

15.7.15 void FTM_ClearStatusFlags (FTM_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	FTM peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration ftm_status_flags_t

15.7.16 static void FTM_SetTimerPeriod (FTM_Type * *base*, uint32_t *ticks*) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note

1. This API allows the user to use the FTM module as a timer. Do not mix usage of this API with FTM's PWM setup API's.
2. Call the utility macros provided in the fsl_common.h to convert usec or msec to ticks.

Parameters

<i>base</i>	FTM peripheral base address
<i>ticks</i>	A timer period in units of ticks, which should be equal or greater than 1.

15.7.17 static uint32_t FTM_GetCurrentTimerCount (FTM_Type * *base*) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Function Documentation

Note

Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

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

Returns

The current counter value in ticks

15.7.18 static void FTM_StartTimer (FTM_Type * *base*, ftm_clock_source_t *clockSource*) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>clockSource</i>	FTM clock source; After the clock source is set, the counter starts running.

15.7.19 static void FTM_StopTimer (FTM_Type * *base*) [inline], [static]

Parameters

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

15.7.20 static void FTM_SetSoftwareCtrlEnable (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, bool *value*) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	Channel to be enabled or disabled

<i>value</i>	true: channel output is affected by software output control false: channel output is unaffected by software output control
--------------	---

15.7.21 static void FTM_SetSoftwareCtrlVal (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, bool *value*) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address.
<i>chnlNumber</i>	Channel to be configured
<i>value</i>	true to set 1, false to set 0

15.7.22 static void FTM_SetGlobalTimeBaseOutputEnable (FTM_Type * *base*, bool *enable*) [inline], [static]

Parameters

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

15.7.23 static void FTM_SetOutputMask (FTM_Type * *base*, ftm_chnl_t *chnlNumber*, bool *mask*) [inline], [static]

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlNumber</i>	Channel to be configured
<i>mask</i>	true: masked, channel is forced to its inactive state; false: unmasked

15.7.24 static void FTM_SetFaultControlEnable (FTM_Type * *base*, ftm_chnl_t *chnlPairNumber*, bool *value*) [inline], [static]

Function Documentation

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: Enable fault control for this channel pair; false: No fault control

15.7.25 `static void FTM_SetDeadTimeEnable (FTM_Type * base, ftm_chnl_t chnlPairNumber, bool value) [inline], [static]`

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: Insert dead time in this channel pair; false: No dead time inserted

15.7.26 `static void FTM_SetComplementaryEnable (FTM_Type * base, ftm_chnl_t chnlPairNumber, bool value) [inline], [static]`

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair-Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: enable complementary mode; false: disable complementary mode

15.7.27 `static void FTM_SetInvertEnable (FTM_Type * base, ftm_chnl_t chnlPairNumber, bool value) [inline], [static]`

Parameters

<i>base</i>	FTM peripheral base address
<i>chnlPair- Number</i>	The FTM channel pair number; options are 0, 1, 2, 3
<i>value</i>	true: enable inverting; false: disable inverting

15.7.28 void FTM_SetupQuadDecode (FTM_Type * *base*, const ftm_phase_params_t * *phaseAParams*, const ftm_phase_params_t * *phaseBParams*, ftm_quad_decode_mode_t *quadMode*)

Parameters

<i>base</i>	FTM peripheral base address
<i>phaseAParams</i>	Phase A configuration parameters
<i>phaseBParams</i>	Phase B configuration parameters
<i>quadMode</i>	Selects encoding mode used in quadrature decoder mode

**15.7.29 static uint32_t FTM_GetQuadDecoderFlags (FTM_Type * *base*)
[inline], [static]**

Parameters

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

Returns

Flag mask of FTM Quad Decoder, see [_ftm_quad_decoder_flags](#).

15.7.30 static void FTM_SetQuadDecoderModuloValue (FTM_Type * *base*, uint32_t *startValue*, uint32_t *overValue*) [inline], [static]

The modulo values configure the minimum and maximum values that the Quad decoder counter can reach. After the counter goes over, the counter value goes to the other side and decrease/increase again.

Function Documentation

Parameters

<i>base</i>	FTM peripheral base address.
<i>startValue</i>	The low limit value for Quad Decoder counter.
<i>overValue</i>	The high limit value for Quad Decoder counter.

15.7.31 `static uint32_t FTM_GetQuadDecoderCounterValue (FTM_Type * base)
[inline], [static]`

Parameters

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

Returns

Current quad Decoder counter value.

15.7.32 `static void FTM_ClearQuadDecoderCounterValue (FTM_Type * base)
[inline], [static]`

The counter is set as the initial value.

Parameters

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

15.7.33 `static void FTM_SetSoftwareTrigger (FTM_Type * base, bool enable)
[inline], [static]`

Parameters

<i>base</i>	FTM peripheral base address
<i>enable</i>	true: software trigger is selected, false: software trigger is not selected

15.7.34 `static void FTM_SetWriteProtection (FTM_Type * base, bool enable)
[inline], [static]`

Parameters

<i>base</i>	FTM peripheral base address
<i>enable</i>	true: Write-protection is enabled, false: Write-protection is disabled

Chapter 16

GPIO: General-Purpose Input/Output Driver

16.1 Overview

Modules

- [FGPIO Driver](#)
- [GPIO Driver](#)

Data Structures

- struct [gpio_pin_config_t](#)
The GPIO pin configuration structure. [More...](#)

Enumerations

- enum [gpio_pin_direction_t](#) {
 [kGPIO_DigitalInput](#) = 0U,
 [kGPIO_DigitalOutput](#) = 1U }
GPIO direction definition.

Driver version

- #define [FSL_GPIO_DRIVER_VERSION](#) ([MAKE_VERSION](#)(2, 1, 1))
GPIO driver version 2.1.1.

16.2 Data Structure Documentation

16.2.1 struct [gpio_pin_config_t](#)

Each pin can only be configured as either an output pin or an input pin at a time. If configured as an input pin, leave the outputConfig unused. Note that in some use cases, the corresponding port property should be configured in advance with the `PORT_SetPinConfig()`.

Data Fields

- [gpio_pin_direction_t](#) `pinDirection`
GPIO direction, input or output.
- `uint8_t` [outputLogic](#)
Set a default output logic, which has no use in input.

Enumeration Type Documentation

16.3 Macro Definition Documentation

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

16.4 Enumeration Type Documentation

16.4.1 enum gpio_pin_direction_t

Enumerator

kGPIO_DigitalInput Set current pin as digital input.

kGPIO_DigitalOutput Set current pin as digital output.

16.5 GPIO Driver

16.5.1 Overview

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

16.5.2 Typical use case

16.5.2.1 Output Operation

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

16.5.2.2 Input Operation

```
/* Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_GPIO_PIN,
    kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};
/* Sets the input pin configuration */
GPIO_PinInit(GPIO_SW1, SW1_PINNUM, &sw1_config);
```

GPIO Configuration

- void [GPIO_PinInit](#) (GPIO_Type *base, uint32_t pin, const [gpio_pin_config_t](#) *config)
Initializes a GPIO pin used by the board.

GPIO Output Operations

- static void [GPIO_WritePinOutput](#) (GPIO_Type *base, uint32_t pin, uint8_t output)
Sets the output level of the multiple GPIO pins to the logic 1 or 0.
- static void [GPIO_SetPinsOutput](#) (GPIO_Type *base, uint32_t mask)
Sets the output level of the multiple GPIO pins to the logic 1.
- static void [GPIO_ClearPinsOutput](#) (GPIO_Type *base, uint32_t mask)
Sets the output level of the multiple GPIO pins to the logic 0.
- static void [GPIO_TogglePinsOutput](#) (GPIO_Type *base, uint32_t mask)
Reverses the current output logic of the multiple GPIO pins.

GPIO Driver

GPIO Input Operations

- static uint32_t [GPIO_ReadPinInput](#) (GPIO_Type *base, uint32_t pin)
Reads the current input value of the GPIO port.

GPIO Interrupt

- uint32_t [GPIO_GetPinsInterruptFlags](#) (GPIO_Type *base)
Reads the GPIO port interrupt status flag.
- void [GPIO_ClearPinsInterruptFlags](#) (GPIO_Type *base, uint32_t mask)
Clears multiple GPIO pin interrupt status flags.

16.5.3 Function Documentation

16.5.3.1 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

16.5.3.2 static void GPIO_WritePinOutput (GPIO_Type * *base*, uint32_t *pin*, uint8_t *output*) [inline], [static]

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>pin</i>	GPIO pin number
<i>output</i>	GPIO pin output logic level. <ul style="list-style-type: none"> • 0: corresponding pin output low-logic level. • 1: corresponding pin output high-logic level.

16.5.3.3 static void GPIO_SetPinsOutput (GPIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

16.5.3.4 static void GPIO_ClearPinsOutput (GPIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

16.5.3.5 static void GPIO_TogglePinsOutput (GPIO_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

16.5.3.6 static uint32_t GPIO_ReadPinInput (GPIO_Type * *base*, uint32_t *pin*) [inline], [static]

GPIO Driver

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>pin</i>	GPIO pin number

Return values

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

16.5.3.7 uint32_t GPIO_GetPinsInterruptFlags (GPIO_Type * *base*)

If a pin is configured to generate the DMA request, the corresponding flag is cleared automatically at the completion of the requested DMA transfer. Otherwise, the flag remains set until a logic one is written to that flag. If configured for a level sensitive interrupt that remains asserted, the flag is set again immediately.

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
-------------	--

Return values

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

16.5.3.8 void GPIO_ClearPinsInterruptFlags (GPIO_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	GPIO peripheral base pointer (GPIOA, GPIOB, GPIOC, and so on.)
<i>mask</i>	GPIO pin number macro

16.6 FGPIO Driver

This chapter describes the programming interface of the FGPIO driver. The FGPIO driver configures the FGPIO module and provides a functional interface to build the GPIO application.

Note

FGPIO (Fast GPIO) is only available in a few MCUs. FGPIO and GPIO share the same peripheral but use different registers. FGPIO is closer to the core than the regular GPIO and it's faster to read and write.

16.6.1 Typical use case

16.6.1.1 Output Operation

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

16.6.1.2 Input Operation

```
/* Input pin configuration */
PORT_SetPinInterruptConfig(BOARD_SW2_PORT, BOARD_SW2_FGPIO_PIN,
    kPORT_InterruptFallingEdge);
NVIC_EnableIRQ(BOARD_SW2_IRQ);
gpio_pin_config_t sw1_config =
{
    kGpioDigitalInput,
    0,
};
/* Sets the input pin configuration */
FGPIO_PinInit(FGPIO_SW1, SW1_PINNUM, &sw1_config);
```




Chapter 17

I2C: Inter-Integrated Circuit Driver

17.1 Overview

Modules

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

17.2 I2C Driver

17.2.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Inter-Integrated Circuit (I2C) module of MCUXpresso SDK devices.

The I2C driver includes functional APIs and transactional APIs.

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

Transactional APIs target the high-level APIs. The transactional APIs can be used to enable the peripheral quickly and also in the application if the code size and performance of transactional APIs satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code using the functional APIs or accessing the hardware registers.

Transactional APIs support asynchronous transfer. This means that the functions [I2C_MasterTransferNonBlocking\(\)](#) set up the interrupt non-blocking transfer. When the transfer completes, the upper layer is notified through a callback function with the status.

17.2.2 Typical use case

17.2.2.1 Master Operation in functional method

```
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];

/* Gets the default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

/* Sends a start and a slave address. */
I2C_MasterStart(EXAMPLE_I2C_MASTER_BASEADDR, 7-bit slave address,
    kI2C_Write/kI2C_Read);

/* Waits for the sent out address. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_MASTER_BASEADDR)) & kI2C_IntPendingFlag))
{
}

if(status & kI2C_ReceiveNakFlag)
{
    return kStatus_I2C_Nak;
}

result = I2C_MasterWriteBlocking(EXAMPLE_I2C_MASTER_BASEADDR, txBuff, BUFFER_SIZE,
    kI2C_TransferDefaultFlag);

if(result)
```

```

{
    return result;
}

```

17.2.2.2 Master Operation in interrupt transactional method

```

i2c_master_handle_t g_m_handle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t status;
status_t result = kStatus_Success;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_handle_t *handle, status_t status, void *
    userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {
        g_MasterCompletionFlag = true;
    }
}

/* Gets a default configuration for master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

I2C_MasterTransferCreateHandle(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle,
    i2c_master_callback, NULL);
I2C_MasterTransferNonBlocking(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_handle, &
    masterXfer);

/* Waits for a transfer to be completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;

```

17.2.2.3 Master Operation in DMA transactional method

```

i2c_master_dma_handle_t g_m_dma_handle;
dma_handle_t dmaHandle;
volatile bool g_MasterCompletionFlag = false;
i2c_master_config_t masterConfig;
uint8_t txBuff[BUFFER_SIZE];
i2c_master_transfer_t masterXfer;

static void i2c_master_callback(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *
    userData)
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)

```

I2C Driver

```
{
    g_MasterCompletionFlag = true;
}

/* Gets the default configuration for the master. */
I2C_MasterGetDefaultConfig(&masterConfig);

/* Initializes the I2C master. */
I2C_MasterInit(EXAMPLE_I2C_MASTER_BASEADDR, &masterConfig, I2C_MASTER_CLK);

masterXfer.slaveAddress = I2C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction = kI2C_Write;
masterXfer.subaddress = NULL;
masterXfer.subaddressSize = 0;
masterXfer.data = txBuff;
masterXfer.dataSize = BUFFER_SIZE;
masterXfer.flags = kI2C_TransferDefaultFlag;

DMAMGR_RequestChannel((dma_request_source_t)DMA_REQUEST_SRC, 0, &dmaHandle);

I2C_MasterTransferCreateHandleDMA(EXAMPLE_I2C_MASTER_BASEADDR, &
    g_m_dma_handle, i2c_master_callback, NULL, &dmaHandle);
I2C_MasterTransferDMA(EXAMPLE_I2C_MASTER_BASEADDR, &g_m_dma_handle, &masterXfer);

/* Wait for transfer completed. */
while (!g_MasterCompletionFlag)
{
}
g_MasterCompletionFlag = false;
```

17.2.2.4 Slave Operation in functional method

```
i2c_slave_config_t slaveConfig;
uint8_t status;
status_t result = kStatus_Success;

I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit
    addressing mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/
    kI2C_RangeMatch;
I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig, I2C_SLAVE_CLK);

/* Waits for an address match. */
while(!((status = I2C_GetStatusFlag(EXAMPLE_I2C_SLAVE_BASEADDR)) & kI2C_AddressMatchFlag))
{
}

/* A slave transmits; master is reading from the slave. */
if (status & kI2C_TransferDirectionFlag)
{
    result = I2C_SlaveWriteBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, txBuff, BUFFER_SIZE);
}
else
{
    I2C_SlaveReadBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, rxBuff, BUFFER_SIZE);
}

return result;
```

17.2.2.5 Slave Operation in interrupt transactional method

```

i2c_slave_config_t slaveConfig;
i2c_slave_handle_t g_s_handle;
volatile bool g_SlaveCompletionFlag = false;

static void i2c_slave_callback(I2C_Type *base, i2c_slave_transfer_t *xfer, void *
    userData)
{
    switch (xfer->event)
    {
        /* Transmit request */
        case kI2C_SlaveTransmitEvent:
            /* Update information for transmit process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Receives request */
        case kI2C_SlaveReceiveEvent:
            /* Update information for received process */
            xfer->data = g_slave_buff;
            xfer->dataSize = I2C_DATA_LENGTH;
            break;

        /* Transfer is done */
        case kI2C_SlaveCompletionEvent:
            g_SlaveCompletionFlag = true;
            break;

        default:
            g_SlaveCompletionFlag = true;
            break;
    }
}

I2C_SlaveGetDefaultConfig(&slaveConfig); /*A default configuration 7-bit
    addressing mode*/
slaveConfig.slaveAddr = 7-bit address
slaveConfig.addressingMode = kI2C_Address7bit/
    kI2C_RangeMatch;

I2C_SlaveInit(EXAMPLE_I2C_SLAVE_BASEADDR, &slaveConfig, I2C_SLAVE_CLK);

I2C_SlaveTransferCreateHandle(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
    i2c_slave_callback, NULL);

I2C_SlaveTransferNonBlocking(EXAMPLE_I2C_SLAVE_BASEADDR, &g_s_handle,
    kI2C_SlaveCompletionEvent);

/* Waits for a transfer to be completed. */
while (!g_SlaveCompletionFlag)
{
}
g_SlaveCompletionFlag = false;

```

Data Structures

- struct [i2c_master_config_t](#)
I2C master user configuration. [More...](#)
- struct [i2c_slave_config_t](#)
I2C slave user configuration. [More...](#)
- struct [i2c_master_transfer_t](#)

I2C Driver

- *I2C master transfer structure. [More...](#)*
• struct [i2c_master_handle_t](#)
- *I2C master handle structure. [More...](#)*
• struct [i2c_slave_transfer_t](#)
- *I2C slave transfer structure. [More...](#)*
• struct [i2c_slave_handle_t](#)
- *I2C slave handle structure. [More...](#)*

Typedefs

- typedef void(* [i2c_master_transfer_callback_t](#))(I2C_Type *base, [i2c_master_handle_t](#) *handle, [status_t](#) status, void *userData)
I2C master transfer callback typedef.
- typedef void(* [i2c_slave_transfer_callback_t](#))(I2C_Type *base, [i2c_slave_transfer_t](#) *xfer, void *userData)
I2C slave transfer callback typedef.

Enumerations

- enum [_i2c_status](#) {
 [kStatus_I2C_Busy](#) = MAKE_STATUS(kStatusGroup_I2C, 0),
 [kStatus_I2C_Idle](#) = MAKE_STATUS(kStatusGroup_I2C, 1),
 [kStatus_I2C_Nak](#) = MAKE_STATUS(kStatusGroup_I2C, 2),
 [kStatus_I2C_ArbitrationLost](#) = MAKE_STATUS(kStatusGroup_I2C, 3),
 [kStatus_I2C_Timeout](#) = MAKE_STATUS(kStatusGroup_I2C, 4),
 [kStatus_I2C_Addr_Nak](#) = MAKE_STATUS(kStatusGroup_I2C, 5) }
I2C status return codes.
- enum [_i2c_flags](#) {
 [kI2C_ReceiveNakFlag](#) = I2C_S_RXAK_MASK,
 [kI2C_IntPendingFlag](#) = I2C_S_IICIF_MASK,
 [kI2C_TransferDirectionFlag](#) = I2C_S_SRW_MASK,
 [kI2C_RangeAddressMatchFlag](#) = I2C_S_RAM_MASK,
 [kI2C_ArbitrationLostFlag](#) = I2C_S_ARBL_MASK,
 [kI2C_BusBusyFlag](#) = I2C_S_BUSY_MASK,
 [kI2C_AddressMatchFlag](#) = I2C_S_IAAS_MASK,
 [kI2C_TransferCompleteFlag](#) = I2C_S_TCF_MASK }
I2C peripheral flags.
- enum [_i2c_interrupt_enable](#) { [kI2C_GlobalInterruptEnable](#) = I2C_C1_IICIE_MASK }
I2C feature interrupt source.
- enum [i2c_direction_t](#) {
 [kI2C_Write](#) = 0x0U,
 [kI2C_Read](#) = 0x1U }
The direction of master and slave transfers.
- enum [i2c_slave_address_mode_t](#) {
 [kI2C_Address7bit](#) = 0x0U,
 [kI2C_RangeMatch](#) = 0x2U }

Addressing mode.

- enum `_i2c_master_transfer_flags` {
`kI2C_TransferDefaultFlag` = 0x0U,
`kI2C_TransferNoStartFlag` = 0x1U,
`kI2C_TransferRepeatedStartFlag` = 0x2U,
`kI2C_TransferNoStopFlag` = 0x4U }

I2C transfer control flag.

- enum `i2c_slave_transfer_event_t` {
`kI2C_SlaveAddressMatchEvent` = 0x01U,
`kI2C_SlaveTransmitEvent` = 0x02U,
`kI2C_SlaveReceiveEvent` = 0x04U,
`kI2C_SlaveTransmitAckEvent` = 0x08U,
`kI2C_SlaveCompletionEvent` = 0x20U,
`kI2C_SlaveGeneralCallEvent` = 0x40U,
`kI2C_SlaveAllEvents` }

Set of events sent to the callback for nonblocking slave transfers.

Driver version

- #define `FSL_I2C_DRIVER_VERSION` (`MAKE_VERSION`(2, 0, 3))
I2C driver version 2.0.3.

Initialization and deinitialization

- void `I2C_MasterInit` (`I2C_Type` *base, const `i2c_master_config_t` *masterConfig, uint32_t srcClock_Hz)
Initializes the I2C peripheral.
- void `I2C_SlaveInit` (`I2C_Type` *base, const `i2c_slave_config_t` *slaveConfig, uint32_t srcClock_Hz)
Initializes the I2C peripheral.
- void `I2C_MasterDeinit` (`I2C_Type` *base)
De-initializes the I2C master peripheral.
- void `I2C_SlaveDeinit` (`I2C_Type` *base)
De-initializes the I2C slave peripheral.
- void `I2C_MasterGetDefaultConfig` (`i2c_master_config_t` *masterConfig)
Sets the I2C master configuration structure to default values.
- void `I2C_SlaveGetDefaultConfig` (`i2c_slave_config_t` *slaveConfig)
Sets the I2C slave configuration structure to default values.
- static void `I2C_Enable` (`I2C_Type` *base, bool enable)
Enables or disables the I2C peripheral operation.

Status

- uint32_t `I2C_MasterGetStatusFlags` (`I2C_Type` *base)
Gets the I2C status flags.

I2C Driver

- static uint32_t [I2C_SlaveGetStatusFlags](#) (I2C_Type *base)
Gets the I2C status flags.
- static void [I2C_MasterClearStatusFlags](#) (I2C_Type *base, uint32_t statusMask)
Clears the I2C status flag state.
- static void [I2C_SlaveClearStatusFlags](#) (I2C_Type *base, uint32_t statusMask)
Clears the I2C status flag state.

Interrupts

- void [I2C_EnableInterrupts](#) (I2C_Type *base, uint32_t mask)
Enables I2C interrupt requests.
- void [I2C_DisableInterrupts](#) (I2C_Type *base, uint32_t mask)
Disables I2C interrupt requests.

DMA Control

- static uint32_t [I2C_GetDataRegAddr](#) (I2C_Type *base)
Gets the I2C tx/rx data register address.

Bus Operations

- void [I2C_MasterSetBaudRate](#) (I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
Sets the I2C master transfer baud rate.
- status_t [I2C_MasterStart](#) (I2C_Type *base, uint8_t address, [i2c_direction_t](#) direction)
Sends a START on the I2C bus.
- status_t [I2C_MasterStop](#) (I2C_Type *base)
Sends a STOP signal on the I2C bus.
- status_t [I2C_MasterRepeatedStart](#) (I2C_Type *base, uint8_t address, [i2c_direction_t](#) direction)
Sends a REPEATED START on the I2C bus.
- status_t [I2C_MasterWriteBlocking](#) (I2C_Type *base, const uint8_t *txBuff, size_t txSize, uint32_t flags)
Performs a polling send transaction on the I2C bus.
- status_t [I2C_MasterReadBlocking](#) (I2C_Type *base, uint8_t *rxBuff, size_t rxSize, uint32_t flags)
Performs a polling receive transaction on the I2C bus.
- status_t [I2C_SlaveWriteBlocking](#) (I2C_Type *base, const uint8_t *txBuff, size_t txSize)
Performs a polling send transaction on the I2C bus.
- void [I2C_SlaveReadBlocking](#) (I2C_Type *base, uint8_t *rxBuff, size_t rxSize)
Performs a polling receive transaction on the I2C bus.
- status_t [I2C_MasterTransferBlocking](#) (I2C_Type *base, [i2c_master_transfer_t](#) *xfer)
Performs a master polling transfer on the I2C bus.

Transactional

- void [I2C_MasterTransferCreateHandle](#) (I2C_Type *base, [i2c_master_handle_t](#) *handle, [i2c_master_transfer_callback_t](#) callback, void *userData)

- Initializes the I2C handle which is used in transactional functions.*
- status_t [I2C_MasterTransferNonBlocking](#) (I2C_Type *base, i2c_master_handle_t *handle, [i2c_master_transfer_t](#) *xfer)
- Performs a master interrupt non-blocking transfer on the I2C bus.*
- status_t [I2C_MasterTransferGetCount](#) (I2C_Type *base, i2c_master_handle_t *handle, size_t *count)
- Gets the master transfer status during a interrupt non-blocking transfer.*
- void [I2C_MasterTransferAbort](#) (I2C_Type *base, i2c_master_handle_t *handle)
- Aborts an interrupt non-blocking transfer early.*
- void [I2C_MasterTransferHandleIRQ](#) (I2C_Type *base, void *i2cHandle)
- Master interrupt handler.*
- void [I2C_SlaveTransferCreateHandle](#) (I2C_Type *base, i2c_slave_handle_t *handle, [i2c_slave_transfer_callback_t](#) callback, void *userData)
- Initializes the I2C handle which is used in transactional functions.*
- status_t [I2C_SlaveTransferNonBlocking](#) (I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)
- Starts accepting slave transfers.*
- void [I2C_SlaveTransferAbort](#) (I2C_Type *base, i2c_slave_handle_t *handle)
- Aborts the slave transfer.*
- status_t [I2C_SlaveTransferGetCount](#) (I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)
- Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.*
- void [I2C_SlaveTransferHandleIRQ](#) (I2C_Type *base, void *i2cHandle)
- Slave interrupt handler.*

17.2.3 Data Structure Documentation

17.2.3.1 struct i2c_master_config_t

Data Fields

- bool [enableMaster](#)
Enables the I2C peripheral at initialization time.
- uint32_t [baudRate_Bps](#)
Baud rate configuration of I2C peripheral.
- uint8_t [glitchFilterWidth](#)
Controls the width of the glitch.

I2C Driver

17.2.3.1.0.45 Field Documentation

17.2.3.1.0.45.1 `bool i2c_master_config_t::enableMaster`

17.2.3.1.0.45.2 `uint32_t i2c_master_config_t::baudRate_Bps`

17.2.3.1.0.45.3 `uint8_t i2c_master_config_t::glitchFilterWidth`

17.2.3.2 `struct i2c_slave_config_t`

Data Fields

- `bool enableSlave`
Enables the I2C peripheral at initialization time.
- `bool enableGeneralCall`
Enables the general call addressing mode.
- `bool enableWakeUp`
Enables/disables waking up MCU from low-power mode.
- `bool enableBaudRateCtl`
Enables/disables independent slave baud rate on SCL in very fast I2C modes.
- `uint16_t slaveAddress`
A slave address configuration.
- `uint16_t upperAddress`
A maximum boundary slave address used in a range matching mode.
- `i2c_slave_address_mode_t addressingMode`
An addressing mode configuration of `i2c_slave_address_mode_config_t`.
- `uint32_t sclStopHoldTime_ns`
the delay from the rising edge of SCL (I2C clock) to the rising edge of SDA (I2C data) while SCL is high (stop condition), SDA hold time and SCL start hold time are also configured according to the SCL stop hold time.

17.2.3.2.0.46 Field Documentation**17.2.3.2.0.46.1** `bool i2c_slave_config_t::enableSlave`**17.2.3.2.0.46.2** `bool i2c_slave_config_t::enableGeneralCall`**17.2.3.2.0.46.3** `bool i2c_slave_config_t::enableWakeUp`**17.2.3.2.0.46.4** `bool i2c_slave_config_t::enableBaudRateCtl`**17.2.3.2.0.46.5** `uint16_t i2c_slave_config_t::slaveAddress`**17.2.3.2.0.46.6** `uint16_t i2c_slave_config_t::upperAddress`**17.2.3.2.0.46.7** `i2c_slave_address_mode_t i2c_slave_config_t::addressingMode`**17.2.3.2.0.46.8** `uint32_t i2c_slave_config_t::sclStopHoldTime_ns`**17.2.3.3 struct i2c_master_transfer_t****Data Fields**

- `uint32_t flags`
A transfer flag which controls the transfer.
- `uint8_t slaveAddress`
7-bit slave address.
- `i2c_direction_t direction`
A transfer direction, read or write.
- `uint32_t subaddress`
A sub address.
- `uint8_t subaddressSize`
A size of the command buffer.
- `uint8_t *volatile data`
A transfer buffer.
- `volatile size_t dataSize`
A transfer size.

17.2.3.3.0.47 Field Documentation**17.2.3.3.0.47.1** `uint32_t i2c_master_transfer_t::flags`**17.2.3.3.0.47.2** `uint8_t i2c_master_transfer_t::slaveAddress`**17.2.3.3.0.47.3** `i2c_direction_t i2c_master_transfer_t::direction`**17.2.3.3.0.47.4** `uint32_t i2c_master_transfer_t::subaddress`

Transferred MSB first.

I2C Driver

17.2.3.3.0.47.5 `uint8_t i2c_master_transfer_t::subaddressSize`

17.2.3.3.0.47.6 `uint8_t* volatile i2c_master_transfer_t::data`

17.2.3.3.0.47.7 `volatile size_t i2c_master_transfer_t::dataSize`

17.2.3.4 `struct i2c_master_handle`

I2C master handle typedef.

Data Fields

- [i2c_master_transfer_t transfer](#)
I2C master transfer copy.
- `size_t` [transferSize](#)
Total bytes to be transferred.
- `uint8_t` [state](#)
A transfer state maintained during transfer.
- [i2c_master_transfer_callback_t completionCallback](#)
A callback function called when the transfer is finished.
- `void *` [userData](#)
A callback parameter passed to the callback function.

17.2.3.4.0.48 Field Documentation

17.2.3.4.0.48.1 `i2c_master_transfer_t i2c_master_handle_t::transfer`

17.2.3.4.0.48.2 `size_t i2c_master_handle_t::transferSize`

17.2.3.4.0.48.3 `uint8_t i2c_master_handle_t::state`

17.2.3.4.0.48.4 `i2c_master_transfer_callback_t i2c_master_handle_t::completionCallback`

17.2.3.4.0.48.5 `void* i2c_master_handle_t::userData`

17.2.3.5 `struct i2c_slave_transfer_t`

Data Fields

- [i2c_slave_transfer_event_t event](#)
A reason that the callback is invoked.
- `uint8_t *volatile` [data](#)
A transfer buffer.
- `volatile size_t` [dataSize](#)
A transfer size.
- `status_t` [completionStatus](#)
Success or error code describing how the transfer completed.
- `size_t` [transferredCount](#)
A number of bytes actually transferred since the start or since the last repeated start.

17.2.3.5.0.49 Field Documentation**17.2.3.5.0.49.1** `i2c_slave_transfer_event_t i2c_slave_transfer_t::event`**17.2.3.5.0.49.2** `uint8_t* volatile i2c_slave_transfer_t::data`**17.2.3.5.0.49.3** `volatile size_t i2c_slave_transfer_t::dataSize`**17.2.3.5.0.49.4** `status_t i2c_slave_transfer_t::completionStatus`

Only applies for [kI2C_SlaveCompletionEvent](#).

17.2.3.5.0.49.5 `size_t i2c_slave_transfer_t::transferredCount`**17.2.3.6 struct _i2c_slave_handle**

I2C slave handle typedef.

Data Fields

- volatile bool [isBusy](#)
Indicates whether a transfer is busy.
- [i2c_slave_transfer_t](#) `transfer`
I2C slave transfer copy.
- uint32_t [eventMask](#)
A mask of enabled events.
- [i2c_slave_transfer_callback_t](#) `callback`
A callback function called at the transfer event.
- void * [userData](#)
A callback parameter passed to the callback.

I2C Driver

17.2.3.6.0.50 Field Documentation

17.2.3.6.0.50.1 volatile bool i2c_slave_handle_t::isBusy

17.2.3.6.0.50.2 i2c_slave_transfer_t i2c_slave_handle_t::transfer

17.2.3.6.0.50.3 uint32_t i2c_slave_handle_t::eventMask

17.2.3.6.0.50.4 i2c_slave_transfer_callback_t i2c_slave_handle_t::callback

17.2.3.6.0.50.5 void* i2c_slave_handle_t::userData

17.2.4 Macro Definition Documentation

17.2.4.1 #define FSL_I2C_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

17.2.5 Typedef Documentation

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

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

17.2.6 Enumeration Type Documentation

17.2.6.1 enum _i2c_status

Enumerator

kStatus_I2C_Busy I2C is busy with current transfer.

kStatus_I2C_Idle Bus is Idle.

kStatus_I2C_Nak NAK received during transfer.

kStatus_I2C_ArbitrationLost Arbitration lost during transfer.

kStatus_I2C_Timeout Wait event timeout.

kStatus_I2C_Addr_Nak NAK received during the address probe.

17.2.6.2 enum _i2c_flags

The following status register flags can be cleared:

- [kI2C_ArbitrationLostFlag](#)
- [kI2C_IntPendingFlag](#)
- #kI2C_StartDetectFlag
- #kI2C_StopDetectFlag

Note

These enumerations are meant to be OR'd together to form a bit mask.

Enumerator

kI2C_ReceiveNakFlag I2C receive NAK flag.
kI2C_IntPendingFlag I2C interrupt pending flag.
kI2C_TransferDirectionFlag I2C transfer direction flag.
kI2C_RangeAddressMatchFlag I2C range address match flag.
kI2C_ArbitrationLostFlag I2C arbitration lost flag.
kI2C_BusBusyFlag I2C bus busy flag.
kI2C_AddressMatchFlag I2C address match flag.
kI2C_TransferCompleteFlag I2C transfer complete flag.

17.2.6.3 enum _i2c_interrupt_enable

Enumerator

kI2C_GlobalInterruptEnable I2C global interrupt.

17.2.6.4 enum i2c_direction_t

Enumerator

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

17.2.6.5 enum i2c_slave_address_mode_t

Enumerator

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

17.2.6.6 enum _i2c_master_transfer_flags

Enumerator

kI2C_TransferDefaultFlag A transfer starts with a start signal, stops with a stop signal.
kI2C_TransferNoStartFlag A transfer starts without a start signal.
kI2C_TransferRepeatedStartFlag A transfer starts with a repeated start signal.
kI2C_TransferNoStopFlag A transfer ends without a stop signal.

I2C Driver

17.2.6.7 enum i2c_slave_transfer_event_t

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to [I2C_SlaveTransferNonBlocking\(\)](#) to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note

These enumerations are meant to be OR'd together to form a bit mask of events.

Enumerator

kI2C_SlaveAddressMatchEvent Received the slave address after a start or repeated start.

kI2C_SlaveTransmitEvent A callback is requested to provide data to transmit (slave-transmitter role).

kI2C_SlaveReceiveEvent A callback is requested to provide a buffer in which to place received data (slave-receiver role).

kI2C_SlaveTransmitAckEvent A callback needs to either transmit an ACK or NACK.

kI2C_SlaveCompletionEvent A stop was detected or finished transfer, completing the transfer.

kI2C_SlaveGeneralCallEvent Received the general call address after a start or repeated start.

kI2C_SlaveAllEvents A bit mask of all available events.

17.2.7 Function Documentation

17.2.7.1 void I2C_MasterInit (I2C_Type * *base*, const i2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

Call this API to ungate the I2C clock and configure the I2C with master configuration.

Note

This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can be custom filled or it can be set with default values by using the [I2C_MasterGetDefaultConfig\(\)](#). After calling this API, the master is ready to transfer. This is an example.

```
* i2c_master_config_t config = {  
*   .enableMaster = true,  
*   .enableStopHold = false,  
*   .highDrive = false,  
*   .baudRate_Bps = 100000,  
*   .glitchFilterWidth = 0  
* };  
* I2C_MasterInit(I2C0, &config, 12000000U);  
*
```

Parameters

<i>base</i>	I2C base pointer
<i>masterConfig</i>	A pointer to the master configuration structure
<i>srcClock_Hz</i>	I2C peripheral clock frequency in Hz

17.2.7.2 void I2C_SlaveInit (I2C_Type * *base*, const i2c_slave_config_t * *slaveConfig*, uint32_t *srcClock_Hz*)

Call this API to ungate the I2C clock and initialize the I2C with the slave configuration.

Note

This API should be called at the beginning of the application. Otherwise, any operation to the I2C module can cause a hard fault because the clock is not enabled. The configuration structure can partly be set with default values by [I2C_SlaveGetDefaultConfig\(\)](#) or it can be custom filled by the user. This is an example.

```
* i2c_slave_config_t config = {
* .enableSlave = true,
* .enableGeneralCall = false,
* .addressingMode = kI2C_Address7bit,
* .slaveAddress = 0x1DU,
* .enableWakeUp = false,
* .enablehighDrive = false,
* .enableBaudRateCtl = false,
* .sclStopHoldTime_ns = 4000
* };
* I2C_SlaveInit(I2C0, &config, 12000000U);
*
```

Parameters

<i>base</i>	I2C base pointer
<i>slaveConfig</i>	A pointer to the slave configuration structure
<i>srcClock_Hz</i>	I2C peripheral clock frequency in Hz

17.2.7.3 void I2C_MasterDeinit (I2C_Type * *base*)

Call this API to gate the I2C clock. The I2C master module can't work unless the I2C_MasterInit is called.

I2C Driver

Parameters

<i>base</i>	I2C base pointer
-------------	------------------

17.2.7.4 void I2C_SlaveDeinit (I2C_Type * *base*)

Calling this API gates the I2C clock. The I2C slave module can't work unless the I2C_SlaveInit is called to enable the clock.

Parameters

<i>base</i>	I2C base pointer
-------------	------------------

17.2.7.5 void I2C_MasterGetDefaultConfig (i2c_master_config_t * *masterConfig*)

The purpose of this API is to get the configuration structure initialized for use in the I2C_MasterConfigure(). Use the initialized structure unchanged in the I2C_MasterConfigure() or modify the structure before calling the I2C_MasterConfigure(). This is an example.

```
* i2c_master_config_t config;  
* I2C_MasterGetDefaultConfig(&config);  
*
```

Parameters

<i>masterConfig</i>	A pointer to the master configuration structure.
---------------------	--

17.2.7.6 void I2C_SlaveGetDefaultConfig (i2c_slave_config_t * *slaveConfig*)

The purpose of this API is to get the configuration structure initialized for use in the I2C_SlaveConfigure(). Modify fields of the structure before calling the I2C_SlaveConfigure(). This is an example.

```
* i2c_slave_config_t config;  
* I2C_SlaveGetDefaultConfig(&config);  
*
```

Parameters

<i>slaveConfig</i>	A pointer to the slave configuration structure.
--------------------	---

17.2.7.7 static void I2C_Enable (I2C_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	I2C base pointer
<i>enable</i>	Pass true to enable and false to disable the module.

17.2.7.8 uint32_t I2C_MasterGetStatusFlags (I2C_Type * *base*)

Parameters

<i>base</i>	I2C base pointer
-------------	------------------

Returns

status flag, use status flag to AND [_i2c_flags](#) to get the related status.

17.2.7.9 static uint32_t I2C_SlaveGetStatusFlags (I2C_Type * *base*) [inline], [static]

Parameters

<i>base</i>	I2C base pointer
-------------	------------------

Returns

status flag, use status flag to AND [_i2c_flags](#) to get the related status.

17.2.7.10 static void I2C_MasterClearStatusFlags (I2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared kI2C_ArbitrationLostFlag and kI2C_IntPendingFlag.

I2C Driver

Parameters

<i>base</i>	I2C base pointer
<i>statusMask</i>	The status flag mask, defined in type <code>i2c_status_flag_t</code> . The parameter can be any combination of the following values: <ul style="list-style-type: none">• <code>kI2C_StartDetectFlag</code> (if available)• <code>kI2C_StopDetectFlag</code> (if available)• <code>kI2C_ArbitrationLostFlag</code>• <code>kI2C_IntPendingFlagFlag</code>

17.2.7.11 static void I2C_SlaveClearStatusFlags (I2C_Type * *base*, uint32_t *statusMask*) [inline], [static]

The following status register flags can be cleared `kI2C_ArbitrationLostFlag` and `kI2C_IntPendingFlag`

Parameters

<i>base</i>	I2C base pointer
<i>statusMask</i>	The status flag mask, defined in type <code>i2c_status_flag_t</code> . The parameter can be any combination of the following values: <ul style="list-style-type: none">• <code>kI2C_StartDetectFlag</code> (if available)• <code>kI2C_StopDetectFlag</code> (if available)• <code>kI2C_ArbitrationLostFlag</code>• <code>kI2C_IntPendingFlagFlag</code>

17.2.7.12 void I2C_EnableInterrupts (I2C_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	I2C base pointer
<i>mask</i>	interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none">• <code>kI2C_GlobalInterruptEnable</code>• <code>kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable</code>• <code>kI2C_SdaTimeoutInterruptEnable</code>

17.2.7.13 void I2C_DisableInterrupts (I2C_Type * *base*, uint32_t *mask*)

Parameters

<i>base</i>	I2C base pointer
<i>mask</i>	interrupt source The parameter can be combination of the following source if defined: <ul style="list-style-type: none"> • kI2C_GlobalInterruptEnable • kI2C_StopDetectInterruptEnable/kI2C_StartDetectInterruptEnable • kI2C_SdaTimeoutInterruptEnable

17.2.7.14 static uint32_t I2C_GetDataRegAddr (I2C_Type * *base*) [inline], [static]

This API is used to provide a transfer address for I2C DMA transfer configuration.

Parameters

<i>base</i>	I2C base pointer
-------------	------------------

Returns

data register address

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

Parameters

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

17.2.7.16 status_t I2C_MasterStart (I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*)

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

I2C Driver

Parameters

<i>base</i>	I2C peripheral base pointer
<i>address</i>	7-bit slave device address.
<i>direction</i>	Master transfer directions(transmit/receive).

Return values

<i>kStatus_Success</i>	Successfully send the start signal.
<i>kStatus_I2C_Busy</i>	Current bus is busy.

17.2.7.17 **status_t I2C_MasterStop (I2C_Type * *base*)**

Return values

<i>kStatus_Success</i>	Successfully send the stop signal.
<i>kStatus_I2C_Timeout</i>	Send stop signal failed, timeout.

17.2.7.18 **status_t I2C_MasterRepeatedStart (I2C_Type * *base*, uint8_t *address*, i2c_direction_t *direction*)**

Parameters

<i>base</i>	I2C peripheral base pointer
<i>address</i>	7-bit slave device address.
<i>direction</i>	Master transfer directions(transmit/receive).

Return values

<i>kStatus_Success</i>	Successfully send the start signal.
<i>kStatus_I2C_Busy</i>	Current bus is busy but not occupied by current I2C master.

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

Parameters

<i>base</i>	The I2C peripheral base pointer.
<i>txBuff</i>	The pointer to the data to be transferred.
<i>txSize</i>	The length in bytes of the data to be transferred.
<i>flags</i>	Transfer control flag to decide whether need to send a stop, use kI2C_Transfer-DefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

<i>kStatus_Success</i>	Successfully complete the data transmission.
<i>kStatus_I2C_Arbitration-Lost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

17.2.7.20 status_t I2C_MasterReadBlocking (I2C_Type * *base*, uint8_t * *rxBuff*, size_t *rxSize*, uint32_t *flags*)

Note

The I2C_MasterReadBlocking function stops the bus before reading the final byte. Without stopping the bus prior for the final read, the bus issues another read, resulting in garbage data being read into the data register.

Parameters

<i>base</i>	I2C peripheral base pointer.
<i>rxBuff</i>	The pointer to the data to store the received data.
<i>rxSize</i>	The length in bytes of the data to be received.
<i>flags</i>	Transfer control flag to decide whether need to send a stop, use kI2C_Transfer-DefaultFlag to issue a stop and kI2C_TransferNoStop to not send a stop.

Return values

<i>kStatus_Success</i>	Successfully complete the data transmission.
------------------------	--

I2C Driver

<i>kStatus_I2C_Timeout</i>	Send stop signal failed, timeout.
----------------------------	-----------------------------------

17.2.7.21 **status_t I2C_SlaveWriteBlocking (I2C_Type * *base*, const uint8_t * *txBuff*, size_t *txSize*)**

Parameters

<i>base</i>	The I2C peripheral base pointer.
<i>txBuff</i>	The pointer to the data to be transferred.
<i>txSize</i>	The length in bytes of the data to be transferred.

Return values

<i>kStatus_Success</i>	Successfully complete the data transmission.
<i>kStatus_I2C_Arbitration-Lost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

17.2.7.22 **void I2C_SlaveReadBlocking (I2C_Type * *base*, uint8_t * *rxBuff*, size_t *rxSize*)**

Parameters

<i>base</i>	I2C peripheral base pointer.
<i>rxBuff</i>	The pointer to the data to store the received data.
<i>rxSize</i>	The length in bytes of the data to be received.

17.2.7.23 **status_t I2C_MasterTransferBlocking (I2C_Type * *base*, i2c_master_transfer_t * *xfer*)**

Note

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

Parameters

<i>base</i>	I2C peripheral base address.
<i>xfer</i>	Pointer to the transfer structure.

Return values

<i>kStatus_Success</i>	Successfully complete the data transmission.
<i>kStatus_I2C_Busy</i>	Previous transmission still not finished.
<i>kStatus_I2C_Timeout</i>	Transfer error, wait signal timeout.
<i>kStatus_I2C_Arbitration-Lost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

17.2.7.24 void I2C_MasterTransferCreateHandle (I2C_Type * *base*, i2c_master_handle_t * *handle*, i2c_master_transfer_callback_t *callback*, void * *userData*)

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_master_handle_t structure to store the transfer state.
<i>callback</i>	pointer to user callback function.
<i>userData</i>	user parameter passed to the callback function.

17.2.7.25 status_t I2C_MasterTransferNonBlocking (I2C_Type * *base*, i2c_master_handle_t * *handle*, i2c_master_transfer_t * *xfer*)

Note

Calling the API returns immediately after transfer initiates. The user needs to call I2C_MasterGetTransferCount to poll the transfer status to check whether the transfer is finished. If the return status is not kStatus_I2C_Busy, the transfer is finished.

Parameters

I2C Driver

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to <code>i2c_master_handle_t</code> structure which stores the transfer state.
<i>xfer</i>	pointer to i2c_master_transfer_t structure.

Return values

<i>kStatus_Success</i>	Successfully start the data transmission.
<i>kStatus_I2C_Busy</i>	Previous transmission still not finished.
<i>kStatus_I2C_Timeout</i>	Transfer error, wait signal timeout.

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

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to <code>i2c_master_handle_t</code> structure which stores the transfer state.
<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

Return values

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

17.2.7.27 `void I2C_MasterTransferAbort (I2C_Type * base, i2c_master_handle_t * handle)`

Note

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

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_master_handle_t structure which stores the transfer state

17.2.7.28 void I2C_MasterTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)

Parameters

<i>base</i>	I2C base pointer.
<i>i2cHandle</i>	pointer to i2c_master_handle_t structure.

17.2.7.29 void I2C_SlaveTransferCreateHandle (I2C_Type * *base*, i2c_slave_handle_t * *handle*, i2c_slave_transfer_callback_t *callback*, void * *userData*)

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_slave_handle_t structure to store the transfer state.
<i>callback</i>	pointer to user callback function.
<i>userData</i>	user parameter passed to the callback function.

17.2.7.30 status_t I2C_SlaveTransferNonBlocking (I2C_Type * *base*, i2c_slave_handle_t * *handle*, uint32_t *eventMask*)

Call this API after calling the [I2C_SlaveInit\(\)](#) and [I2C_SlaveTransferCreateHandle\(\)](#) to start processing transactions driven by an I2C master. The slave monitors the I2C bus and passes events to the callback that was passed into the call to [I2C_SlaveTransferCreateHandle\(\)](#). The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of [i2c_slave_transfer_event_t](#) enumerators for the events you wish to receive. The [kI2C_SlaveTransmitEvent](#) and [#kLPI2C_SlaveReceiveEvent](#) events are always enabled and do not need to be included in the mask. Alternatively, pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the [kI2C_SlaveAllEvents](#) constant is provided as a convenient way to enable all events.

I2C Driver

Parameters

<i>base</i>	The I2C peripheral base address.
<i>handle</i>	Pointer to #i2c_slave_handle_t structure which stores the transfer state.
<i>eventMask</i>	Bit mask formed by OR'ing together i2c_slave_transfer_event_t enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and kI2C_SlaveAllEvents to enable all events.

Return values

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

17.2.7.31 void I2C_SlaveTransferAbort (I2C_Type * *base*, i2c_slave_handle_t * *handle*)

Note

This API can be called at any time to stop slave for handling the bus events.

Parameters

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

17.2.7.32 status_t I2C_SlaveTransferGetCount (I2C_Type * *base*, i2c_slave_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	I2C base pointer.
<i>handle</i>	pointer to i2c_slave_handle_t structure.
<i>count</i>	Number of bytes transferred so far by the non-blocking transaction.

Return values

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

17.2.7.33 void I2C_SlaveTransferHandleIRQ (I2C_Type * *base*, void * *i2cHandle*)

Parameters

<i>base</i>	I2C base pointer.
<i>i2cHandle</i>	pointer to i2c_slave_handle_t structure which stores the transfer state

I2C eDMA Driver

17.3 I2C eDMA Driver

17.3.1 Overview

Data Structures

- struct [i2c_master_edma_handle_t](#)
I2C master eDMA transfer structure. [More...](#)

Typedefs

- typedef void(* [i2c_master_edma_transfer_callback_t](#))(I2C_Type *base, i2c_master_edma_handle_t *handle, status_t status, void *userData)
I2C master eDMA transfer callback typedef.

I2C Block eDMA Transfer Operation

- void [I2C_MasterCreateEDMAHandle](#) (I2C_Type *base, i2c_master_edma_handle_t *handle, [i2c_master_edma_transfer_callback_t](#) callback, void *userData, [edma_handle_t](#) *edmaHandle)
Initializes the I2C handle which is used in transactional functions.
- status_t [I2C_MasterTransferEDMA](#) (I2C_Type *base, i2c_master_edma_handle_t *handle, [i2c_master_transfer_t](#) *xfer)
Performs a master eDMA non-blocking transfer on the I2C bus.
- status_t [I2C_MasterTransferGetCountEDMA](#) (I2C_Type *base, i2c_master_edma_handle_t *handle, size_t *count)
Gets a master transfer status during the eDMA non-blocking transfer.
- void [I2C_MasterTransferAbortEDMA](#) (I2C_Type *base, i2c_master_edma_handle_t *handle)
Aborts a master eDMA non-blocking transfer early.

17.3.2 Data Structure Documentation

17.3.2.1 struct [i2c_master_edma_handle](#)

I2C master eDMA handle typedef.

Data Fields

- [i2c_master_transfer_t](#) transfer
I2C master transfer structure.
- size_t [transferSize](#)
Total bytes to be transferred.
- uint8_t [nbytes](#)
eDMA minor byte transfer count initially configured.
- uint8_t [state](#)

- I2C master transfer status.*
- `edma_handle_t * dmaHandle`
The eDMA handler used.
- `i2c_master_edma_transfer_callback_t completionCallback`
A callback function called after the eDMA transfer is finished.
- `void * userData`
A callback parameter passed to the callback function.

17.3.2.1.0.51 Field Documentation

17.3.2.1.0.51.1 `i2c_master_transfer_t i2c_master_edma_handle_t::transfer`

17.3.2.1.0.51.2 `size_t i2c_master_edma_handle_t::transferSize`

17.3.2.1.0.51.3 `uint8_t i2c_master_edma_handle_t::nbytes`

17.3.2.1.0.51.4 `uint8_t i2c_master_edma_handle_t::state`

17.3.2.1.0.51.5 `edma_handle_t* i2c_master_edma_handle_t::dmaHandle`

17.3.2.1.0.51.6 `i2c_master_edma_transfer_callback_t i2c_master_edma_handle_t::completion-Callback`

17.3.2.1.0.51.7 `void* i2c_master_edma_handle_t::userData`

17.3.3 Typedef Documentation

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

17.3.4 Function Documentation

17.3.4.1 `void I2C_MasterCreateEDMAHandle (I2C_Type * base, i2c_master_edma_handle_t * handle, i2c_master_edma_transfer_callback_t callback, void * userData, edma_handle_t * edmaHandle)`

Parameters

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

I2C eDMA Driver

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

17.3.4.2 `status_t I2C_MasterTransferEDMA (I2C_Type * base, i2c_master_edma_handle_t * handle, i2c_master_transfer_t * xfer)`

Parameters

<i>base</i>	I2C peripheral base address.
<i>handle</i>	A pointer to the <code>i2c_master_edma_handle_t</code> structure.
<i>xfer</i>	A pointer to the transfer structure of i2c_master_transfer_t .

Return values

<i>kStatus_Success</i>	Successfully completed the data transmission.
<i>kStatus_I2C_Busy</i>	A previous transmission is still not finished.
<i>kStatus_I2C_Timeout</i>	Transfer error, waits for a signal timeout.
<i>kStatus_I2C_Arbitration-Lost</i>	Transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	Transfer error, receive NAK during transfer.

17.3.4.3 `status_t I2C_MasterTransferGetCountEDMA (I2C_Type * base, i2c_master_edma_handle_t * handle, size_t * count)`

Parameters

<i>base</i>	I2C peripheral base address.
<i>handle</i>	A pointer to the <code>i2c_master_edma_handle_t</code> structure.
<i>count</i>	A number of bytes transferred by the non-blocking transaction.

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

Parameters

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

I2C DMA Driver

17.4 I2C DMA Driver

17.4.1 Overview

Data Structures

- struct [i2c_master_dma_handle_t](#)
I2C master DMA transfer structure. [More...](#)

Typedefs

- typedef void(* [i2c_master_dma_transfer_callback_t](#))(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)
I2C master DMA transfer callback typedef.

I2C Block DMA Transfer Operation

- void [I2C_MasterTransferCreateHandleDMA](#) (I2C_Type *base, i2c_master_dma_handle_t *handle, [i2c_master_dma_transfer_callback_t](#) callback, void *userData, dma_handle_t *dmaHandle)
Initializes the I2C handle which is used in transactional functions.
- status_t [I2C_MasterTransferDMA](#) (I2C_Type *base, i2c_master_dma_handle_t *handle, [i2c_master_transfer_t](#) *xfer)
Performs a master DMA non-blocking transfer on the I2C bus.
- status_t [I2C_MasterTransferGetCountDMA](#) (I2C_Type *base, i2c_master_dma_handle_t *handle, size_t *count)
Gets a master transfer status during a DMA non-blocking transfer.
- void [I2C_MasterTransferAbortDMA](#) (I2C_Type *base, i2c_master_dma_handle_t *handle)
Aborts a master DMA non-blocking transfer early.

17.4.2 Data Structure Documentation

17.4.2.1 struct [i2c_master_dma_handle](#)

I2C master DMA handle typedef.

Data Fields

- [i2c_master_transfer_t](#) transfer
I2C master transfer struct.
- size_t [transferSize](#)
Total bytes to be transferred.
- uint8_t [state](#)
I2C master transfer status.
- dma_handle_t * [dmaHandle](#)

The DMA handler used.

- [i2c_master_dma_transfer_callback_t completionCallback](#)
A callback function called after the DMA transfer finished.
- void * [userData](#)
A callback parameter passed to the callback function.

17.4.2.1.0.52 Field Documentation

17.4.2.1.0.52.1 `i2c_master_transfer_t i2c_master_dma_handle_t::transfer`

17.4.2.1.0.52.2 `size_t i2c_master_dma_handle_t::transferSize`

17.4.2.1.0.52.3 `uint8_t i2c_master_dma_handle_t::state`

17.4.2.1.0.52.4 `dma_handle_t* i2c_master_dma_handle_t::dmaHandle`

17.4.2.1.0.52.5 `i2c_master_dma_transfer_callback_t i2c_master_dma_handle_t::completion-Callback`

17.4.2.1.0.52.6 `void* i2c_master_dma_handle_t::userData`

17.4.3 Typedef Documentation

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

17.4.4 Function Documentation

17.4.4.1 `void I2C_MasterTransferCreateHandleDMA (I2C_Type * base, i2c_master_dma_handle_t * handle, i2c_master_dma_transfer_callback_t callback, void * userData, dma_handle_t * dmaHandle)`

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	Pointer to the <code>i2c_master_dma_handle_t</code> structure
<i>callback</i>	Pointer to the user callback function
<i>userData</i>	A user parameter passed to the callback function
<i>dmaHandle</i>	DMA handle pointer

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

I2C DMA Driver

Parameters

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

Return values

<i>kStatus_Success</i>	Successfully completes the data transmission.
<i>kStatus_I2C_Busy</i>	A previous transmission is still not finished.
<i>kStatus_I2C_Timeout</i>	A transfer error, waits for the signal timeout.
<i>kStatus_I2C_Arbitration-Lost</i>	A transfer error, arbitration lost.
<i>kStatus_I2C_Nak</i>	A transfer error, receives NAK during transfer.

17.4.4.3 `status_t I2C_MasterTransferGetCountDMA (I2C_Type * base, i2c_master_dma_handle_t * handle, size_t * count)`

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	A pointer to the <code>i2c_master_dma_handle_t</code> structure
<i>count</i>	A number of bytes transferred so far by the non-blocking transaction.

17.4.4.4 `void I2C_MasterTransferAbortDMA (I2C_Type * base, i2c_master_dma_handle_t * handle)`

Parameters

<i>base</i>	I2C peripheral base address
<i>handle</i>	A pointer to the <code>i2c_master_dma_handle_t</code> structure.

17.5 I2C FreeRTOS Driver

17.5.1 Overview

I2C RTOS Operation

- status_t **I2C_RTOS_Init** (i2c_rtos_handle_t *handle, I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)
Initializes I2C.
- status_t **I2C_RTOS_Deinit** (i2c_rtos_handle_t *handle)
Deinitializes the I2C.
- status_t **I2C_RTOS_Transfer** (i2c_rtos_handle_t *handle, i2c_master_transfer_t *transfer)
Performs the I2C transfer.

17.5.2 Function Documentation

17.5.2.1 status_t I2C_RTOS_Init (i2c_rtos_handle_t * *handle*, I2C_Type * *base*, const i2c_master_config_t * *masterConfig*, uint32_t *srcClock_Hz*)

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

Parameters

<i>handle</i>	The RTOS I2C handle, the pointer to an allocated space for RTOS context.
<i>base</i>	The pointer base address of the I2C instance to initialize.
<i>masterConfig</i>	The configuration structure to set-up I2C in master mode.
<i>srcClock_Hz</i>	The frequency of an input clock of the I2C module.

Returns

status of the operation.

17.5.2.2 status_t I2C_RTOS_Deinit (i2c_rtos_handle_t * *handle*)

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

Parameters

I2C FreeRTOS Driver

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

17.5.2.3 **status_t I2C_RTOS_Transfer (i2c_rtos_handle_t * *handle*, i2c_master_transfer_t * *transfer*)**

This function performs the I2C transfer according to the data given in the transfer structure.

Parameters

<i>handle</i>	The RTOS I2C handle.
<i>transfer</i>	A structure specifying the transfer parameters.

Returns

status of the operation.

Chapter 18

LLWU: Low-Leakage Wakeup Unit Driver

18.1 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.

18.2 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.

18.3 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.

18.4 Digital pin filter for external wakeup pin configurations

Configures the digital pin filter of the external wakeup pins' working modes, gets, and clears the pin filter flags. Digital pin filters are accessed by the `filterIndex`, which is started from 1. Numbers of external pins depend on the SoC configuration.

Data Structures

- struct `llwu_external_pin_filter_mode_t`
An external input pin filter control structure. [More...](#)

Enumerations

- enum `llwu_external_pin_mode_t` {
 `kLLWU_ExternalPinDisable` = 0U,
 `kLLWU_ExternalPinRisingEdge` = 1U,
 `kLLWU_ExternalPinFallingEdge` = 2U,
 `kLLWU_ExternalPinAnyEdge` = 3U }
External input pin control modes.
- enum `llwu_pin_filter_mode_t` {
 `kLLWU_PinFilterDisable` = 0U,
 `kLLWU_PinFilterRisingEdge` = 1U,
 `kLLWU_PinFilterFallingEdge` = 2U,
 `kLLWU_PinFilterAnyEdge` = 3U }
Digital filter control modes.

Macro Definition Documentation

Driver version

- #define **FSL_LLWU_DRIVER_VERSION** (**MAKE_VERSION**(2, 0, 1))
LLWU driver version 2.0.1.

Low-Leakage Wakeup Unit Control APIs

- void **LLWU_SetExternalWakeupPinMode** (LLWU_Type *base, uint32_t pinIndex, **llwu_external_pin_mode_t** pinMode)
Sets the external input pin source mode.
- bool **LLWU_GetExternalWakeupPinFlag** (LLWU_Type *base, uint32_t pinIndex)
Gets the external wakeup source flag.
- void **LLWU_ClearExternalWakeupPinFlag** (LLWU_Type *base, uint32_t pinIndex)
Clears the external wakeup source flag.
- static void **LLWU_EnableInternalModuleInterruptWakup** (LLWU_Type *base, uint32_t moduleIndex, bool enable)
Enables/disables the internal module source.
- static bool **LLWU_GetInternalWakeupModuleFlag** (LLWU_Type *base, uint32_t moduleIndex)
Gets the external wakeup source flag.
- void **LLWU_SetPinFilterMode** (LLWU_Type *base, uint32_t filterIndex, **llwu_external_pin_filter_mode_t** filterMode)
Sets the pin filter configuration.
- bool **LLWU_GetPinFilterFlag** (LLWU_Type *base, uint32_t filterIndex)
Gets the pin filter configuration.
- void **LLWU_ClearPinFilterFlag** (LLWU_Type *base, uint32_t filterIndex)
Clears the pin filter configuration.
- void **LLWU_SetResetPinMode** (LLWU_Type *base, bool pinEnable, bool enableInLowLeakageMode)
Sets the reset pin mode.

18.5 Data Structure Documentation

18.5.1 struct llwu_external_pin_filter_mode_t

Data Fields

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

18.6 Macro Definition Documentation

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

18.7 Enumeration Type Documentation

18.7.1 enum llwu_external_pin_mode_t

Enumerator

kLLWU_ExternalPinDisable Pin disabled as a wakeup input.
kLLWU_ExternalPinRisingEdge Pin enabled with the rising edge detection.
kLLWU_ExternalPinFallingEdge Pin enabled with the falling edge detection.
kLLWU_ExternalPinAnyEdge Pin enabled with any change detection.

18.7.2 enum llwu_pin_filter_mode_t

Enumerator

kLLWU_PinFilterDisable Filter disabled.
kLLWU_PinFilterRisingEdge Filter positive edge detection.
kLLWU_PinFilterFallingEdge Filter negative edge detection.
kLLWU_PinFilterAnyEdge Filter any edge detection.

18.8 Function Documentation

18.8.1 void LLWU_SetExternalWakeupPinMode (LLWU_Type * *base*, uint32_t *pinIndex*, llwu_external_pin_mode_t *pinMode*)

This function sets the external input pin source mode that is used as a wake up source.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>pinIndex</i>	A pin index to be enabled as an external wakeup source starting from 1.
<i>pinMode</i>	A pin configuration mode defined in the llwu_external_pin_modes_t.

18.8.2 bool LLWU_GetExternalWakeupPinFlag (LLWU_Type * *base*, uint32_t *pinIndex*)

This function checks the external pin flag to detect whether the MCU is woken up by the specific pin.

Function Documentation

Parameters

<i>base</i>	LLWU peripheral base address.
<i>pinIndex</i>	A pin index, which starts from 1.

Returns

True if the specific pin is a wakeup source.

18.8.3 void LLWU_ClearExternalWakeupPinFlag (LLWU_Type * *base*, uint32_t *pinIndex*)

This function clears the external wakeup source flag for a specific pin.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>pinIndex</i>	A pin index, which starts from 1.

18.8.4 static void LLWU_EnableInternalModuleInterruptWakup (LLWU_Type * *base*, uint32_t *moduleIndex*, bool *enable*) [inline], [static]

This function enables/disables the internal module source mode that is used as a wake up source.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>moduleIndex</i>	A module index to be enabled as an internal wakeup source starting from 1.
<i>enable</i>	An enable or a disable setting

18.8.5 static bool LLWU_GetInternalWakeupModuleFlag (LLWU_Type * *base*, uint32_t *moduleIndex*) [inline], [static]

This function checks the external pin flag to detect whether the system is woken up by the specific pin.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>moduleIndex</i>	A module index, which starts from 1.

Returns

True if the specific pin is a wake up source.

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

This function sets the pin filter configuration.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>filterIndex</i>	A pin filter index used to enable/disable the digital filter, starting from 1.
<i>filterMode</i>	A filter mode configuration

18.8.7 bool LLWU_GetPinFilterFlag (LLWU_Type * *base*, uint32_t *filterIndex*)

This function gets the pin filter flag.

Parameters

<i>base</i>	LLWU peripheral base address.
<i>filterIndex</i>	A pin filter index, which starts from 1.

Returns

True if the flag is a source of the existing low-leakage power mode.

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

This function clears the pin filter flag.

Function Documentation

Parameters

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

18.8.9 void LLWU_SetResetPinMode (LLWU_Type * *base*, bool *pinEnable*, bool *enableInLowLeakageMode*)

This function determines how the reset pin is used as a low leakage mode exit source.

Parameters

<i>pinEnable</i>	Enable reset the pin filter
<i>pinFilter-Enable</i>	Specify whether the pin filter is enabled in Low-Leakage power mode.

Chapter 19

LPTMR: Low-Power Timer

19.1 Overview

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

19.2 Function groups

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

19.2.1 Initialization and deinitialization

The function [LPTMR_Init\(\)](#) initializes the LPTMR with specified configurations. The function [LPTMR_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the LPTMR for a timer or a pulse counter mode. It also sets up the LPTMR's free running mode operation and a clock source.

The function [LPTMR_DeInit\(\)](#) disables the LPTMR module and gates the module clock.

19.2.2 Timer period Operations

The function [LPTMR_SetTimerPeriod\(\)](#) sets the timer period in units of count. Timers counts from 0 to the count value set here.

The function [LPTMR_GetCurrentTimerCount\(\)](#) reads the current timer counting value. This function returns the real-time timer counting value ranging from 0 to a timer period.

The timer period operation function takes the count value in ticks. Call the utility macros provided in the `fsl_common.h` file to convert to microseconds or milliseconds.

19.2.3 Start and Stop timer operations

The function [LPTMR_StartTimer\(\)](#) starts the timer counting. After calling this function, the timer counts up to the counter value set earlier by using the [LPTMR_SetPeriod\(\)](#) function. Each time the timer reaches the count value and increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

The function [LPTMR_StopTimer\(\)](#) stops the timer counting and resets the timer's counter register.

Typical use case

19.2.4 Status

Provides functions to get and clear the LPTMR status.

19.2.5 Interrupt

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

19.3 Typical use case

19.3.1 LPTMR tick example

Updates the LPTMR period and toggles an LED periodically.

```
int main(void)
{
    uint32_t currentCounter = 0U;
    lptmr_config_t lptmrConfig;

    LED_INIT();

    /* Board pin, clock, debug console initialization */
    BOARD_InitHardware();

    /* Configures the LPTMR */
    LPTMR_GetDefaultConfig(&lptmrConfig);

    /* Initializes the LPTMR */
    LPTMR_Init(LPTMR0, &lptmrConfig);

    /* Sets the timer period */
    LPTMR_SetTimerPeriod(LPTMR0, USEC_TO_COUNT(1000000U, LPTMR_SOURCE_CLOCK));

    /* Enables a timer interrupt */
    LPTMR_EnableInterrupts(LPTMR0,
        kLPTMR_TimerInterruptEnable);

    /* Enables the NVIC */
    EnableIRQ(LPTMR0_IRQn);

    PRINTF("Low Power Timer Example\r\n");

    /* Starts counting */
    LPTMR_StartTimer(LPTMR0);
    while (1)
    {
        if (currentCounter != lptmrCounter)
        {
            currentCounter = lptmrCounter;
            PRINTF("LPTMR interrupt No.%d \r\n", currentCounter);
        }
    }
}
```

Data Structures

- struct [lptmr_config_t](#)
LPTMR config structure. [More...](#)

Enumerations

- enum `lptmr_pin_select_t` {
`kLPTMR_PinSelectInput_0` = 0x0U,
`kLPTMR_PinSelectInput_1` = 0x1U,
`kLPTMR_PinSelectInput_2` = 0x2U,
`kLPTMR_PinSelectInput_3` = 0x3U }
LPTMR pin selection used in pulse counter mode.
- enum `lptmr_pin_polarity_t` {
`kLPTMR_PinPolarityActiveHigh` = 0x0U,
`kLPTMR_PinPolarityActiveLow` = 0x1U }
LPTMR pin polarity used in pulse counter mode.
- enum `lptmr_timer_mode_t` {
`kLPTMR_TimerModeTimeCounter` = 0x0U,
`kLPTMR_TimerModePulseCounter` = 0x1U }
LPTMR timer mode selection.
- enum `lptmr_prescaler_glitch_value_t` {
`kLPTMR_Prescale_Glitch_0` = 0x0U,
`kLPTMR_Prescale_Glitch_1` = 0x1U,
`kLPTMR_Prescale_Glitch_2` = 0x2U,
`kLPTMR_Prescale_Glitch_3` = 0x3U,
`kLPTMR_Prescale_Glitch_4` = 0x4U,
`kLPTMR_Prescale_Glitch_5` = 0x5U,
`kLPTMR_Prescale_Glitch_6` = 0x6U,
`kLPTMR_Prescale_Glitch_7` = 0x7U,
`kLPTMR_Prescale_Glitch_8` = 0x8U,
`kLPTMR_Prescale_Glitch_9` = 0x9U,
`kLPTMR_Prescale_Glitch_10` = 0xAU,
`kLPTMR_Prescale_Glitch_11` = 0xBU,
`kLPTMR_Prescale_Glitch_12` = 0xCU,
`kLPTMR_Prescale_Glitch_13` = 0xDU,
`kLPTMR_Prescale_Glitch_14` = 0xEU,
`kLPTMR_Prescale_Glitch_15` = 0xFU }
LPTMR prescaler/glitch filter values.
- enum `lptmr_prescaler_clock_select_t` {
`kLPTMR_PrescalerClock_0` = 0x0U,
`kLPTMR_PrescalerClock_1` = 0x1U,
`kLPTMR_PrescalerClock_2` = 0x2U,
`kLPTMR_PrescalerClock_3` = 0x3U }
LPTMR prescaler/glitch filter clock select.
- enum `lptmr_interrupt_enable_t` { `kLPTMR_TimerInterruptEnable` = `LPTMR_CSR_TIE_MASK` }
List of the LPTMR interrupts.
- enum `lptmr_status_flags_t` { `kLPTMR_TimerCompareFlag` = `LPTMR_CSR_TCF_MASK` }
List of the LPTMR status flags.

Driver version

- #define `FSL_LPTMR_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 1)`)

Initialization and deinitialization

- void [LPTMR_Init](#) (LPTMR_Type *base, const [lptmr_config_t](#) *config)
Ungates the LPTMR clock and configures the peripheral for a basic operation.
- void [LPTMR_Deinit](#) (LPTMR_Type *base)
Gates the LPTMR clock.
- void [LPTMR_GetDefaultConfig](#) ([lptmr_config_t](#) *config)
Fills in the LPTMR configuration structure with default settings.

Interrupt Interface

- static void [LPTMR_EnableInterrupts](#) (LPTMR_Type *base, uint32_t mask)
Enables the selected LPTMR interrupts.
- static void [LPTMR_DisableInterrupts](#) (LPTMR_Type *base, uint32_t mask)
Disables the selected LPTMR interrupts.
- static uint32_t [LPTMR_GetEnabledInterrupts](#) (LPTMR_Type *base)
Gets the enabled LPTMR interrupts.

Status Interface

- static uint32_t [LPTMR_GetStatusFlags](#) (LPTMR_Type *base)
Gets the LPTMR status flags.
- static void [LPTMR_ClearStatusFlags](#) (LPTMR_Type *base, uint32_t mask)
Clears the LPTMR status flags.

Read and write the timer period

- static void [LPTMR_SetTimerPeriod](#) (LPTMR_Type *base, uint32_t ticks)
Sets the timer period in units of count.
- static uint32_t [LPTMR_GetCurrentTimerCount](#) (LPTMR_Type *base)
Reads the current timer counting value.

Timer Start and Stop

- static void [LPTMR_StartTimer](#) (LPTMR_Type *base)
Starts the timer.
- static void [LPTMR_StopTimer](#) (LPTMR_Type *base)
Stops the timer.

19.4 Data Structure Documentation

19.4.1 struct [lptmr_config_t](#)

This structure holds the configuration settings for the LPTMR peripheral. To initialize this structure to reasonable defaults, call the [LPTMR_GetDefaultConfig\(\)](#) function and pass a pointer to your configuration structure instance.

The configuration struct can be made constant so it resides in flash.

Data Fields

- [lptmr_timer_mode_t timerMode](#)
Time counter mode or pulse counter mode.
- [lptmr_pin_select_t pinSelect](#)
LPTMR pulse input pin select; used only in pulse counter mode.
- [lptmr_pin_polarity_t pinPolarity](#)
LPTMR pulse input pin polarity; used only in pulse counter mode.
- bool [enableFreeRunning](#)
True: enable free running, counter is reset on overflow False: counter is reset when the compare flag is set.
- bool [bypassPrescaler](#)
True: bypass prescaler; false: use clock from prescaler.
- [lptmr_prescaler_clock_select_t prescalerClockSource](#)
LPTMR clock source.
- [lptmr_prescaler_glitch_value_t value](#)
Prescaler or glitch filter value.

19.5 Enumeration Type Documentation

19.5.1 enum lptmr_pin_select_t

Enumerator

kLPTMR_PinSelectInput_0 Pulse counter input 0 is selected.
kLPTMR_PinSelectInput_1 Pulse counter input 1 is selected.
kLPTMR_PinSelectInput_2 Pulse counter input 2 is selected.
kLPTMR_PinSelectInput_3 Pulse counter input 3 is selected.

19.5.2 enum lptmr_pin_polarity_t

Enumerator

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

19.5.3 enum lptmr_timer_mode_t

Enumerator

kLPTMR_TimerModeTimeCounter Time Counter mode.
kLPTMR_TimerModePulseCounter Pulse Counter mode.

19.5.4 enum lptmr_prescaler_glitch_value_t

Enumerator

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

19.5.5 enum lptmr_prescaler_clock_select_t

Note

Clock connections are SoC-specific

Enumerator

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

19.5.6 enum lptmr_interrupt_enable_t

Enumerator

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

19.5.7 enum lptmr_status_flags_t

Enumerator

kLPTMR_TimerCompareFlag Timer compare flag.

19.6 Function Documentation

19.6.1 void LPTMR_Init (LPTMR_Type * *base*, const lptmr_config_t * *config*)

Note

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

Parameters

<i>base</i>	LPTMR peripheral base address
<i>config</i>	A pointer to the LPTMR configuration structure.

19.6.2 void LPTMR_Deinit (LPTMR_Type * *base*)

Parameters

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

19.6.3 void LPTMR_GetDefaultConfig (lptmr_config_t * *config*)

The default values are as follows.

```
* config->timerMode = kLPTMR_TimerModeTimeCounter;
* config->pinSelect = kLPTMR_PinSelectInput_0;
* config->pinPolarity = kLPTMR_PinPolarityActiveHigh;
* config->enableFreeRunning = false;
* config->bypassPrescaler = true;
* config->prescalerClockSource = kLPTMR_PrescalerClock_1;
* config->value = kLPTMR_Prescale_Glitch_0;
*
```

Parameters

Function Documentation

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

19.6.4 static void LPTMR_EnableInterrupts (LPTMR_Type * *base*, uint32_t *mask*)
[inline], [static]

Parameters

<i>base</i>	LPTMR peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t

19.6.5 static void LPTMR_DisableInterrupts (LPTMR_Type * *base*, uint32_t *mask*)
[inline], [static]

Parameters

<i>base</i>	LPTMR peripheral base address
<i>mask</i>	The interrupts to disable. This is a logical OR of members of the enumeration lptmr_interrupt_enable_t .

19.6.6 static uint32_t LPTMR_GetEnabledInterrupts (LPTMR_Type * *base*)
[inline], [static]

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [lptmr_interrupt_enable_t](#)

19.6.7 static uint32_t LPTMR_GetStatusFlags (LPTMR_Type * *base*) **[inline], [static]**

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration [lptmr_status_flags_t](#)

19.6.8 static void LPTMR_ClearStatusFlags (LPTMR_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	LPTMR peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration lptmr_status_flags_t .

19.6.9 static void LPTMR_SetTimerPeriod (LPTMR_Type * *base*, uint32_t *ticks*) [inline], [static]

Timers counts from 0 until it equals the count value set here. The count value is written to the CMR register.

Note

1. The TCF flag is set with the CNR equals the count provided here and then increments.
2. Call the utility macros provided in the `fsl_common.h` to convert to ticks.

Parameters

<i>base</i>	LPTMR peripheral base address
<i>ticks</i>	A timer period in units of ticks, which should be equal or greater than 1.

19.6.10 static uint32_t LPTMR_GetCurrentTimerCount (LPTMR_Type * *base*) [inline], [static]

This function returns the real-time timer counting value in a range from 0 to a timer period.

Function Documentation

Note

Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

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

Returns

The current counter value in ticks

19.6.11 `static void LPTMR_StartTimer (LPTMR_Type * base) [inline], [static]`

After calling this function, the timer counts up to the CMR register value. Each time the timer reaches the CMR value and then increments, it generates a trigger pulse and sets the timeout interrupt flag. An interrupt is also triggered if the timer interrupt is enabled.

Parameters

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

19.6.12 `static void LPTMR_StopTimer (LPTMR_Type * base) [inline], [static]`

This function stops the timer and resets the timer's counter register.

Parameters

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

Chapter 20

PDB: Programmable Delay Block

20.1 Overview

The MCUXpresso SDK provides a peripheral driver for the Programmable Delay Block (PDB) module of MCUXpresso SDK devices.

The PDB driver includes a basic PDB counter, trigger generators for ADC, DAC, and pulse-out.

The basic PDB counter can be used as a general programmable timer with an interrupt. The counter increases automatically with the divided clock signal after it is triggered to start by an external trigger input or the software trigger. There are "milestones" for the output trigger event. When the counter is equal to any of these "milestones", the corresponding trigger is generated and sent out to other modules. These "milestones" are for the following events.

- Counter delay interrupt, which is the interrupt for the PDB module
- ADC pre-trigger to trigger the ADC conversion
- DAC interval trigger to trigger the DAC buffer and move the buffer read pointer
- Pulse-out triggers to generate a single of rising and falling edges, which can be assembled to a window.

The "milestone" values have a flexible load mode. To call the APIs to set these value is equivalent to writing data to their buffer. The loading event occurs as the load mode describes. This design ensures that all "milestones" can be updated at the same time.

20.2 Typical use case

20.2.1 Working as basic PDB counter with a PDB interrupt.

```
int main(void)
{
    // ...
    EnableIRQ(DEMO_PDB_IRQ_ID);

    // ...
    // Configures the PDB counter.
    PDB_GetDefaultConfig(&pdbConfigStruct);
    PDB_Init(DEMO_PDB_INSTANCE, &pdbConfigStruct);

    // Configures the delay interrupt.
    PDB_SetModulusValue(DEMO_PDB_INSTANCE, 1000U);
    PDB_SetCounterDelayValue(DEMO_PDB_INSTANCE, 1000U); // The available delay
    value is less than or equal to the modulus value.
    PDB_EnableInterrupts(DEMO_PDB_INSTANCE,
        kPDB_DelayInterruptEnable);
    PDB_DoLoadValues(DEMO_PDB_INSTANCE);

    while (1)
    {
        // ...
        g_PdbDelayInterruptFlag = false;
    }
}
```

Typical use case

```
PDB_DoSoftwareTrigger (DEMO_PDB_INSTANCE);
while (!g_PdbDelayInterruptFlag)
{
}
}

void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
    // ...
    g_PdbDelayInterruptFlag = true;
    PDB_ClearStatusFlags (DEMO_PDB_INSTANCE,
        kPDB_DelayEventFlag);
}
```

20.2.2 Working with an additional trigger. The ADC trigger is used as an example.

```
void DEMO_PDB_IRQ_HANDLER_FUNC(void)
{
    PDB_ClearStatusFlags (DEMO_PDB_INSTANCE,
        kPDB_DelayEventFlag);
    g_PdbDelayInterruptCounter++;
    g_PdbDelayInterruptFlag = true;
}

void DEMO_PDB_InitADC(void)
{
    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

    ADC16_GetDefaultConfig(&adc16ConfigStruct);
    ADC16_Init (DEMO_PDB_ADC_INSTANCE, &adc16ConfigStruct);
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) && FSL_FEATURE_ADC16_HAS_CALIBRATION
    ADC16_EnableHardwareTrigger (DEMO_PDB_ADC_INSTANCE, false);
    ADC16_DoAutoCalibration (DEMO_PDB_ADC_INSTANCE);
#endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION */
    ADC16_EnableHardwareTrigger (DEMO_PDB_ADC_INSTANCE, true);

    adc16ChannelConfigStruct.channelNumber = DEMO_PDB_ADC_USER_CHANNEL;
    adc16ChannelConfigStruct.enableInterruptOnConversionCompleted =
        true; /* Enable the interrupt. */
#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) && FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adc16ChannelConfigStruct.enabledDifferentialConversion = false;
#endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE */
    ADC16_SetChannelConfig (DEMO_PDB_ADC_INSTANCE, DEMO_PDB_ADC_CHANNEL_GROUP, &
        adc16ChannelConfigStruct);
}

void DEMO_PDB_ADC_IRQ_HANDLER_FUNCTION(void)
{
    uint32_t tmp32;

    tmp32 = ADC16_GetChannelConversionValue (DEMO_PDB_ADC_INSTANCE,
        DEMO_PDB_ADC_CHANNEL_GROUP); /* Read to clear COCO flag. */
    g_AdcInterruptCounter++;
    g_AdcInterruptFlag = true;
}

int main(void)
{
    // ...

    EnableIRQ (DEMO_PDB_IRQ_ID);
    EnableIRQ (DEMO_PDB_ADC_IRQ_ID);
}
```

```

// ...

// Configures the PDB counter.
PDB_GetDefaultConfig(&pdbConfigStruct);
PDB_Init(DEMO_PDB_INSTANCE, &pdbConfigStruct);

// Configures the delay interrupt.
PDB_SetModulusValue(DEMO_PDB_INSTANCE, 1000U);
PDB_SetCounterDelayValue(DEMO_PDB_INSTANCE, 1000U); // The available delay
value is less than or equal to the modulus value.
PDB_EnableInterrupts(DEMO_PDB_INSTANCE,
    kPDB_DelayInterruptEnable);

// Configures the ADC pre-trigger.
pdbAdcPreTriggerConfigStruct.enablePreTriggerMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
pdbAdcPreTriggerConfigStruct.enableOutputMask = 1U << DEMO_PDB_ADC_PRETRIGGER_CHANNEL;
pdbAdcPreTriggerConfigStruct.enableBackToBackOperationMask = 0U;
PDB_SetADCPreTriggerConfig(DEMO_PDB_INSTANCE, DEMO_PDB_ADC_TRIGGER_CHANNEL, &
    pdbAdcPreTriggerConfigStruct);
PDB_SetADCPreTriggerDelayValue(DEMO_PDB_INSTANCE,
    DEMO_PDB_ADC_TRIGGER_CHANNEL, DEMO_PDB_ADC_PRETRIGGER_CHANNEL, 200U);
// The available pre-trigger delay value is less than or equal to the modulus
value.

PDB_DoLoadValues(DEMO_PDB_INSTANCE);

// Configures the ADC.
DEMO_PDB_InitADC();

while (1)
{
    g_PdbDelayInterruptFlag = false;
    g_AdcInterruptFlag = false;
    PDB_DoSoftwareTrigger(DEMO_PDB_INSTANCE);
    while ((!g_PdbDelayInterruptFlag) || (!g_AdcInterruptFlag))
    {
        // ...
    }
}

```

Data Structures

- struct [pdb_config_t](#)
PDB module configuration. [More...](#)
- struct [pdb_adc_pretrigger_config_t](#)
PDB ADC Pre-trigger configuration. [More...](#)
- struct [pdb_dac_trigger_config_t](#)
PDB DAC trigger configuration. [More...](#)

Enumerations

- enum [_pdb_status_flags](#) {
[kPDB_LoadOKFlag](#) = PDB_SC_LDOK_MASK,
[kPDB_DelayEventFlag](#) = PDB_SC_PDBIF_MASK }
PDB flags.
- enum [_pdb_adc_pretrigger_flags](#) {
[kPDB_ADCPreTriggerChannel0Flag](#) = PDB_S_CF(1U << 0),
[kPDB_ADCPreTriggerChannel1Flag](#) = PDB_S_CF(1U << 1),
[kPDB_ADCPreTriggerChannel0ErrorFlag](#) = PDB_S_ERR(1U << 0),

Typical use case

```
kPDB_ADCPreTriggerChannel1ErrorFlag = PDB_S_ERR(1U << 1) }  
    PDB ADC PreTrigger channel flags.  
• enum _pdb_interrupt_enable {  
    kPDB_SequenceErrorInterruptEnable = PDB_SC_PDBEIE_MASK,  
    kPDB_DelayInterruptEnable = PDB_SC_PDBIE_MASK }  
    PDB buffer interrupts.  
• enum pdb_load_value_mode_t {  
    kPDB_LoadValueImmediately = 0U,  
    kPDB_LoadValueOnCounterOverflow = 1U,  
    kPDB_LoadValueOnTriggerInput = 2U,  
    kPDB_LoadValueOnCounterOverflowOrTriggerInput = 3U }  
    PDB load value mode.  
• enum pdb_prescaler_divider_t {  
    kPDB_PrescalerDivider1 = 0U,  
    kPDB_PrescalerDivider2 = 1U,  
    kPDB_PrescalerDivider4 = 2U,  
    kPDB_PrescalerDivider8 = 3U,  
    kPDB_PrescalerDivider16 = 4U,  
    kPDB_PrescalerDivider32 = 5U,  
    kPDB_PrescalerDivider64 = 6U,  
    kPDB_PrescalerDivider128 = 7U }  
    Prescaler divider.  
• enum pdb_divider_multiplication_factor_t {  
    kPDB_DividerMultiplicationFactor1 = 0U,  
    kPDB_DividerMultiplicationFactor10 = 1U,  
    kPDB_DividerMultiplicationFactor20 = 2U,  
    kPDB_DividerMultiplicationFactor40 = 3U }  
    Multiplication factor select for prescaler.  
• enum pdb_trigger_input_source_t {  
    kPDB_TriggerInput0 = 0U,  
    kPDB_TriggerInput1 = 1U,  
    kPDB_TriggerInput2 = 2U,  
    kPDB_TriggerInput3 = 3U,  
    kPDB_TriggerInput4 = 4U,  
    kPDB_TriggerInput5 = 5U,  
    kPDB_TriggerInput6 = 6U,  
    kPDB_TriggerInput7 = 7U,  
    kPDB_TriggerInput8 = 8U,  
    kPDB_TriggerInput9 = 9U,  
    kPDB_TriggerInput10 = 10U,  
    kPDB_TriggerInput11 = 11U,  
    kPDB_TriggerInput12 = 12U,  
    kPDB_TriggerInput13 = 13U,  
    kPDB_TriggerInput14 = 14U,  
    kPDB_TriggerSoftware = 15U }  
    Trigger input source.
```

Driver version

- #define **FSL_PDB_DRIVER_VERSION** (**MAKE_VERSION**(2, 0, 1))
PDB driver version 2.0.1.

Initialization

- void **PDB_Init** (PDB_Type *base, const **pdb_config_t** *config)
Initializes the PDB module.
- void **PDB_Deinit** (PDB_Type *base)
De-initializes the PDB module.
- void **PDB_GetDefaultConfig** (**pdb_config_t** *config)
Initializes the PDB user configuration structure.
- static void **PDB_Enable** (PDB_Type *base, bool enable)
Enables the PDB module.

Basic Counter

- static void **PDB_DoSoftwareTrigger** (PDB_Type *base)
Triggers the PDB counter by software.
- static void **PDB_DoLoadValues** (PDB_Type *base)
Loads the counter values.
- static void **PDB_EnableDMA** (PDB_Type *base, bool enable)
Enables the DMA for the PDB module.
- static void **PDB_EnableInterrupts** (PDB_Type *base, uint32_t mask)
Enables the interrupts for the PDB module.
- static void **PDB_DisableInterrupts** (PDB_Type *base, uint32_t mask)
Disables the interrupts for the PDB module.
- static uint32_t **PDB_GetStatusFlags** (PDB_Type *base)
Gets the status flags of the PDB module.
- static void **PDB_ClearStatusFlags** (PDB_Type *base, uint32_t mask)
Clears the status flags of the PDB module.
- static void **PDB_SetModulusValue** (PDB_Type *base, uint32_t value)
Specifies the counter period.
- static uint32_t **PDB_GetCounterValue** (PDB_Type *base)
Gets the PDB counter's current value.
- static void **PDB_SetCounterDelayValue** (PDB_Type *base, uint32_t value)
Sets the value for the PDB counter delay event.

ADC Pre-trigger

- static void **PDB_SetADCPreTriggerConfig** (PDB_Type *base, uint32_t channel, **pdb_adc_pretrigger_config_t** *config)
Configures the ADC pre-trigger in the PDB module.
- static void **PDB_SetADCPreTriggerDelayValue** (PDB_Type *base, uint32_t channel, uint32_t pre-Channel, uint32_t value)
Sets the value for the ADC pre-trigger delay event.
- static uint32_t **PDB_GetADCPreTriggerStatusFlags** (PDB_Type *base, uint32_t channel)
Gets the ADC pre-trigger's status flags.
- static void **PDB_ClearADCPreTriggerStatusFlags** (PDB_Type *base, uint32_t channel, uint32_t mask)

Data Structure Documentation

Clears the ADC pre-trigger status flags.

Pulse-Out Trigger

- static void [PDB_EnablePulseOutTrigger](#) (PDB_Type *base, uint32_t channelMask, bool enable)
Enables the pulse out trigger channels.
- static void [PDB_SetPulseOutTriggerDelayValue](#) (PDB_Type *base, uint32_t channel, uint32_t value1, uint32_t value2)
Sets event values for the pulse out trigger.

20.3 Data Structure Documentation

20.3.1 struct pdb_config_t

Data Fields

- [pdb_load_value_mode_t](#) loadValueMode
Select the load value mode.
- [pdb_prescaler_divider_t](#) prescalerDivider
Select the prescaler divider.
- [pdb_divider_multiplication_factor_t](#) dividerMultiplicationFactor
Multiplication factor select for prescaler.
- [pdb_trigger_input_source_t](#) triggerInputSource
Select the trigger input source.
- bool [enableContinuousMode](#)
Enable the PDB operation in Continuous mode.

20.3.1.0.0.53 Field Documentation

20.3.1.0.0.53.1 [pdb_load_value_mode_t](#) pdb_config_t::loadValueMode

20.3.1.0.0.53.2 [pdb_prescaler_divider_t](#) pdb_config_t::prescalerDivider

20.3.1.0.0.53.3 [pdb_divider_multiplication_factor_t](#) pdb_config_t::dividerMultiplicationFactor

20.3.1.0.0.53.4 [pdb_trigger_input_source_t](#) pdb_config_t::triggerInputSource

20.3.1.0.0.53.5 bool pdb_config_t::enableContinuousMode

20.3.2 struct pdb_adc_pretrigger_config_t

Data Fields

- uint32_t [enablePreTriggerMask](#)
PDB Channel Pre-trigger Enable.
- uint32_t [enableOutputMask](#)
PDB Channel Pre-trigger Output Select.
- uint32_t [enableBackToBackOperationMask](#)
PDB Channel pre-trigger Back-to-Back Operation Enable.

20.3.2.0.0.54 Field Documentation

20.3.2.0.0.54.1 uint32_t pdb_adc_pretrigger_config_t::enablePreTriggerMask

20.3.2.0.0.54.2 uint32_t pdb_adc_pretrigger_config_t::enableOutputMask

PDB channel's corresponding pre-trigger asserts when the counter reaches the channel delay register.

20.3.2.0.0.54.3 uint32_t pdb_adc_pretrigger_config_t::enableBackToBackOperationMask

Back-to-back operation enables the ADC conversions complete to trigger the next PDB channel pre-trigger and trigger output, so that the ADC conversions can be triggered on next set of configuration and results registers.

20.3.3 struct pdb_dac_trigger_config_t

Data Fields

- bool [enableExternalTriggerInput](#)
Enables the external trigger for DAC interval counter.
- bool [enableIntervalTrigger](#)
Enables the DAC interval trigger.

20.3.3.0.0.55 Field Documentation

20.3.3.0.0.55.1 bool pdb_dac_trigger_config_t::enableExternalTriggerInput

20.3.3.0.0.55.2 bool pdb_dac_trigger_config_t::enableIntervalTrigger

20.4 Macro Definition Documentation

20.4.1 #define FSL_PDB_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

20.5 Enumeration Type Documentation

20.5.1 enum _pdb_status_flags

Enumerator

kPDB_LoadOKFlag This flag is automatically cleared when the values in buffers are loaded into the internal registers after the LDOK bit is set or the PDBEN is cleared.

kPDB_DelayEventFlag PDB timer delay event flag.

20.5.2 enum _pdb_adc_pretrigger_flags

Enumerator

kPDB_ADCPreTriggerChannel0Flag Pre-trigger 0 flag.
kPDB_ADCPreTriggerChannel1Flag Pre-trigger 1 flag.
kPDB_ADCPreTriggerChannel0ErrorFlag Pre-trigger 0 Error.
kPDB_ADCPreTriggerChannel1ErrorFlag Pre-trigger 1 Error.

20.5.3 enum _pdb_interrupt_enable

Enumerator

kPDB_SequenceErrorInterruptEnable PDB sequence error interrupt enable.
kPDB_DelayInterruptEnable PDB delay interrupt enable.

20.5.4 enum pdb_load_value_mode_t

Selects the mode to load the internal values after doing the load operation (write 1 to PDBx_SC[LDOK]). These values are for the following operations.

- PDB counter (PDBx_MOD, PDBx_IDLY)
- ADC trigger (PDBx_CHnDLYm)
- DAC trigger (PDBx_DACINTx)
- CMP trigger (PDBx_POyDLY)

Enumerator

kPDB_LoadValueImmediately Load immediately after 1 is written to LDOK.
kPDB_LoadValueOnCounterOverflow Load when the PDB counter overflows (reaches the MOD register value).
kPDB_LoadValueOnTriggerInput Load a trigger input event is detected.
kPDB_LoadValueOnCounterOverflowOrTriggerInput Load either when the PDB counter overflows or a trigger input is detected.

20.5.5 enum pdb_prescaler_divider_t

Counting uses the peripheral clock divided by multiplication factor selected by times of MULT.

Enumerator

kPDB_PrescalerDivider1 Divider x1.

kPDB_PrescalerDivider2 Divider x2.
kPDB_PrescalerDivider4 Divider x4.
kPDB_PrescalerDivider8 Divider x8.
kPDB_PrescalerDivider16 Divider x16.
kPDB_PrescalerDivider32 Divider x32.
kPDB_PrescalerDivider64 Divider x64.
kPDB_PrescalerDivider128 Divider x128.

20.5.6 enum pdb_divider_multiplication_factor_t

Selects the multiplication factor of the prescaler divider for the counter clock.

Enumerator

kPDB_DividerMultiplicationFactor1 Multiplication factor is 1.
kPDB_DividerMultiplicationFactor10 Multiplication factor is 10.
kPDB_DividerMultiplicationFactor20 Multiplication factor is 20.
kPDB_DividerMultiplicationFactor40 Multiplication factor is 40.

20.5.7 enum pdb_trigger_input_source_t

Selects the trigger input source for the PDB. The trigger input source can be internal or external (EXTRG pin), or the software trigger. See chip configuration details for the actual PDB input trigger connections.

Enumerator

kPDB_TriggerInput0 Trigger-In 0.
kPDB_TriggerInput1 Trigger-In 1.
kPDB_TriggerInput2 Trigger-In 2.
kPDB_TriggerInput3 Trigger-In 3.
kPDB_TriggerInput4 Trigger-In 4.
kPDB_TriggerInput5 Trigger-In 5.
kPDB_TriggerInput6 Trigger-In 6.
kPDB_TriggerInput7 Trigger-In 7.
kPDB_TriggerInput8 Trigger-In 8.
kPDB_TriggerInput9 Trigger-In 9.
kPDB_TriggerInput10 Trigger-In 10.
kPDB_TriggerInput11 Trigger-In 11.
kPDB_TriggerInput12 Trigger-In 12.
kPDB_TriggerInput13 Trigger-In 13.
kPDB_TriggerInput14 Trigger-In 14.
kPDB_TriggerSoftware Trigger-In 15, software trigger.

Function Documentation

20.6 Function Documentation

20.6.1 void PDB_Init (PDB_Type * *base*, const pdb_config_t * *config*)

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

- Enable the clock for PDB instance.
- Configure the PDB module.
- Enable the PDB module.

Parameters

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

20.6.2 void PDB_Deinit (PDB_Type * *base*)

Parameters

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

20.6.3 void PDB_GetDefaultConfig (pdb_config_t * *config*)

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

```
* config->loadValueMode = kPDB_LoadValueImmediately;
* config->prescalerDivider = kPDB_PrescalerDivider1;
* config->dividerMultiplicationFactor = kPDB_DividerMultiplicationFactor1
* ;
* config->triggerInputSource = kPDB_TriggerSoftware;
* config->enableContinuousMode = false;
*
```

Parameters

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

20.6.4 static void PDB_Enable (PDB_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>enable</i>	Enable the module or not.

20.6.5 static void PDB_DoSoftwareTrigger (PDB_Type * *base*) [inline], [static]

Parameters

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

20.6.6 static void PDB_DoLoadValues (PDB_Type * *base*) [inline], [static]

This function loads the counter values from the internal buffer. See "pdb_load_value_mode_t" about PDB's load mode.

Parameters

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

20.6.7 static void PDB_EnableDMA (PDB_Type * *base*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>enable</i>	Enable the feature or not.

20.6.8 static void PDB_EnableInterrupts (PDB_Type * *base*, uint32_t *mask*) [inline], [static]

Function Documentation

Parameters

<i>base</i>	PDB peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_pdb_interrupt_enable".

20.6.9 static void PDB_DisableInterrupts (PDB_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>mask</i>	Mask value for interrupts. See "_pdb_interrupt_enable".

20.6.10 static uint32_t PDB_GetStatusFlags (PDB_Type * *base*) [inline], [static]

Parameters

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

Returns

Mask value for asserted flags. See "_pdb_status_flags".

20.6.11 static void PDB_ClearStatusFlags (PDB_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>mask</i>	Mask value of flags. See "_pdb_status_flags".

20.6.12 static void PDB_SetModulusValue (PDB_Type * *base*, uint32_t *value*) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>value</i>	Setting value for the modulus. 16-bit is available.

20.6.13 static uint32_t PDB_GetCounterValue (PDB_Type * *base*) [inline], [static]

Parameters

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

Returns

PDB counter's current value.

20.6.14 static void PDB_SetCounterDelayValue (PDB_Type * *base*, uint32_t *value*) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>value</i>	Setting value for PDB counter delay event. 16-bit is available.

20.6.15 static void PDB_SetADCPreTriggerConfig (PDB_Type * *base*, uint32_t *channel*, pdb_adc_pretrigger_config_t * *config*) [inline], [static]

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.
<i>config</i>	Pointer to the configuration structure. See "pdb_adc_pretrigger_config_t".

20.6.16 `static void PDB_SetADCPreTriggerDelayValue (PDB_Type * base, uint32_t channel, uint32_t preChannel, uint32_t value) [inline], [static]`

This function sets the value for ADC pre-trigger delay event. It specifies the delay value for the channel's corresponding pre-trigger. The pre-trigger asserts when the PDB counter is equal to the set value.

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.
<i>preChannel</i>	Channel group index for ADC instance.
<i>value</i>	Setting value for ADC pre-trigger delay event. 16-bit is available.

20.6.17 `static uint32_t PDB_GetADCPreTriggerStatusFlags (PDB_Type * base,
uint32_t channel) [inline], [static]`

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.

Returns

Mask value for asserted flags. See "_pdb_adc_pretrigger_flags".

20.6.18 `static void PDB_ClearADCPreTriggerStatusFlags (PDB_Type * base,
uint32_t channel, uint32_t mask) [inline], [static]`

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for ADC instance.
<i>mask</i>	Mask value for flags. See "_pdb_adc_pretrigger_flags".

20.6.19 `static void PDB_EnablePulseOutTrigger (PDB_Type * base, uint32_t
channelMask, bool enable) [inline], [static]`

Parameters

Function Documentation

<i>base</i>	PDB peripheral base address.
<i>channelMask</i>	Channel mask value for multiple pulse out trigger channel.
<i>enable</i>	Whether the feature is enabled or not.

**20.6.20 static void PDB_SetPulseOutTriggerDelayValue (PDB_Type * *base*,
uint32_t *channel*, uint32_t *value1*, uint32_t *value2*) [inline],
[static]**

This function is used to set event values for the pulse output trigger. These pulse output trigger delay values specify the delay for the PDB Pulse-out. Pulse-out goes high when the PDB counter is equal to the pulse output high value (*value1*). Pulse-out goes low when the PDB counter is equal to the pulse output low value (*value2*).

Parameters

<i>base</i>	PDB peripheral base address.
<i>channel</i>	Channel index for pulse out trigger channel.
<i>value1</i>	Setting value for pulse out high.
<i>value2</i>	Setting value for pulse out low.

Chapter 21

PIT: Periodic Interrupt Timer

21.1 Overview

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

21.2 Function groups

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

21.2.1 Initialization and deinitialization

The function [PIT_Init\(\)](#) initializes the PIT with specified configurations. The function [PIT_GetDefaultConfig\(\)](#) gets the default configurations. The initialization function configures the PIT operation in debug mode.

The function [PIT_SetTimerChainMode\(\)](#) configures the chain mode operation of each PIT channel.

The function [PIT_Deinit\(\)](#) disables the PIT timers and disables the module clock.

21.2.2 Timer period Operations

The function [PITR_SetTimerPeriod\(\)](#) sets the timer period in units of count. Timers begin counting down from the value set by this function until it reaches 0.

The function [PIT_GetCurrentTimerCount\(\)](#) reads the current timer counting value. This function returns the real-time timer counting value, in a range from 0 to a timer period.

The timer period operation functions takes the count value in ticks. Users can call the utility macros provided in `fsl_common.h` to convert to microseconds or milliseconds.

21.2.3 Start and Stop timer operations

The function [PIT_StartTimer\(\)](#) starts the timer counting. After calling this function, the timer loads the period value set earlier via the [PIT_SetPeriod\(\)](#) function and starts counting down to 0. When the timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

The function [PIT_StopTimer\(\)](#) stops the timer counting.

Typical use case

21.2.4 Status

Provides functions to get and clear the PIT status.

21.2.5 Interrupt

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

21.3 Typical use case

21.3.1 PIT tick example

Updates the PIT period and toggles an LED periodically.

```
int main(void)
{
    /* Structure of initialize PIT */
    pit_config_t pitConfig;

    /* Initialize and enable LED */
    LED_INIT();

    /* Board pin, clock, debug console init */
    BOARD_InitHardware();

    PIT_GetDefaultConfig(&pitConfig);

    /* Init pit module */
    PIT_Init(PIT, &pitConfig);

    /* Set timer period for channel 0 */
    PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, USEC_TO_COUNT(1000000U,
        PIT_SOURCE_CLOCK));

    /* Enable timer interrupts for channel 0 */
    PIT_EnableInterrupts(PIT, kPIT_Chnl_0,
        kPIT_TimerInterruptEnable);

    /* Enable at the NVIC */
    EnableIRQ(PIT_IRQ_ID);

    /* Start channel 0 */
    PRINTF("\r\nStarting channel No.0 ...");
    PIT_StartTimer(PIT, kPIT_Chnl_0);

    while (true)
    {
        /* Check whether occur interrupt and toggle LED */
        if (true == pitIsrFlag)
        {
            PRINTF("\r\n Channel No.0 interrupt is occurred !");
            LED_TOGGLE();
            pitIsrFlag = false;
        }
    }
}
```

Data Structures

- struct `pit_config_t`
PIT configuration structure. [More...](#)

Enumerations

- enum `pit_chnl_t` {
 `kPIT_Chnl_0` = 0U,
 `kPIT_Chnl_1`,
 `kPIT_Chnl_2`,
 `kPIT_Chnl_3` }
List of PIT channels.
- enum `pit_interrupt_enable_t` { `kPIT_TimerInterruptEnable` = `PIT_TCTRL_TIE_MASK` }
List of PIT interrupts.
- enum `pit_status_flags_t` { `kPIT_TimerFlag` = `PIT_TFLG_TIF_MASK` }
List of PIT status flags.

Driver version

- #define `FSL_PIT_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 0)`)
Version 2.0.0.

Initialization and deinitialization

- void `PIT_Init` (`PIT_Type *base`, const `pit_config_t *config`)
Un-gates the PIT clock, enables the PIT module, and configures the peripheral for basic operations.
- void `PIT_Deinit` (`PIT_Type *base`)
Gates the PIT clock and disables the PIT module.
- static void `PIT_GetDefaultConfig` (`pit_config_t *config`)
Fills in the PIT configuration structure with the default settings.

Interrupt Interface

- static void `PIT_EnableInterrupts` (`PIT_Type *base`, `pit_chnl_t` channel, `uint32_t` mask)
Enables the selected PIT interrupts.
- static void `PIT_DisableInterrupts` (`PIT_Type *base`, `pit_chnl_t` channel, `uint32_t` mask)
Disables the selected PIT interrupts.
- static `uint32_t` `PIT_GetEnabledInterrupts` (`PIT_Type *base`, `pit_chnl_t` channel)
Gets the enabled PIT interrupts.

Status Interface

- static `uint32_t` `PIT_GetStatusFlags` (`PIT_Type *base`, `pit_chnl_t` channel)
Gets the PIT status flags.
- static void `PIT_ClearStatusFlags` (`PIT_Type *base`, `pit_chnl_t` channel, `uint32_t` mask)
Clears the PIT status flags.

Enumeration Type Documentation

Read and Write the timer period

- static void [PIT_SetTimerPeriod](#) (PIT_Type *base, [pit_chnl_t](#) channel, uint32_t count)
Sets the timer period in units of count.
- static uint32_t [PIT_GetCurrentTimerCount](#) (PIT_Type *base, [pit_chnl_t](#) channel)
Reads the current timer counting value.

Timer Start and Stop

- static void [PIT_StartTimer](#) (PIT_Type *base, [pit_chnl_t](#) channel)
Starts the timer counting.
- static void [PIT_StopTimer](#) (PIT_Type *base, [pit_chnl_t](#) channel)
Stops the timer counting.

21.4 Data Structure Documentation

21.4.1 struct pit_config_t

This structure holds the configuration settings for the PIT peripheral. To initialize this structure to reasonable defaults, call the [PIT_GetDefaultConfig\(\)](#) function and pass a pointer to your config structure instance.

The configuration structure can be made constant so it resides in flash.

Data Fields

- bool [enableRunInDebug](#)
true: Timers run in debug mode; false: Timers stop in debug mode

21.5 Enumeration Type Documentation

21.5.1 enum pit_chnl_t

Note

Actual number of available channels is SoC dependent

Enumerator

kPIT_Chnl_0 PIT channel number 0.
kPIT_Chnl_1 PIT channel number 1.
kPIT_Chnl_2 PIT channel number 2.
kPIT_Chnl_3 PIT channel number 3.

21.5.2 enum pit_interrupt_enable_t

Enumerator

kPIT_TimerInterruptEnable Timer interrupt enable.

21.5.3 enum pit_status_flags_t

Enumerator

kPIT_TimerFlag Timer flag.

21.6 Function Documentation

21.6.1 void PIT_Init (PIT_Type * *base*, const pit_config_t * *config*)

Note

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

Parameters

<i>base</i>	PIT peripheral base address
<i>config</i>	Pointer to the user's PIT config structure

21.6.2 void PIT_Deinit (PIT_Type * *base*)

Parameters

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

21.6.3 static void PIT_GetDefaultConfig (pit_config_t * *config*) [inline], [static]

The default values are as follows.

```
* config->enableRunInDebug = false;
*
```

Function Documentation

Parameters

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

21.6.4 static void PIT_EnableInterrupts (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration pit_interrupt_enable_t

21.6.5 static void PIT_DisableInterrupts (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number
<i>mask</i>	The interrupts to disable. This is a logical OR of members of the enumeration pit_interrupt_enable_t

21.6.6 static uint32_t PIT_GetEnabledInterrupts (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [pit_interrupt_enable_t](#)

21.6.7 `static uint32_t PIT_GetStatusFlags (PIT_Type * base, pit_chnl_t channel)`
`[inline], [static]`

Function Documentation

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number

Returns

The status flags. This is the logical OR of members of the enumeration [pit_status_flags_t](#)

21.6.8 static void PIT_ClearStatusFlags (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration pit_status_flags_t

21.6.9 static void PIT_SetTimerPeriod (PIT_Type * *base*, pit_chnl_t *channel*, uint32_t *count*) [inline], [static]

Timers begin counting from the value set by this function until it reaches 0, then it generates an interrupt and load this register value again. Writing a new value to this register does not restart the timer. Instead, the value is loaded after the timer expires.

Note

Users can call the utility macros provided in `fsl_common.h` to convert to ticks.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number

<i>count</i>	Timer period in units of ticks
--------------	--------------------------------

21.6.10 **static uint32_t PIT_GetCurrentTimerCount (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]**

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note

Users can call the utility macros provided in fsl_common.h to convert ticks to usec or msec.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number

Returns

Current timer counting value in ticks

21.6.11 **static void PIT_StartTimer (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]**

After calling this function, timers load period value, count down to 0 and then load the respective start value again. Each time a timer reaches 0, it generates a trigger pulse and sets the timeout interrupt flag.

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number.

21.6.12 **static void PIT_StopTimer (PIT_Type * *base*, pit_chnl_t *channel*) [inline], [static]**

This function stops every timer counting. Timers reload their periods respectively after the next time they call the PIT_DRV_StartTimer.

Function Documentation

Parameters

<i>base</i>	PIT peripheral base address
<i>channel</i>	Timer channel number.

Chapter 22

PMC: Power Management Controller

22.1 Overview

The MCUXpresso SDK provides a Peripheral driver for the Power Management Controller (PMC) module of MCUXpresso SDK devices. The PMC module contains internal voltage regulator, power on reset, low-voltage detect system, and high-voltage detect system.

Data Structures

- struct [pmc_low_volt_detect_config_t](#)
Low-voltage Detect Configuration Structure. [More...](#)
- struct [pmc_low_volt_warning_config_t](#)
Low-voltage Warning Configuration Structure. [More...](#)

Driver version

- #define [FSL_PMC_DRIVER_VERSION](#) ([MAKE_VERSION](#)(2, 0, 0))
PMC driver version.

Power Management Controller Control APIs

- void [PMC_ConfigureLowVoltDetect](#) (PMC_Type *base, const [pmc_low_volt_detect_config_t](#) *config)
Configures the low-voltage detect setting.
- static bool [PMC_GetLowVoltDetectFlag](#) (PMC_Type *base)
Gets the Low-voltage Detect Flag status.
- static void [PMC_ClearLowVoltDetectFlag](#) (PMC_Type *base)
Acknowledges clearing the Low-voltage Detect flag.
- void [PMC_ConfigureLowVoltWarning](#) (PMC_Type *base, const [pmc_low_volt_warning_config_t](#) *config)
Configures the low-voltage warning setting.
- static bool [PMC_GetLowVoltWarningFlag](#) (PMC_Type *base)
Gets the Low-voltage Warning Flag status.
- static void [PMC_ClearLowVoltWarningFlag](#) (PMC_Type *base)
Acknowledges the Low-voltage Warning flag.

22.2 Data Structure Documentation

22.2.1 struct pmc_low_volt_detect_config_t

Data Fields

- bool [enableInt](#)

Function Documentation

- *Enable interrupt when Low-voltage detect.*
• bool [enableReset](#)
Enable system reset when Low-voltage detect.

22.2.2 struct pmc_low_volt_warning_config_t

Data Fields

- bool [enableInt](#)
Enable interrupt when low-voltage warning.

22.3 Macro Definition Documentation

22.3.1 #define FSL_PMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 0))

Version 2.0.0.

22.4 Function Documentation

22.4.1 void PMC_ConfigureLowVoltDetect (PMC_Type * *base*, const pmc_low_volt_detect_config_t * *config*)

This function configures the low-voltage detect setting, including the trip point voltage setting, enables or disables the interrupt, enables or disables the system reset.

Parameters

<i>base</i>	PMC peripheral base address.
<i>config</i>	Low-voltage detect configuration structure.

22.4.2 static bool PMC_GetLowVoltDetectFlag (PMC_Type * *base*) [inline], [static]

This function reads the current LVDF status. If it returns 1, a low-voltage event is detected.

Parameters

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

Returns

Current low-voltage detect flag

- true: Low-voltage detected
- false: Low-voltage not detected

22.4.3 static void **PMC_ClearLowVoltDetectFlag** (**PMC_Type** * *base*) [**inline**], [**static**]

This function acknowledges the low-voltage detection errors (write 1 to clear LVDF).

Parameters

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

22.4.4 void **PMC_ConfigureLowVoltWarning** (**PMC_Type** * *base*, const **pmc_low_volt_warning_config_t** * *config*)

This function configures the low-voltage warning setting, including the trip point voltage setting and enabling or disabling the interrupt.

Parameters

<i>base</i>	PMC peripheral base address.
<i>config</i>	Low-voltage warning configuration structure.

22.4.5 static bool **PMC_GetLowVoltWarningFlag** (**PMC_Type** * *base*) [**inline**], [**static**]

This function polls the current LVWF status. When 1 is returned, it indicates a low-voltage warning event. LVWF is set when V Supply transitions below the trip point or after reset and V Supply is already below the V LVW.

Parameters

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

Returns

Current LVWF status

- true: Low-voltage Warning Flag is set.
- false: the Low-voltage Warning does not happen.

Function Documentation

22.4.6 static void PMC_ClearLowVoltWarningFlag (PMC_Type * *base*)
[inline], [static]

This function acknowledges the low voltage warning errors (write 1 to clear LVWF).

Parameters

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

Chapter 23

PORT: Port Control and Interrupts

23.1 Overview

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

23.2 Typical configuration use case

23.2.1 Input PORT configuration

```
/* Input pin PORT configuration */
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainDisable,
    kPORT_LowDriveStrength,
    kPORT_MuxAsGpio,
    kPORT_UnLockRegister,
};
/* Sets the configuration */
PORT_SetPinConfig(PORTA, 4, &config);
```

23.2.2 I2C PORT Configuration

```
/* I2C pin PORTconfiguration */
port_pin_config_t config = {
    kPORT_PullUp,
    kPORT_FastSlewRate,
    kPORT_PassiveFilterDisable,
    kPORT_OpenDrainEnable,
    kPORT_LowDriveStrength,
    kPORT_MuxAlt5,
    kPORT_UnLockRegister,
};
PORT_SetPinConfig(PORTE, 24u, &config);
PORT_SetPinConfig(PORTE, 25u, &config);
```

Enumerations

- enum `port_interrupt_t` {
 `kPORT_InterruptOrDMADisabled` = 0x0U,
 `kPORT_InterruptLogicZero` = 0x8U,
 `kPORT_InterruptRisingEdge` = 0x9U,
 `kPORT_InterruptFallingEdge` = 0xAU,
 `kPORT_InterruptEitherEdge` = 0xBU,
 `kPORT_InterruptLogicOne` = 0xCU }
 Configures the interrupt generation condition.

Enumeration Type Documentation

Driver version

- #define `FSL_PORT_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 2)`)
Version 2.0.2.

23.3 Macro Definition Documentation

23.3.1 #define `FSL_PORT_DRIVER_VERSION` (`MAKE_VERSION(2, 0, 2)`)

23.4 Enumeration Type Documentation

23.4.1 enum `port_interrupt_t`

Enumerator

- `kPORT_InterruptOrDMADisabled`*** Interrupt/DMA request is disabled.
- `kPORT_InterruptLogicZero`*** Interrupt when logic zero.
- `kPORT_InterruptRisingEdge`*** Interrupt on rising edge.
- `kPORT_InterruptFallingEdge`*** Interrupt on falling edge.
- `kPORT_InterruptEitherEdge`*** Interrupt on either edge.
- `kPORT_InterruptLogicOne`*** Interrupt when logic one.

Chapter 24

RCM: Reset Control Module Driver

24.1 Overview

The MCUXpresso SDK provides a Peripheral driver for the Reset Control Module (RCM) module of MCUXpresso SDK devices.

Data Structures

- struct [rcm_reset_pin_filter_config_t](#)
Reset pin filter configuration. [More...](#)

Enumerations

- enum [rcm_reset_source_t](#) {
 [kRCM_SourceLvd](#) = RCM_SRS0_LVD_MASK,
 [kRCM_SourceWdog](#) = RCM_SRS0_WDOG_MASK,
 [kRCM_SourcePin](#) = RCM_SRS0_PIN_MASK,
 [kRCM_SourcePor](#) = RCM_SRS0_POR_MASK,
 [kRCM_SourceLockup](#) = RCM_SRS1_LOCKUP_MASK << 8U,
 [kRCM_SourceSw](#) = RCM_SRS1_SW_MASK << 8U,
 [kRCM_SourceSackerr](#) = RCM_SRS1_SACKERR_MASK << 8U }
 System Reset Source Name definitions.
- enum [rcm_run_wait_filter_mode_t](#) {
 [kRCM_FilterDisable](#) = 0U,
 [kRCM_FilterBusClock](#) = 1U,
 [kRCM_FilterLpoClock](#) = 2U }
 Reset pin filter select in Run and Wait modes.

Driver version

- #define [FSL_RCM_DRIVER_VERSION](#) ([MAKE_VERSION](#)(2, 0, 1))
 RCM driver version 2.0.1.

Reset Control Module APIs

- static uint32_t [RCM_GetPreviousResetSources](#) (RCM_Type *base)
 Gets the reset source status which caused a previous reset.
- void [RCM_ConfigureResetPinFilter](#) (RCM_Type *base, const [rcm_reset_pin_filter_config_t](#) *config)
 Configures the reset pin filter.

Enumeration Type Documentation

24.2 Data Structure Documentation

24.2.1 struct rcm_reset_pin_filter_config_t

Data Fields

- bool [enableFilterInStop](#)
Reset pin filter select in stop mode.
- [rcm_run_wait_filter_mode_t](#) [filterInRunWait](#)
Reset pin filter in run/wait mode.
- uint8_t [busClockFilterCount](#)
Reset pin bus clock filter width.

24.2.1.0.0.56 Field Documentation

24.2.1.0.0.56.1 bool rcm_reset_pin_filter_config_t::enableFilterInStop

24.2.1.0.0.56.2 rcm_run_wait_filter_mode_t rcm_reset_pin_filter_config_t::filterInRunWait

24.2.1.0.0.56.3 uint8_t rcm_reset_pin_filter_config_t::busClockFilterCount

24.3 Macro Definition Documentation

24.3.1 #define FSL_RCM_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

24.4 Enumeration Type Documentation

24.4.1 enum rcm_reset_source_t

Enumerator

kRCM_SourceLvd Low-voltage detect reset.
kRCM_SourceWdog Watchdog reset.
kRCM_SourcePin External pin reset.
kRCM_SourcePor Power on reset.
kRCM_SourceLockup Core lock up reset.
kRCM_SourceSw Software reset.
kRCM_SourceSackerr Parameter could get all reset flags.

24.4.2 enum rcm_run_wait_filter_mode_t

Enumerator

kRCM_FilterDisable All filtering disabled.
kRCM_FilterBusClock Bus clock filter enabled.
kRCM_FilterLpoClock LPO clock filter enabled.

24.5 Function Documentation

24.5.1 static uint32_t RCM_GetPreviousResetSources (RCM_Type * *base*) [inline], [static]

This function gets the current reset source status. Use source masks defined in the `rcm_reset_source_t` to get the desired source status.

This is an example.

```
uint32_t resetStatus;

// To get all reset source statuses.
resetStatus = RCM_GetPreviousResetSources(RCM) & kRCM_SourceAll;

// To test whether the MCU is reset using Watchdog.
resetStatus = RCM_GetPreviousResetSources(RCM) &
              kRCM_SourceWdog;

// To test multiple reset sources.
resetStatus = RCM_GetPreviousResetSources(RCM) & (
              kRCM_SourceWdog | kRCM_SourcePin);
```

Parameters

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

Returns

All reset source status bit map.

24.5.2 void RCM_ConfigureResetPinFilter (RCM_Type * *base*, const rcm_reset_pin_filter_config_t * *config*)

This function sets the reset pin filter including the filter source, filter width, and so on.

Parameters

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

Chapter 25

RNGA: Random Number Generator Accelerator Driver

25.1 Overview

The MCUXpresso SDK provides Peripheral driver for the Random Number Generator Accelerator (RNGA) block of MCUXpresso SDK devices.

25.2 RNGA Initialization

1. To initialize the RNGA module, call the [RNGA_Init\(\)](#) function. This function automatically enables the RNGA module and its clock.
2. After calling the [RNGA_Init\(\)](#) function, the RNGA is enabled and the counter starts working.
3. To disable the RNGA module, call the [RNGA_Deinit\(\)](#) function.

25.3 Get random data from RNGA

1. [RNGA_GetRandomData\(\)](#) function gets random data from the RNGA module.

25.4 RNGA Set/Get Working Mode

The RNGA works either in sleep mode or normal mode

1. [RNGA_SetMode\(\)](#) function sets the RNGA mode.
2. [RNGA_GetMode\(\)](#) function gets the RNGA working mode.

25.5 Seed RNGA

1. [RNGA_Seed\(\)](#) function inputs an entropy value that the RNGA can use to seed the pseudo random algorithm.

This example code shows how to initialize and get random data from the RNGA driver:

```
{
    status_t      status;
    uint32_t      data;

    /* Initialize RNGA */
    status = RNGA_Init(RNG);

    /* Read Random data*/
    status = RNGA_GetRandomData(RNG, data, sizeof(data));

    if(status == kStatus_Success)
    {
        /* Print data*/
        PRINTF("Random = 0x%X\r\n", i, data );
        PRINTF("Succeed.\r\n");
    }
    else
    {

```

Seed RNGA

```
        PRINTF("RNGA failed! (0x%x)\r\n", status);
    }

    /* Deinitialize RNGA */
    RNGA_Deinit(RNG);
}
```

Note

It is important to note that there is no known cryptographic proof showing this is a secure method for generating random data. In fact, there may be an attack against this random number generator if its output is used directly in a cryptographic application. The attack is based on the linearity of the internal shift registers. Therefore, it is highly recommended that the random data produced by this module be used as an entropy source to provide an input seed to a NIST-approved pseudo-random-number generator based on DES or SHA-1 and defined in NIST FIPS PUB 186-2 Appendix 3 and NIST FIPS PUB SP 800-90. The requirement is needed to maximize the entropy of this input seed. To do this, when data is extracted from RNGA as quickly as the hardware allows, there are one to two bits of added entropy per 32-bit word. Any single bit of that word contains that entropy. Therefore, when used as an entropy source, a random number should be generated for each bit of entropy required and the least significant bit (any bit would be equivalent) of each word retained. The remainder of each random number should then be discarded. Used this way, even with full knowledge of the internal state of RNGA and all prior random numbers, an attacker is not able to predict the values of the extracted bits. Other sources of entropy can be used along with RNGA to generate the seed to the pseudorandom algorithm. The more random sources combined to create the seed, the better. The following is a list of sources that can be easily combined with the output of this module.

- Current time using highest precision possible
- Real-time system inputs that can be characterized as "random"
- Other entropy supplied directly by the user

Enumerations

- enum `rnga_mode_t` {
 `kRNGA_ModeNormal` = 0U,
 `kRNGA_ModeSleep` = 1U }
 RNGA working mode.

Functions

- void `RNGA_Init` (RNG_Type *base)
 Initializes the RNGA.
- void `RNGA_Deinit` (RNG_Type *base)
 Shuts down the RNGA.
- status_t `RNGA_GetRandomData` (RNG_Type *base, void *data, size_t data_size)
 Gets random data.
- void `RNGA_Seed` (RNG_Type *base, uint32_t seed)
 Feeds the RNGA module.
- void `RNGA_SetMode` (RNG_Type *base, `rnga_mode_t` mode)

- *Sets the RNGA in normal mode or sleep mode.*
rnga_mode_t RNGA_GetMode (RNG_Type *base)
Gets the RNGA working mode.

Driver version

- #define **FSL_RNGA_DRIVER_VERSION** (MAKE_VERSION(2, 0, 1))
RNGA driver version 2.0.1.

25.6 Macro Definition Documentation

25.6.1 #define FSL_RNGA_DRIVER_VERSION (MAKE_VERSION(2, 0, 1))

25.7 Enumeration Type Documentation

25.7.1 enum rnga_mode_t

Enumerator

kRNGA_ModeNormal Normal Mode. The ring-oscillator clocks are active; RNGA generates entropy (randomness) from the clocks and stores it in shift registers.

kRNGA_ModeSleep Sleep Mode. The ring-oscillator clocks are inactive; RNGA does not generate entropy.

25.8 Function Documentation

25.8.1 void RNGA_Init (RNG_Type * *base*)

This function initializes the RNGA. When called, the RNGA entropy generation starts immediately.

Parameters

<i>base</i>	RNGA base address
-------------	-------------------

25.8.2 void RNGA_Deinit (RNG_Type * *base*)

This function shuts down the RNGA.

Parameters

Function Documentation

<i>base</i>	RNGA base address
-------------	-------------------

25.8.3 **status_t RNGA_GetRandomData (RNG_Type * *base*, void * *data*, size_t *data_size*)**

This function gets random data from the RNGA.

Parameters

<i>base</i>	RNGA base address
<i>data</i>	pointer to user buffer to be filled by random data
<i>data_size</i>	size of data in bytes

Returns

RNGA status

25.8.4 **void RNGA_Seed (RNG_Type * *base*, uint32_t *seed*)**

This function inputs an entropy value that the RNGA uses to seed its pseudo-random algorithm.

Parameters

<i>base</i>	RNGA base address
<i>seed</i>	input seed value

25.8.5 **void RNGA_SetMode (RNG_Type * *base*, rnga_mode_t *mode*)**

This function sets the RNGA in sleep mode or normal mode.

Parameters

<i>base</i>	RNGA base address
-------------	-------------------

<i>mode</i>	normal mode or sleep mode
-------------	---------------------------

25.8.6 `rnga_mode_t` `RNGA_GetMode (RNG_Type * base)`

This function gets the RNGA working mode.

Parameters

<i>base</i>	RNGA base address
-------------	-------------------

Returns

normal mode or sleep mode

Chapter 26

RTC: Real Time Clock

26.1 Overview

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

26.2 Function groups

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

26.2.1 Initialization and deinitialization

The function [RTC_Init\(\)](#) initializes the RTC with specified configurations. The function [RTC_GetDefaultConfig\(\)](#) gets the default configurations.

The function [RTC_Deinit\(\)](#) disables the RTC timer and disables the module clock.

26.2.2 Set & Get Datetime

The function [RTC_SetDatetime\(\)](#) sets the timer period in seconds. Users pass in the details in date & time format by using the below data structure.

```
typedef struct _rtc_datetime
{
    uint16_t year;
    uint8_t month;
    uint8_t day;
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
} rtc_datetime_t;
```

The function [RTC_GetDatetime\(\)](#) reads the current timer value in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

26.2.3 Set & Get Alarm

The function [RTC_SetAlarm\(\)](#) sets the alarm time period in seconds. Users pass in the details in date & time format by using the datetime data structure.

The function [RTC_GetAlarm\(\)](#) reads the alarm time in seconds, converts it to date & time format and stores it into a datetime structure passed in by the user.

Typical use case

26.2.4 Start & Stop timer

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

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

26.2.5 Status

Provides functions to get and clear the RTC status.

26.2.6 Interrupt

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

26.2.7 RTC Oscillator

Some SoC's allow control of the RTC oscillator through the RTC module.

The function `RTC_SetOscCapLoad()` allows the user to modify the capacitor load configuration of the RTC oscillator.

26.2.8 Monotonic Counter

Some SoC's have a 64-bit Monotonic counter available in the RTC module.

The function `RTC_SetMonotonicCounter()` writes a 64-bit to the counter.

The function `RTC_GetMonotonicCounter()` reads the monotonic counter and returns the 64-bit counter value to the user.

The function `RTC_IncrementMonotonicCounter()` increments the Monotonic Counter by one.

26.3 Typical use case

26.3.1 RTC tick example

Example to set the RTC current time and trigger an alarm.

```
int main(void)
{
    uint32_t sec;
    uint32_t currSeconds;
    rtc_datetime_t date;
    rtc_config_t rtcConfig;

    /* Board pin, clock, debug console init */
```

```

BOARD_InitHardware();
/* Init RTC */
RTC_GetDefaultConfig(&rtcConfig);
RTC_Init(RTC, &rtcConfig);
/* Select RTC clock source */
BOARD_SetRtcClockSource();

PRINTF("RTC example: set up time to wake up an alarm\r\n");

/* Set a start date time and start RT */
date.year = 2014U;
date.month = 12U;
date.day = 25U;
date.hour = 19U;
date.minute = 0;
date.second = 0;

/* RTC time counter has to be stopped before setting the date & time in the TSR register */
RTC_StopTimer(RTC);

/* Set RTC time to default */
RTC_SetDatetime(RTC, &date);

/* Enable RTC alarm interrupt */
RTC_EnableInterrupts(RTC, kRTC_AlarmInterruptEnable);

/* Enable at the NVIC */
EnableIRQ(RTC_IRQn);

/* Start the RTC time counter */
RTC_StartTimer(RTC);

/* This loop will set the RTC alarm */
while (1)
{
    busyWait = true;
    /* Get date time */
    RTC_GetDatetime(RTC, &date);

    /* print default time */
    PRINTF("Current datetime: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day, date.hour,
        date.minute, date.second);

    /* Get alarm time from the user */
    sec = 0;
    PRINTF("Input the number of second to wait for alarm \r\n");
    PRINTF("The second must be positive value\r\n");
    while (sec < 1)
    {
        SCANF("%d", &sec);
    }

    /* Read the RTC seconds register to get current time in seconds */
    currSeconds = RTC->TSR;

    /* Add alarm seconds to current time */
    currSeconds += sec;

    /* Set alarm time in seconds */
    RTC->TAR = currSeconds;

    /* Get alarm time */
    RTC_GetAlarm(RTC, &date);

    /* Print alarm time */
    PRINTF("Alarm will occur at: %04hd-%02hd-%02hd %02hd:%02hd:%02hd\r\n", date.
year, date.month, date.day,

```

Typical use case

```
    date.hour, date.minute, date.second);

    /* Wait until alarm occurs */
    while (busyWait)
    {
    }

    PRINTF("\r\n Alarm occurs !!!! ");
}
}
```

Data Structures

- struct [rtc_datetime_t](#)
Structure is used to hold the date and time. [More...](#)
- struct [rtc_config_t](#)
RTC config structure. [More...](#)

Enumerations

- enum [rtc_interrupt_enable_t](#) {
 [kRTC_TimeInvalidInterruptEnable](#) = RTC_IER_TIIE_MASK,
 [kRTC_TimeOverflowInterruptEnable](#) = RTC_IER_TOIE_MASK,
 [kRTC_AlarmInterruptEnable](#) = RTC_IER_TAIE_MASK,
 [kRTC_SecondsInterruptEnable](#) = RTC_IER_TSIE_MASK }
List of RTC interrupts.
- enum [rtc_status_flags_t](#) {
 [kRTC_TimeInvalidFlag](#) = RTC_SR_TIF_MASK,
 [kRTC_TimeOverflowFlag](#) = RTC_SR_TOF_MASK,
 [kRTC_AlarmFlag](#) = RTC_SR_TAF_MASK }
List of RTC flags.

Functions

- static void [RTC_Reset](#) (RTC_Type *base)
Performs a software reset on the RTC module.

Driver version

- #define [FSL_RTC_DRIVER_VERSION](#) ([MAKE_VERSION](#)(2, 0, 0))
Version 2.0.0.

Initialization and deinitialization

- void [RTC_Init](#) (RTC_Type *base, const [rtc_config_t](#) *config)
Ungates the RTC clock and configures the peripheral for basic operation.
- static void [RTC_Deinit](#) (RTC_Type *base)
Stops the timer and gate the RTC clock.
- void [RTC_GetDefaultConfig](#) ([rtc_config_t](#) *config)
Fills in the RTC config struct with the default settings.

Current Time & Alarm

- status_t [RTC_SetDatetime](#) (RTC_Type *base, const [rtc_datetime_t](#) *datetime)
Sets the RTC date and time according to the given time structure.
- void [RTC_GetDatetime](#) (RTC_Type *base, [rtc_datetime_t](#) *datetime)
Gets the RTC time and stores it in the given time structure.
- status_t [RTC_SetAlarm](#) (RTC_Type *base, const [rtc_datetime_t](#) *alarmTime)
Sets the RTC alarm time.
- void [RTC_GetAlarm](#) (RTC_Type *base, [rtc_datetime_t](#) *datetime)
Returns the RTC alarm time.

Interrupt Interface

- static void [RTC_EnableInterrupts](#) (RTC_Type *base, uint32_t mask)
Enables the selected RTC interrupts.
- static void [RTC_DisableInterrupts](#) (RTC_Type *base, uint32_t mask)
Disables the selected RTC interrupts.
- static uint32_t [RTC_GetEnabledInterrupts](#) (RTC_Type *base)
Gets the enabled RTC interrupts.

Status Interface

- static uint32_t [RTC_GetStatusFlags](#) (RTC_Type *base)
Gets the RTC status flags.
- void [RTC_ClearStatusFlags](#) (RTC_Type *base, uint32_t mask)
Clears the RTC status flags.

Timer Start and Stop

- static void [RTC_StartTimer](#) (RTC_Type *base)
Starts the RTC time counter.
- static void [RTC_StopTimer](#) (RTC_Type *base)
Stops the RTC time counter.

26.4 Data Structure Documentation

26.4.1 struct rtc_datetime_t

Data Fields

- uint16_t [year](#)
Range from 1970 to 2099.
- uint8_t [month](#)
Range from 1 to 12.
- uint8_t [day](#)
Range from 1 to 31 (depending on month).
- uint8_t [hour](#)
Range from 0 to 23.
- uint8_t [minute](#)
Range from 0 to 59.

Enumeration Type Documentation

- uint8_t [second](#)
Range from 0 to 59.

26.4.1.0.0.57 Field Documentation

26.4.1.0.0.57.1 uint16_t rtc_datetime_t::year

26.4.1.0.0.57.2 uint8_t rtc_datetime_t::month

26.4.1.0.0.57.3 uint8_t rtc_datetime_t::day

26.4.1.0.0.57.4 uint8_t rtc_datetime_t::hour

26.4.1.0.0.57.5 uint8_t rtc_datetime_t::minute

26.4.1.0.0.57.6 uint8_t rtc_datetime_t::second

26.4.2 struct rtc_config_t

This structure holds the configuration settings for the RTC peripheral. To initialize this structure to reasonable defaults, call the [RTC_GetDefaultConfig\(\)](#) function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Data Fields

- bool [wakeupSelect](#)
true: Wakeup pin outputs the 32 KHz clock; false: Wakeup pin used to wakeup the chip
- bool [updateMode](#)
true: Registers can be written even when locked under certain conditions, false: No writes allowed when registers are locked
- bool [supervisorAccess](#)
true: Non-supervisor accesses are allowed; false: Non-supervisor accesses are not supported
- uint32_t [compensationInterval](#)
Compensation interval that is written to the CIR field in RTC TCR Register.
- uint32_t [compensationTime](#)
Compensation time that is written to the TCR field in RTC TCR Register.

26.5 Enumeration Type Documentation

26.5.1 enum rtc_interrupt_enable_t

Enumerator

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

kRTC_SecondsInterruptEnable Seconds interrupt.

26.5.2 enum rtc_status_flags_t

Enumerator

kRTC_TimeInvalidFlag Time invalid flag.

kRTC_TimeOverflowFlag Time overflow flag.

kRTC_AlarmFlag Alarm flag.

26.6 Function Documentation

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

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

Note

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

Parameters

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

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

Parameters

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

26.6.3 void RTC_GetDefaultConfig (rtc_config_t * *config*)

The default values are as follows.

```
* config->wakeupSelect = false;
* config->updateMode = false;
* config->supervisorAccess = false;
* config->compensationInterval = 0;
* config->compensationTime = 0;
*
```

Function Documentation

Parameters

<i>config</i>	Pointer to the user's RTC configuration structure.
---------------	--

26.6.4 **status_t RTC_SetDatetime (RTC_Type * *base*, const rtc_datetime_t * *datetime*)**

The RTC counter must be stopped prior to calling this function because writes to the RTC seconds register fail if the RTC counter is running.

Parameters

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

Returns

kStatus_Success: Success in setting the time and starting the RTC
kStatus_InvalidArgument: Error because the datetime format is incorrect

26.6.5 **void RTC_GetDatetime (RTC_Type * *base*, rtc_datetime_t * *datetime*)**

Parameters

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

26.6.6 **status_t RTC_SetAlarm (RTC_Type * *base*, const rtc_datetime_t * *alarmTime*)**

The function checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

<i>base</i>	RTC peripheral base address
<i>alarmTime</i>	Pointer to the structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the RTC alarm
 kStatus_InvalidArgument: Error because the alarm datetime format is incorrect
 kStatus_Fail: Error because the alarm time has already passed

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

Parameters

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

26.6.8 static void RTC_EnableInterrupts (RTC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	RTC peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

26.6.9 static void RTC_DisableInterrupts (RTC_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	RTC peripheral base address
<i>mask</i>	The interrupts to enable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

26.6.10 static uint32_t RTC_GetEnabledInterrupts (RTC_Type * *base*) [inline], [static]

Function Documentation

Parameters

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

Returns

The enabled interrupts. This is the logical OR of members of the enumeration [rtc_interrupt_enable_t](#)

26.6.11 **static uint32_t RTC_GetStatusFlags (RTC_Type * *base*) [inline], [static]**

Parameters

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

Returns

The status flags. This is the logical OR of members of the enumeration [rtc_status_flags_t](#)

26.6.12 **void RTC_ClearStatusFlags (RTC_Type * *base*, uint32_t *mask*)**

Parameters

<i>base</i>	RTC peripheral base address
<i>mask</i>	The status flags to clear. This is a logical OR of members of the enumeration rtc_status_flags_t

26.6.13 **static void RTC_StartTimer (RTC_Type * *base*) [inline], [static]**

After calling this function, the timer counter increments once a second provided SR[TOF] or SR[TIF] are not set.

Parameters

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

26.6.14 static void RTC_StopTimer (RTC_Type * *base*) [inline], [static]

RTC's seconds register can be written to only when the timer is stopped.

Parameters

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

26.6.15 static void RTC_Reset (RTC_Type * *base*) [inline], [static]

This resets all RTC registers except for the SWR bit and the RTC_WAR and RTC_RAR registers. The SWR bit is cleared by software explicitly clearing it.

Parameters

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

Chapter 27

SAI: Serial Audio Interface

27.1 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.

27.2 Typical use case

27.2.1 SAI Send/receive using an interrupt method

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

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

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

void main(void)
{
    //...
```

Typical use case

```
SAI_TxGetDefaultConfig(&user_config);

SAI_TxInit(SAI0, &user_config);
SAI_TransferTxCreateHandle(SAI0, &g_saiHandle, SAI_UserCallback, NULL);

//Configure sai format
SAI_TransferTxSetTransferFormat(SAI0, &g_saiHandle, mclkSource, mclk);

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

// Send out.
SAI_TransferSendNonBlocking(SAI0, &g_saiHandle, &sendXfer);

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

// ...
}
```

27.2.2 SAI Send/receive using a DMA method

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

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

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

void main(void)
{
    //...

    SAI_TxGetDefaultConfig(&user_config);
    SAI_TxInit(SAI0, &user_config);

    // Sets up the DMA.
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, SAI_TX_DMA_CHANNEL, SAI_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, SAI_TX_DMA_CHANNEL);

    DMA_Init(DMA0);

    /* Creates the DMA handle. */
    DMA_CreateHandle(&g_saiTxDmaHandle, DMA0, SAI_TX_DMA_CHANNEL);

    SAI_TransferTxCreateHandleDMA(SAI0, &g_saiTxDmaHandle, SAI_UserCallback,
        NULL);

    // Prepares to send.
```

```

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

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

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

// ...
}

```

Modules

- [SAI DMA Driver](#)
- [SAI eDMA Driver](#)

Data Structures

- struct [sai_config_t](#)
SAI user configuration structure. [More...](#)
- struct [sai_transfer_format_t](#)
sai transfer format [More...](#)
- struct [sai_transfer_t](#)
SAI transfer structure. [More...](#)
- struct [sai_handle_t](#)
SAI handle structure. [More...](#)

Macros

- #define [SAI_XFER_QUEUE_SIZE](#) (4)
SAI transfer queue size, user can refine it according to use case.

Typedefs

- typedef void(* [sai_transfer_callback_t](#))(I2S_Type *base, sai_handle_t *handle, status_t status, void *userData)
SAI transfer callback prototype.

Enumerations

- enum [_sai_status_t](#) {
[kStatus_SAI_TxBusy](#) = MAKE_STATUS(kStatusGroup_SAI, 0),
[kStatus_SAI_RxBusy](#) = MAKE_STATUS(kStatusGroup_SAI, 1),
[kStatus_SAI_TxError](#) = MAKE_STATUS(kStatusGroup_SAI, 2),
[kStatus_SAI_RxError](#) = MAKE_STATUS(kStatusGroup_SAI, 3),
[kStatus_SAI_QueueFull](#) = MAKE_STATUS(kStatusGroup_SAI, 4),
[kStatus_SAI_TxIdle](#) = MAKE_STATUS(kStatusGroup_SAI, 5),
[kStatus_SAI_RxIdle](#) = MAKE_STATUS(kStatusGroup_SAI, 6) }

Typical use case

- SAI return status.*
 - enum `sai_protocol_t` {
 `kSAI_BusLeftJustified` = 0x0U,
 `kSAI_BusRightJustified`,
 `kSAI_BusI2S`,
 `kSAI_BusPCMA`,
 `kSAI_BusPCMB` }
- Define the SAI bus type.*
 - enum `sai_master_slave_t` {
 `kSAI_Master` = 0x0U,
 `kSAI_Slave` = 0x1U }
- Master or slave mode.*
 - enum `sai_mono_stereo_t` {
 `kSAI_Stereo` = 0x0U,
 `kSAI_MonoLeft`,
 `kSAI_MonoRight` }
- Mono or stereo audio format.*
 - enum `sai_sync_mode_t` {
 `kSAI_ModeAsync` = 0x0U,
 `kSAI_ModeSync`,
 `kSAI_ModeSyncWithOtherTx`,
 `kSAI_ModeSyncWithOtherRx` }
- Synchronous or asynchronous mode.*
 - enum `sai_mclk_source_t` {
 `kSAI_MclkSourceSysclk` = 0x0U,
 `kSAI_MclkSourceSelect1`,
 `kSAI_MclkSourceSelect2`,
 `kSAI_MclkSourceSelect3` }
- Master clock source.*
 - enum `sai_bclk_source_t` {
 `kSAI_BclkSourceBusclk` = 0x0U,
 `kSAI_BclkSourceMclkDiv`,
 `kSAI_BclkSourceOtherSai0`,
 `kSAI_BclkSourceOtherSai1` }
- Bit clock source.*
 - enum `_sai_interrupt_enable_t` {
 `kSAI_WordStartInterruptEnable`,
 `kSAI_SyncErrorInterruptEnable` = I2S_TCSR_SEIE_MASK,
 `kSAI_FIFOWarningInterruptEnable` = I2S_TCSR_FWIE_MASK,
 `kSAI_FIFOErrorInterruptEnable` = I2S_TCSR_FEIE_MASK }
- The SAI interrupt enable flag.*
 - enum `_sai_dma_enable_t` { `kSAI_FIFOWarningDMAEnable` = I2S_TCSR_FWDE_MASK }
- The DMA request sources.*
 - enum `_sai_flags` {
 `kSAI_WordStartFlag` = I2S_TCSR_WSF_MASK,
 `kSAI_SyncErrorFlag` = I2S_TCSR_SEF_MASK,
 `kSAI_FIFOErrorFlag` = I2S_TCSR_FEF_MASK,

```
kSAI_FIFOWarningFlag = I2S_TCSR_FWF_MASK }
```

The SAI status flag.

- enum `sai_reset_type_t` {
`kSAI_ResetTypeSoftware` = I2S_TCSR_SR_MASK,
`kSAI_ResetTypeFIFO` = I2S_TCSR_FR_MASK,
`kSAI_ResetAll` = I2S_TCSR_SR_MASK | I2S_TCSR_FR_MASK }

The reset type.

- enum `sai_sample_rate_t` {
`kSAI_SampleRate8KHz` = 8000U,
`kSAI_SampleRate11025Hz` = 11025U,
`kSAI_SampleRate12KHz` = 12000U,
`kSAI_SampleRate16KHz` = 16000U,
`kSAI_SampleRate22050Hz` = 22050U,
`kSAI_SampleRate24KHz` = 24000U,
`kSAI_SampleRate32KHz` = 32000U,
`kSAI_SampleRate44100Hz` = 44100U,
`kSAI_SampleRate48KHz` = 48000U,
`kSAI_SampleRate96KHz` = 96000U }

Audio sample rate.

- enum `sai_word_width_t` {
`kSAI_WordWidth8bits` = 8U,
`kSAI_WordWidth16bits` = 16U,
`kSAI_WordWidth24bits` = 24U,
`kSAI_WordWidth32bits` = 32U }

Audio word width.

Driver version

- #define `FSL_SAI_DRIVER_VERSION` (MAKE_VERSION(2, 1, 2))
Version 2.1.2.

Initialization and deinitialization

- void `SAI_TxInit` (I2S_Type *base, const `sai_config_t` *config)
Initializes the SAI Tx peripheral.
- void `SAI_RxInit` (I2S_Type *base, const `sai_config_t` *config)
Initializes the the SAI Rx peripheral.
- void `SAI_TxGetDefaultConfig` (`sai_config_t` *config)
Sets the SAI Tx configuration structure to default values.
- void `SAI_RxGetDefaultConfig` (`sai_config_t` *config)
Sets the SAI Rx configuration structure to default values.
- void `SAI_Deinit` (I2S_Type *base)
De-initializes the SAI peripheral.
- void `SAI_TxReset` (I2S_Type *base)
Resets the SAI Tx.
- void `SAI_RxReset` (I2S_Type *base)
Resets the SAI Rx.
- void `SAI_TxEnable` (I2S_Type *base, bool enable)

Typical use case

- *Enables/disables the SAI Tx.*
void [SAI_RxEnable](#) (I2S_Type *base, bool enable)
Enables/disables the SAI Rx.

Status

- static uint32_t [SAI_TxGetStatusFlag](#) (I2S_Type *base)
Gets the SAI Tx status flag state.
- static void [SAI_TxClearStatusFlags](#) (I2S_Type *base, uint32_t mask)
Clears the SAI Tx status flag state.
- static uint32_t [SAI_RxGetStatusFlag](#) (I2S_Type *base)
Gets the SAI Rx status flag state.
- static void [SAI_RxClearStatusFlags](#) (I2S_Type *base, uint32_t mask)
Clears the SAI Rx status flag state.

Interrupts

- static void [SAI_TxEnableInterrupts](#) (I2S_Type *base, uint32_t mask)
Enables the SAI Tx interrupt requests.
- static void [SAI_RxEnableInterrupts](#) (I2S_Type *base, uint32_t mask)
Enables the SAI Rx interrupt requests.
- static void [SAI_TxDisableInterrupts](#) (I2S_Type *base, uint32_t mask)
Disables the SAI Tx interrupt requests.
- static void [SAI_RxDisableInterrupts](#) (I2S_Type *base, uint32_t mask)
Disables the SAI Rx interrupt requests.

DMA Control

- static void [SAI_TxEnableDMA](#) (I2S_Type *base, uint32_t mask, bool enable)
Enables/disables the SAI Tx DMA requests.
- static void [SAI_RxEnableDMA](#) (I2S_Type *base, uint32_t mask, bool enable)
Enables/disables the SAI Rx DMA requests.
- static uint32_t [SAI_TxGetDataRegisterAddress](#) (I2S_Type *base, uint32_t channel)
Gets the SAI Tx data register address.
- static uint32_t [SAI_RxGetDataRegisterAddress](#) (I2S_Type *base, uint32_t channel)
Gets the SAI Rx data register address.

Bus Operations

- void [SAI_TxSetFormat](#) (I2S_Type *base, [sai_transfer_format_t](#) *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- void [SAI_RxSetFormat](#) (I2S_Type *base, [sai_transfer_format_t](#) *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- void [SAI_WriteBlocking](#) (I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Sends data using a blocking method.
- static void [SAI_WriteData](#) (I2S_Type *base, uint32_t channel, uint32_t data)
Writes data into SAI FIFO.

- void [SAI_ReadBlocking](#) (I2S_Type *base, uint32_t channel, uint32_t bitWidth, uint8_t *buffer, uint32_t size)
Receives data using a blocking method.
- static uint32_t [SAI_ReadData](#) (I2S_Type *base, uint32_t channel)
Reads data from the SAI FIFO.

Transactional

- void [SAI_TransferTxCreateHandle](#) (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
Initializes the SAI Tx handle.
- void [SAI_TransferRxCreateHandle](#) (I2S_Type *base, sai_handle_t *handle, sai_transfer_callback_t callback, void *userData)
Initializes the SAI Rx handle.
- status_t [SAI_TransferTxSetFormat](#) (I2S_Type *base, sai_handle_t *handle, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- status_t [SAI_TransferRxSetFormat](#) (I2S_Type *base, sai_handle_t *handle, sai_transfer_format_t *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- status_t [SAI_TransferSendNonBlocking](#) (I2S_Type *base, sai_handle_t *handle, sai_transfer_t *xfer)
Performs an interrupt non-blocking send transfer on SAI.
- status_t [SAI_TransferReceiveNonBlocking](#) (I2S_Type *base, sai_handle_t *handle, sai_transfer_t *xfer)
Performs an interrupt non-blocking receive transfer on SAI.
- status_t [SAI_TransferGetSendCount](#) (I2S_Type *base, sai_handle_t *handle, size_t *count)
Gets a set byte count.
- status_t [SAI_TransferGetReceiveCount](#) (I2S_Type *base, sai_handle_t *handle, size_t *count)
Gets a received byte count.
- void [SAI_TransferAbortSend](#) (I2S_Type *base, sai_handle_t *handle)
Aborts the current send.
- void [SAI_TransferAbortReceive](#) (I2S_Type *base, sai_handle_t *handle)
Aborts the the current IRQ receive.
- void [SAI_TransferTxHandleIRQ](#) (I2S_Type *base, sai_handle_t *handle)
Tx interrupt handler.
- void [SAI_TransferRxHandleIRQ](#) (I2S_Type *base, sai_handle_t *handle)
Tx interrupt handler.

27.3 Data Structure Documentation

27.3.1 struct sai_config_t

Data Fields

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

Data Structure Documentation

- [sai_mclk_source_t mclkSource](#)
Master Clock source.
- [sai_bclk_source_t bclkSource](#)
Bit Clock source.
- [sai_master_slave_t masterSlave](#)
Master or slave.

27.3.2 struct sai_transfer_format_t

Data Fields

- [uint32_t sampleRate_Hz](#)
Sample rate of audio data.
- [uint32_t bitWidth](#)
Data length of audio data, usually 8/16/24/32 bits.
- [sai_mono_stereo_t stereo](#)
Mono or stereo.
- [uint32_t masterClockHz](#)
Master clock frequency in Hz.
- [uint8_t channel](#)
Data channel used in transfer.
- [sai_protocol_t protocol](#)
Which audio protocol used.

27.3.2.0.0.58 Field Documentation

27.3.2.0.0.58.1 [uint8_t sai_transfer_format_t::channel](#)

27.3.3 struct sai_transfer_t

Data Fields

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

27.3.3.0.0.59 Field Documentation

27.3.3.0.0.59.1 `uint8_t* sai_transfer_t::data`

27.3.3.0.0.59.2 `size_t sai_transfer_t::dataSize`

27.3.4 struct _sai_handle

Data Fields

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

27.4 Macro Definition Documentation

27.4.1 `#define SAI_XFER_QUEUE_SIZE (4)`

27.5 Enumeration Type Documentation

27.5.1 `enum _sai_status_t`

Enumerator

kStatus_SAI_TxBusy SAI Tx is busy.
kStatus_SAI_RxBusy SAI Rx is busy.
kStatus_SAI_TxError SAI Tx FIFO error.
kStatus_SAI_RxError SAI Rx FIFO error.
kStatus_SAI_QueueFull SAI transfer queue is full.
kStatus_SAI_TxIdle SAI Tx is idle.
kStatus_SAI_RxIdle SAI Rx is idle.

27.5.2 enum sai_protocol_t

Enumerator

kSAI_BusLeftJustified Uses left justified format.
kSAI_BusRightJustified Uses right justified format.
kSAI_BusI2S Uses I2S format.
kSAI_BusPCMA Uses I2S PCM A format.
kSAI_BusPCMB Uses I2S PCM B format.

27.5.3 enum sai_master_slave_t

Enumerator

kSAI_Master Master mode.
kSAI_Slave Slave mode.

27.5.4 enum sai_mono_stereo_t

Enumerator

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

27.5.5 enum sai_sync_mode_t

Enumerator

kSAI_ModeAsync Asynchronous mode.
kSAI_ModeSync Synchronous mode (with receiver or transmit)
kSAI_ModeSyncWithOtherTx Synchronous with another SAI transmit.
kSAI_ModeSyncWithOtherRx Synchronous with another SAI receiver.

27.5.6 enum sai_mclk_source_t

Enumerator

kSAI_MclkSourceSysclk Master clock from the system clock.
kSAI_MclkSourceSelect1 Master clock from source 1.
kSAI_MclkSourceSelect2 Master clock from source 2.
kSAI_MclkSourceSelect3 Master clock from source 3.

27.5.7 enum sai_bclk_source_t

Enumerator

- kSAI_BclkSourceBusclk* Bit clock using bus clock.
- kSAI_BclkSourceMclkDiv* Bit clock using master clock divider.
- kSAI_BclkSourceOtherSai0* Bit clock from other SAI device.
- kSAI_BclkSourceOtherSai1* Bit clock from other SAI device.

27.5.8 enum _sai_interrupt_enable_t

Enumerator

- kSAI_WordStartInterruptEnable* Word start flag, means the first word in a frame detected.
- kSAI_SyncErrorInterruptEnable* Sync error flag, means the sync error is detected.
- kSAI_FIFOWarningInterruptEnable* FIFO warning flag, means the FIFO is empty.
- kSAI_FIFOErrorInterruptEnable* FIFO error flag.

27.5.9 enum _sai_dma_enable_t

Enumerator

- kSAI_FIFOWarningDMAEnable* FIFO warning caused by the DMA request.

27.5.10 enum _sai_flags

Enumerator

- kSAI_WordStartFlag* Word start flag, means the first word in a frame detected.
- kSAI_SyncErrorFlag* Sync error flag, means the sync error is detected.
- kSAI_FIFOErrorFlag* FIFO error flag.
- kSAI_FIFOWarningFlag* FIFO warning flag.

27.5.11 enum sai_reset_type_t

Enumerator

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

Function Documentation

27.5.12 enum sai_sample_rate_t

Enumerator

kSAI_SampleRate8KHz Sample rate 8000 Hz.
kSAI_SampleRate11025Hz Sample rate 11025 Hz.
kSAI_SampleRate12KHz Sample rate 12000 Hz.
kSAI_SampleRate16KHz Sample rate 16000 Hz.
kSAI_SampleRate22050Hz Sample rate 22050 Hz.
kSAI_SampleRate24KHz Sample rate 24000 Hz.
kSAI_SampleRate32KHz Sample rate 32000 Hz.
kSAI_SampleRate44100Hz Sample rate 44100 Hz.
kSAI_SampleRate48KHz Sample rate 48000 Hz.
kSAI_SampleRate96KHz Sample rate 96000 Hz.

27.5.13 enum sai_word_width_t

Enumerator

kSAI_WordWidth8bits Audio data width 8 bits.
kSAI_WordWidth16bits Audio data width 16 bits.
kSAI_WordWidth24bits Audio data width 24 bits.
kSAI_WordWidth32bits Audio data width 32 bits.

27.6 Function Documentation

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

Un-gates the SAI clock, resets the module, and configures SAI Tx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI_TxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAIM module can cause a hard fault because the clock is not enabled.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

<i>config</i>	SAI configuration structure.
---------------	------------------------------

27.6.2 void SAI_RxInit (I2S_Type * *base*, const sai_config_t * *config*)

Ungates the SAI clock, resets the module, and configures the SAI Rx with a configuration structure. The configuration structure can be custom filled or set with default values by [SAI_RxGetDefaultConfig\(\)](#).

Note

This API should be called at the beginning of the application to use the SAI driver. Otherwise, accessing the SAI module can cause a hard fault because the clock is not enabled.

Parameters

<i>base</i>	SAI base pointer
<i>config</i>	SAI configuration structure.

27.6.3 void SAI_TxGetDefaultConfig (sai_config_t * *config*)

This API initializes the configuration structure for use in SAI_TxConfig(). The initialized structure can remain unchanged in SAI_TxConfig(), or it can be modified before calling SAI_TxConfig(). This is an example.

```
sai_config_t config;
SAI_TxGetDefaultConfig(&config);
```

Parameters

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

27.6.4 void SAI_RxGetDefaultConfig (sai_config_t * *config*)

This API initializes the configuration structure for use in SAI_RxConfig(). The initialized structure can remain unchanged in SAI_RxConfig() or it can be modified before calling SAI_RxConfig(). This is an example.

```
sai_config_t config;
SAI_RxGetDefaultConfig(&config);
```

Function Documentation

Parameters

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

27.6.5 void SAI_Deinit (I2S_Type * *base*)

This API gates the SAI clock. The SAI module can't operate unless SAI_TxInit or SAI_RxInit is called to enable the clock.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

27.6.6 void SAI_TxReset (I2S_Type * *base*)

This function enables the software reset and FIFO reset of SAI Tx. After reset, clear the reset bit.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

27.6.7 void SAI_RxReset (I2S_Type * *base*)

This function enables the software reset and FIFO reset of SAI Rx. After reset, clear the reset bit.

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

27.6.8 void SAI_TxEnable (I2S_Type * *base*, bool *enable*)

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

<i>enable</i>	True means enable SAI Tx, false means disable.
---------------	--

27.6.9 void SAI_RxEnable (I2S_Type * *base*, bool *enable*)

Parameters

<i>base</i>	SAI base pointer
<i>enable</i>	True means enable SAI Rx, false means disable.

27.6.10 static uint32_t SAI_TxGetStatusFlag (I2S_Type * *base*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Returns

SAI Tx status flag value. Use the Status Mask to get the status value needed.

27.6.11 static void SAI_TxClearStatusFlags (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	State mask. It can be a combination of the following source if defined: <ul style="list-style-type: none"> • kSAI_WordStartFlag • kSAI_SyncErrorFlag • kSAI_FIFOErrorFlag

27.6.12 static uint32_t SAI_RxGetStatusFlag (I2S_Type * *base*) [inline], [static]

Function Documentation

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

Returns

SAI Rx status flag value. Use the Status Mask to get the status value needed.

27.6.13 static void SAI_RxClearStatusFlags (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	State mask. It can be a combination of the following sources if defined. <ul style="list-style-type: none">• kSAI_WordStartFlag• kSAI_SyncErrorFlag• kSAI_FIFOErrorFlag

27.6.14 static void SAI_TxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none">• kSAI_WordStartInterruptEnable• kSAI_SyncErrorInterruptEnable• kSAI_FIFOWarningInterruptEnable• kSAI_FIFORequestInterruptEnable• kSAI_FIFOErrorInterruptEnable

27.6.15 static void SAI_RxEnableInterrupts (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

27.6.16 static void SAI_TxDisableInterrupts (l2s_Type * *base*, uint32_t *mask*)
[inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none"> • kSAI_WordStartInterruptEnable • kSAI_SyncErrorInterruptEnable • kSAI_FIFOWarningInterruptEnable • kSAI_FIFORequestInterruptEnable • kSAI_FIFOErrorInterruptEnable

Function Documentation

27.6.17 static void SAI_RxDisableInterrupts (I2S_Type * *base*, uint32_t *mask*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	interrupt source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none">• kSAI_WordStartInterruptEnable• kSAI_SyncErrorInterruptEnable• kSAI_FIFOWarningInterruptEnable• kSAI_FIFORequestInterruptEnable• kSAI_FIFOErrorInterruptEnable

27.6.18 static void SAI_TxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	DMA source The parameter can be combination of the following sources if defined. <ul style="list-style-type: none">• kSAI_FIFOWarningDMAEnable• kSAI_FIFORequestDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

27.6.19 static void SAI_RxEnableDMA (I2S_Type * *base*, uint32_t *mask*, bool *enable*) [inline], [static]

Parameters

<i>base</i>	SAI base pointer
<i>mask</i>	DMA source The parameter can be a combination of the following sources if defined. <ul style="list-style-type: none">• kSAI_FIFOWarningDMAEnable• kSAI_FIFORequestDMAEnable
<i>enable</i>	True means enable DMA, false means disable DMA.

27.6.20 `static uint32_t SAI_TxGetDataRegisterAddress (I2S_Type * base, uint32_t channel) [inline], [static]`

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

Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

27.6.21 static uint32_t SAI_RxGetDataRegisterAddress (I2S_Type * *base*, uint32_t *channel*) [inline], [static]

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Which data channel used.

Returns

data register address.

27.6.22 void SAI_TxSetFormat (I2S_Type * *base*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.

<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.
---------------------------	---

27.6.23 void SAI_RxSetFormat (I2S_Type * *base*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If the bit clock source is a master clock, this value should equal the masterClockHz.

27.6.24 void SAI_WriteBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be written.
<i>size</i>	Bytes to be written.

27.6.25 static void SAI_WriteData (I2S_Type * *base*, uint32_t *channel*, uint32_t *data*) [inline], [static]

Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>data</i>	Data needs to be written.

27.6.26 void SAI_ReadBlocking (I2S_Type * *base*, uint32_t *channel*, uint32_t *bitWidth*, uint8_t * *buffer*, uint32_t *size*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.
<i>bitWidth</i>	How many bits in an audio word; usually 8/16/24/32 bits.
<i>buffer</i>	Pointer to the data to be read.
<i>size</i>	Bytes to be read.

**27.6.27 static uint32_t SAI_ReadData (I2S_Type * *base*, uint32_t *channel*)
[inline], [static]**

Parameters

<i>base</i>	SAI base pointer.
<i>channel</i>	Data channel used.

Returns

Data in SAI FIFO.

27.6.28 void SAI_TransferTxCreateHandle (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_callback_t *callback*, void * *userData*)

This function initializes the Tx handle for the SAI Tx transactional APIs. Call this function once to get the handle initialized.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI handle pointer.
<i>callback</i>	Pointer to the user callback function.
<i>userData</i>	User parameter passed to the callback function

27.6.29 void SAI_TransferRxCreateHandle (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_callback_t *callback*, void * *userData*)

This function initializes the Rx handle for the SAI Rx transactional APIs. Call this function once to get the handle initialized.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>callback</i>	Pointer to the user callback function.
<i>userData</i>	User parameter passed to the callback function.

27.6.30 status_t SAI_TransferTxSetFormat (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.

Function Documentation

<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.
---------------------------	---

Returns

Status of this function. Return value is the status_t.

27.6.31 status_t SAI_TransferRxSetFormat (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI handle pointer.
<i>format</i>	Pointer to the SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is a master clock, this value should equal the masterClockHz in format.

Returns

Status of this function. Return value is one of status_t.

27.6.32 status_t SAI_TransferSendNonBlocking (I2S_Type * *base*, sai_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This API returns immediately after the transfer initiates. Call the SAI_TxGetTransferStatusIRQ to poll the transfer status and check whether the transfer is finished. If the return status is not kStatus_-SAI_Busy, the transfer is finished.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the <code>sai_handle_t</code> structure which stores the transfer state.
<i>xfer</i>	Pointer to the <code>sai_transfer_t</code> structure.

Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_TxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

27.6.33 `status_t SAI_TransferReceiveNonBlocking (I2S_Type * base, sai_handle_t * handle, sai_transfer_t * xfer)`

Note

This API returns immediately after the transfer initiates. Call the `SAI_RxGetTransferStatusIRQ` to poll the transfer status and check whether the transfer is finished. If the return status is not `kStatus_SAI_Busy`, the transfer is finished.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	Pointer to the <code>sai_handle_t</code> structure which stores the transfer state.
<i>xfer</i>	Pointer to the <code>sai_transfer_t</code> structure.

Return values

<i>kStatus_Success</i>	Successfully started the data receive.
<i>kStatus_SAI_RxBusy</i>	Previous receive still not finished.
<i>kStatus_InvalidArgument</i>	The input parameter is invalid.

27.6.34 `status_t SAI_TransferGetSendCount (I2S_Type * base, sai_handle_t * handle, size_t * count)`

Function Documentation

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count sent.

Return values

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

27.6.35 status_t SAI_TransferGetReceiveCount (I2S_Type * *base*, sai_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.
<i>count</i>	Bytes count received.

Return values

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

27.6.36 void SAI_TransferAbortSend (I2S_Type * *base*, sai_handle_t * *handle*)

Note

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.

27.6.37 void SAI_TransferAbortReceive (I2S_Type * *base*, sai_handle_t * *handle*)

Note

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

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	Pointer to the sai_handle_t structure which stores the transfer state.

27.6.38 void SAI_TransferTxHandleIRQ (I2S_Type * *base*, sai_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure.

27.6.39 void SAI_TransferRxHandleIRQ (I2S_Type * *base*, sai_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	Pointer to the sai_handle_t structure.

27.7 SAI DMA Driver

27.7.1 Overview

Data Structures

- struct [sai_dma_handle_t](#)
SAI DMA transfer handle, users should not touch the content of the handle. [More...](#)

Typedefs

- typedef void(* [sai_dma_callback_t](#))(I2S_Type *base, sai_dma_handle_t *handle, status_t status, void *userData)
Define SAI DMA callback.

DMA Transactional

- void [SAI_TransferTxCreateHandleDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, [sai_dma_callback_t](#) callback, void *userData, dma_handle_t *dmaHandle)
Initializes the SAI master DMA handle.
- void [SAI_TransferRxCreateHandleDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, [sai_dma_callback_t](#) callback, void *userData, dma_handle_t *dmaHandle)
Initializes the SAI slave DMA handle.
- void [SAI_TransferTxSetFormatDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, [sai_transfer_format_t](#) *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- void [SAI_TransferRxSetFormatDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, [sai_transfer_format_t](#) *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- status_t [SAI_TransferSendDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, [sai_transfer_t](#) *xfer)
Performs a non-blocking SAI transfer using DMA.
- status_t [SAI_TransferReceiveDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, [sai_transfer_t](#) *xfer)
Performs a non-blocking SAI transfer using DMA.
- void [SAI_TransferAbortSendDMA](#) (I2S_Type *base, sai_dma_handle_t *handle)
Aborts a SAI transfer using DMA.
- void [SAI_TransferAbortReceiveDMA](#) (I2S_Type *base, sai_dma_handle_t *handle)
Aborts a SAI transfer using DMA.
- status_t [SAI_TransferGetSendCountDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, size_t *count)
Gets byte count sent by SAI.
- status_t [SAI_TransferGetReceiveCountDMA](#) (I2S_Type *base, sai_dma_handle_t *handle, size_t *count)
Gets byte count received by SAI.

27.7.2 Data Structure Documentation

27.7.2.1 struct _sai_dma_handle

Data Fields

- dma_handle_t * [dmaHandle](#)
DMA handler for SAI send.
- uint8_t [bytesPerFrame](#)
Bytes in a frame.
- uint8_t [channel](#)
Which Data channel SAI use.
- uint32_t [state](#)
SAI DMA transfer internal state.
- [sai_dma_callback_t](#) [callback](#)
Callback for users while transfer finish or error occurred.
- void * [userData](#)
User callback parameter.
- [sai_transfer_t](#) [saiQueue](#) [SAI_XFER_QUEUE_SIZE]
Transfer queue storing queued transfer.
- size_t [transferSize](#) [SAI_XFER_QUEUE_SIZE]
Data bytes need to transfer.
- volatile uint8_t [queueUser](#)
Index for user to queue transfer.
- volatile uint8_t [queueDriver](#)
Index for driver to get the transfer data and size.

27.7.2.1.0.60 Field Documentation

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

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

27.7.3 Function Documentation

27.7.3.1 **void SAI_TransferTxCreateHandleDMA (I2S_Type * *base*, [sai_dma_handle_t](#) * *handle*, [sai_dma_callback_t](#) *callback*, void * *userData*, dma_handle_t * *dmaHandle*)**

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

Parameters

SAI DMA Driver

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	DMA handle pointer, this handle shall be static allocated by users.

27.7.3.2 void SAI_TransferRxCreateHandleDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, sai_dma_callback_t *callback*, void * *userData*, dma_handle_t * *dmaHandle*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	DMA handle pointer, this handle shall be static allocated by users.

27.7.3.3 void SAI_TransferTxSetFormatDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to the format.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.

<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If bit clock source is master. clock, this value should equals to masterClockHz in format.

Return values

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

27.7.3.4 void SAI_TransferRxSetFormatDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets eDMA parameter according to format.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If bit clock source is master. clock, this value should equals to masterClockHz in format.

Return values

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

27.7.3.5 status_t SAI_TransferSendDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This interface returns immediately after the transfer initiates. Call the SAI_GetTransferStatus to poll the transfer status to check whether the SAI transfer finished.

SAI DMA Driver

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

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

27.7.3.6 **status_t SAI_TransferReceiveDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, sai_transfer_t * *xfer*)**

Note

This interface returns immediately after transfer initiates. Call SAI_GetTransferStatus to poll the transfer status to check whether the SAI transfer is finished.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI DMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

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

27.7.3.7 **void SAI_TransferAbortSendDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*)**

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.

27.7.3.8 void SAI_TransferAbortReceiveDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.

27.7.3.9 status_t SAI_TransferGetSendCountDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI DMA handle pointer.
<i>count</i>	Bytes count sent by SAI.

Return values

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

27.7.3.10 status_t SAI_TransferGetReceiveCountDMA (I2S_Type * *base*, sai_dma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
-------------	-------------------

SAI DMA Driver

<i>handle</i>	SAI DMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

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

27.8 SAI eDMA Driver

27.8.1 Overview

Data Structures

- struct [sai_edma_handle_t](#)
SAI DMA transfer handle, users should not touch the content of the handle. [More...](#)

Typedefs

- typedef void(* [sai_edma_callback_t](#))(I2S_Type *base, sai_edma_handle_t *handle, status_t status, void *userData)
SAI eDMA transfer callback function for finish and error.

eDMA Transactional

- void [SAI_TransferTxCreateHandleEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, [sai_edma_callback_t](#) callback, void *userData, [edma_handle_t](#) *dmaHandle)
Initializes the SAI eDMA handle.
- void [SAI_TransferRxCreateHandleEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, [sai_edma_callback_t](#) callback, void *userData, [edma_handle_t](#) *dmaHandle)
Initializes the SAI Rx eDMA handle.
- void [SAI_TransferTxSetFormatEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, [sai_transfer_format_t](#) *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Tx audio format.
- void [SAI_TransferRxSetFormatEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, [sai_transfer_format_t](#) *format, uint32_t mclkSourceClockHz, uint32_t bclkSourceClockHz)
Configures the SAI Rx audio format.
- status_t [SAI_TransferSendEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, [sai_transfer_t](#) *xfer)
Performs a non-blocking SAI transfer using DMA.
- status_t [SAI_TransferReceiveEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, [sai_transfer_t](#) *xfer)
Performs a non-blocking SAI receive using eDMA.
- void [SAI_TransferAbortSendEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle)
Aborts a SAI transfer using eDMA.
- void [SAI_TransferAbortReceiveEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle)
Aborts a SAI receive using eDMA.
- status_t [SAI_TransferGetSendCountEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, size_t *count)
Gets byte count sent by SAI.
- status_t [SAI_TransferGetReceiveCountEDMA](#) (I2S_Type *base, sai_edma_handle_t *handle, size_t *count)
Gets byte count received by SAI.

27.8.2 Data Structure Documentation

27.8.2.1 struct _sai_edma_handle

Data Fields

- [edma_handle_t](#) * [dmaHandle](#)
DMA handler for SAI send.
- [uint8_t](#) [nbytes](#)
eDMA minor byte transfer count initially configured.
- [uint8_t](#) [bytesPerFrame](#)
Bytes in a frame.
- [uint8_t](#) [channel](#)
Which data channel.
- [uint8_t](#) [count](#)
The transfer data count in a DMA request.
- [uint32_t](#) [state](#)
Internal state for SAI eDMA transfer.
- [sai_edma_callback_t](#) [callback](#)
Callback for users while transfer finish or error occurs.
- [void](#) * [userData](#)
User callback parameter.
- [edma_tcd_t](#) [tcd](#) [[SAI_XFER_QUEUE_SIZE](#)+1U]
TCD pool for eDMA transfer.
- [sai_transfer_t](#) [saiQueue](#) [[SAI_XFER_QUEUE_SIZE](#)]
Transfer queue storing queued transfer.
- [size_t](#) [transferSize](#) [[SAI_XFER_QUEUE_SIZE](#)]
Data bytes need to transfer.
- [volatile uint8_t](#) [queueUser](#)
Index for user to queue transfer.
- [volatile uint8_t](#) [queueDriver](#)
Index for driver to get the transfer data and size.

27.8.2.1.0.61 Field Documentation**27.8.2.1.0.61.1** `uint8_t sai_edma_handle_t::nbytes`**27.8.2.1.0.61.2** `edma_tcd_t sai_edma_handle_t::tcd[SAI_XFER_QUEUE_SIZE+1U]`**27.8.2.1.0.61.3** `sai_transfer_t sai_edma_handle_t::saiQueue[SAI_XFER_QUEUE_SIZE]`**27.8.2.1.0.61.4** `volatile uint8_t sai_edma_handle_t::queueUser`**27.8.3 Function Documentation**

27.8.3.1 `void SAI_TransferTxCreateHandleEDMA (I2S_Type * base, sai_edma_handle_t * handle, sai_edma_callback_t callback, void * userData, edma_handle_t * dmaHandle)`

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

SAI eDMA Driver

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	eDMA handle pointer, this handle shall be static allocated by users.

27.8.3.2 void SAI_TransferRxCreateHandleEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_edma_callback_t *callback*, void * *userData*, edma_handle_t * *dmaHandle*)

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

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>base</i>	SAI peripheral base address.
<i>callback</i>	Pointer to user callback function.
<i>userData</i>	User parameter passed to the callback function.
<i>dmaHandle</i>	eDMA handle pointer, this handle shall be static allocated by users.

27.8.3.3 void SAI_TransferTxSetFormatEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *blkSourceClockHz*)

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.

27.8.3.4 void SAI_TransferRxSetFormatEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_format_t * *format*, uint32_t *mclkSourceClockHz*, uint32_t *bclkSourceClockHz*)

The audio format can be changed at run-time. This function configures the sample rate and audio data format to be transferred. This function also sets the eDMA parameter according to formatting requirements.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>format</i>	Pointer to SAI audio data format structure.
<i>mclkSource-ClockHz</i>	SAI master clock source frequency in Hz.
<i>bclkSource-ClockHz</i>	SAI bit clock source frequency in Hz. If a bit clock source is the master clock, this value should equal to masterClockHz in format.

Return values

<i>kStatus_Success</i>	Audio format set successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.

27.8.3.5 status_t SAI_TransferSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_t * *xfer*)

SAI eDMA Driver

Note

This interface returns immediately after the transfer initiates. Call SAI_GetTransferStatus to poll the transfer status and check whether the SAI transfer is finished.

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to the DMA transfer structure.

Return values

<i>kStatus_Success</i>	Start a SAI eDMA send successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_TxBusy</i>	SAI is busy sending data.

27.8.3.6 status_t SAI_TransferReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, sai_transfer_t * *xfer*)

Note

This interface returns immediately after the transfer initiates. Call the SAI_GetReceiveRemaining-Bytes to poll the transfer status and check whether the SAI transfer is finished.

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.
<i>xfer</i>	Pointer to DMA transfer structure.

Return values

<i>kStatus_Success</i>	Start a SAI eDMA receive successfully.
<i>kStatus_InvalidArgument</i>	The input argument is invalid.
<i>kStatus_RxBusy</i>	SAI is busy receiving data.

27.8.3.7 void SAI_TransferAbortSendEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.

27.8.3.8 void SAI_TransferAbortReceiveEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*)

Parameters

<i>base</i>	SAI base pointer
<i>handle</i>	SAI eDMA handle pointer.

27.8.3.9 status_t SAI_TransferGetSendCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer.
<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count sent by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is no non-blocking transaction in progress.

27.8.3.10 status_t SAI_TransferGetReceiveCountEDMA (I2S_Type * *base*, sai_edma_handle_t * *handle*, size_t * *count*)

Parameters

<i>base</i>	SAI base pointer
-------------	------------------

SAI eDMA Driver

<i>handle</i>	SAI eDMA handle pointer.
<i>count</i>	Bytes count received by SAI.

Return values

<i>kStatus_Success</i>	Succeed get the transfer count.
<i>kStatus_NoTransferInProgress</i>	There is no non-blocking transaction in progress.

Chapter 28

SIM: System Integration Module Driver

28.1 Overview

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

Data Structures

- struct [sim_uid_t](#)
Unique ID. [More...](#)

Enumerations

- enum [_sim_flash_mode](#) {
[kSIM_FlashDisableInWait](#) = SIM_FCFG1_FLASHDOZE_MASK,
[kSIM_FlashDisable](#) = SIM_FCFG1_FLASHDIS_MASK }
Flash enable mode.

Functions

- void [SIM_GetUniqueId](#) ([sim_uid_t](#) *uid)
Gets the unique identification register value.
- static void [SIM_SetFlashMode](#) (uint8_t mode)
Sets the flash enable mode.

Driver version

- #define [FSL_SIM_DRIVER_VERSION](#) ([MAKE_VERSION](#)(2, 0, 0))
Driver version 2.0.0.

28.2 Data Structure Documentation

28.2.1 struct sim_uid_t

Data Fields

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

Function Documentation

28.2.1.0.0.62 Field Documentation

28.2.1.0.0.62.1 uint32_t sim_uid_t::MH

28.2.1.0.0.62.2 uint32_t sim_uid_t::ML

28.2.1.0.0.62.3 uint32_t sim_uid_t::L

28.3 Enumeration Type Documentation

28.3.1 enum _sim_flash_mode

Enumerator

kSIM_FlashDisableInWait Disable flash in wait mode.

kSIM_FlashDisable Disable flash in normal mode.

28.4 Function Documentation

28.4.1 void SIM_GetUniqueld (sim_uid_t * uid)

Parameters

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

28.4.2 static void SIM_SetFlashMode (uint8_t mode) [inline], [static]

Parameters

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

Chapter 29

SMC: System Mode Controller Driver

29.1 Overview

The MCUXpresso SDK provides a peripheral driver for the System Mode Controller (SMC) module of MCUXpresso SDK devices. The SMC module sequences the system in and out of all low-power stop and run modes.

API functions are provided to configure the system for working in a dedicated power mode. For different power modes, `SMC_SetPowerModexxx()` function accepts different parameters. System power mode state transitions are not available between power modes. For details about available transitions, see the power mode transitions section in the SoC reference manual.

29.2 Typical use case

29.2.1 Enter wait or stop modes

SMC driver provides APIs to set MCU to different wait modes and stop modes. Pre and post functions are used for setting the modes. The pre functions and post functions are used as follows.

1. Disable/enable the interrupt through PRIMASK. This is an example use case. The application sets the wakeup interrupt and calls SMC function `SMC_SetPowerModeStop` to set the MCU to STOP mode, but the wakeup interrupt happens so quickly that the ISR completes before the function `SMC_SetPowerModeStop`. As a result, the MCU enters the STOP mode and never is woken up by the interrupt. In this use case, the application first disables the interrupt through PRIMASK, sets the wakeup interrupt, and enters the STOP mode. After wakeup, enable the interrupt through PRIMASK. The MCU can still be woken up by disabling the interrupt through PRIMASK. The pre and post functions handle the PRIMASK.
2. Disable/enable the flash speculation. When entering stop modes, the flash speculation might be interrupted. As a result, pre functions disable the flash speculation and post functions enable it.

```
SMC_PreEnterStopModes();  
  
/* Enable the wakeup interrupt here. */  
  
SMC_SetPowerModeStop(SMC, kSMC_PartialStop);  
  
SMC_PostExitStopModes();
```

Enumerations

- enum `smc_power_mode_protection_t` {
 `kSMC_AllowPowerModeVlp` = `SMC_PMPROT_AVLP_MASK`,
 `kSMC_AllowPowerModeAll` }
 Power Modes Protection.

Typical use case

- enum `smc_power_state_t` {
 `kSMC_PowerStateRun` = 0x01U << 0U,
 `kSMC_PowerStateStop` = 0x01U << 1U,
 `kSMC_PowerStateVlpr` = 0x01U << 2U,
 `kSMC_PowerStateVlprw` = 0x01U << 3U,
 `kSMC_PowerStateVlps` = 0x01U << 4U }
 Power Modes in PMSTAT.
- enum `smc_run_mode_t` {
 `kSMC_RunNormal` = 0U,
 `kSMC_RunVlpr` = 2U }
 Run mode definition.
- enum `smc_stop_mode_t` {
 `kSMC_StopNormal` = 0U,
 `kSMC_StopVlps` = 2U }
 Stop mode definition.
- enum `smc_partial_stop_option_t` {
 `kSMC_PartialStop` = 0U,
 `kSMC_PartialStop1` = 1U,
 `kSMC_PartialStop2` = 2U }
 Partial STOP option.
- enum `_smc_status` { `kStatus_SMC_StopAbort` = MAKE_STATUS(kStatusGroup_POWER, 0) }
 SMC configuration status.

Driver version

- #define `FSL_SMC_DRIVER_VERSION` (MAKE_VERSION(2, 0, 3))
 SMC driver version 2.0.3.

System mode controller APIs

- static void `SMC_SetPowerModeProtection` (SMC_Type *base, uint8_t allowedModes)
 Configures all power mode protection settings.
- static `smc_power_state_t` `SMC_GetPowerModeState` (SMC_Type *base)
 Gets the current power mode status.
- void `SMC_PreEnterStopModes` (void)
 Prepares to enter stop modes.
- void `SMC_PostExitStopModes` (void)
 Recovers after wake up from stop modes.
- static void `SMC_PreEnterWaitModes` (void)
 Prepares to enter wait modes.
- static void `SMC_PostExitWaitModes` (void)
 Recovers after wake up from stop modes.
- status_t `SMC_SetPowerModeRun` (SMC_Type *base)
 Configures the system to RUN power mode.
- status_t `SMC_SetPowerModeWait` (SMC_Type *base)
 Configures the system to WAIT power mode.
- status_t `SMC_SetPowerModeStop` (SMC_Type *base, `smc_partial_stop_option_t` option)
 Configures the system to Stop power mode.
- status_t `SMC_SetPowerModeVlpr` (SMC_Type *base)

- *Configures the system to VLPR power mode.*
status_t [SMC_SetPowerModeVlpw](#) (SMC_Type *base)
- *Configures the system to VLPW power mode.*
status_t [SMC_SetPowerModeVlps](#) (SMC_Type *base)
- *Configures the system to VLPS power mode.*

29.3 Macro Definition Documentation

29.3.1 #define FSL_SMC_DRIVER_VERSION (MAKE_VERSION(2, 0, 3))

29.4 Enumeration Type Documentation

29.4.1 enum smc_power_mode_protection_t

Enumerator

kSMC_AllowPowerModeVlp Allow Very-Low-power Mode.
kSMC_AllowPowerModeAll Allow all power mode.

29.4.2 enum smc_power_state_t

Enumerator

kSMC_PowerStateRun 0000_0001 - Current power mode is RUN
kSMC_PowerStateStop 0000_0010 - Current power mode is STOP
kSMC_PowerStateVlpr 0000_0100 - Current power mode is VLPR
kSMC_PowerStateVlpw 0000_1000 - Current power mode is VLPW
kSMC_PowerStateVlps 0001_0000 - Current power mode is VLPS

29.4.3 enum smc_run_mode_t

Enumerator

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

29.4.4 enum smc_stop_mode_t

Enumerator

kSMC_StopNormal Normal STOP mode.
kSMC_StopVlps Very-low-power STOP mode.

Function Documentation

29.4.5 enum smc_partial_stop_option_t

Enumerator

- kSMC_PartialStop* STOP - Normal Stop mode.
- kSMC_PartialStop1* Partial Stop with both system and bus clocks disabled.
- kSMC_PartialStop2* Partial Stop with system clock disabled and bus clock enabled.

29.4.6 enum _smc_status

Enumerator

- kStatus_SMC_StopAbort* Entering Stop mode is abort.

29.5 Function Documentation

29.5.1 static void SMC_SetPowerModeProtection (SMC_Type * *base*, uint8_t *allowedModes*) [inline], [static]

This function configures the power mode protection settings for supported power modes in the specified chip family. The available power modes are defined in the `smc_power_mode_protection_t`. This should be done at an early system level initialization stage. See the reference manual for details. This register can only write once after the power reset.

The allowed modes are passed as bit map. For example, to allow LLS and VLLS, use `SMC_SetPowerModeProtection(kSMC_AllowPowerModeVlls | kSMC_AllowPowerModeVlps)`. To allow all modes, use `SMC_SetPowerModeProtection(kSMC_AllowPowerModeAll)`.

Parameters

<i>base</i>	SMC peripheral base address.
<i>allowedModes</i>	Bitmap of the allowed power modes.

29.5.2 static smc_power_state_t SMC_GetPowerModeState (SMC_Type * *base*) [inline], [static]

This function returns the current power mode status. After the application switches the power mode, it should always check the status to check whether it runs into the specified mode or not. The application should check this mode before switching to a different mode. The system requires that only certain modes can switch to other specific modes. See the reference manual for details and the `smc_power_state_t` for information about the power status.

Parameters

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

Returns

Current power mode status.

29.5.3 void SMC_PreEnterStopModes (void)

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

29.5.4 void SMC_PostExitStopModes (void)

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

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

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

29.5.6 static void SMC_PostExitWaitModes (void) [inline], [static]

This function should be called after wake up from WAIT/VLPW modes. It is used with [SMC_PreEnterWaitModes](#).

29.5.7 status_t SMC_SetPowerModeRun (SMC_Type * *base*)

Parameters

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

Returns

SMC configuration error code.

29.5.8 status_t SMC_SetPowerModeWait (SMC_Type * *base*)

Function Documentation

Parameters

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

Returns

SMC configuration error code.

29.5.9 **status_t SMC_SetPowerModeStop (SMC_Type * *base*, smc_partial_stop_option_t *option*)**

Parameters

<i>base</i>	SMC peripheral base address.
<i>option</i>	Partial Stop mode option.

Returns

SMC configuration error code.

29.5.10 **status_t SMC_SetPowerModeVlpr (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

29.5.11 **status_t SMC_SetPowerModeVlprw (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.

29.5.12 **status_t SMC_SetPowerModeVlps (SMC_Type * *base*)**

Parameters

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

Returns

SMC configuration error code.



Chapter 30

UART: Universal Asynchronous Receiver/Transmitter Driver

30.1 Overview

Modules

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

30.2 UART Driver

30.2.1 Overview

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

The UART driver includes functional APIs and transactional APIs.

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

Transactional APIs can be used to enable the peripheral quickly and in the application if the code size and performance of transactional APIs can satisfy the requirements. If the code size and performance are critical requirements, see the transactional API implementation and write custom code. All transactional APIs use the `uart_handle_t` as the second parameter. Initialize the handle by calling the [UART_TransferCreateHandle\(\)](#) API.

Transactional APIs support asynchronous transfer, which means that the functions [UART_TransferSendNonBlocking\(\)](#) and [UART_TransferReceiveNonBlocking\(\)](#) set up an interrupt for data transfer. When the transfer completes, the upper layer is notified through a callback function with the `kStatus_UART_TxIdle` and `kStatus_UART_RxIdle`.

Transactional receive APIs support the ring buffer. Prepare the memory for the ring buffer and pass in the start address and size while calling the [UART_TransferCreateHandle\(\)](#). If passing `NULL`, the ring buffer feature is disabled. When the ring buffer is enabled, the received data is saved to the ring buffer in the background. The [UART_TransferReceiveNonBlocking\(\)](#) function first gets data from the ring buffer. If the ring buffer does not have enough data, the function first returns the data in the ring buffer and then saves the received data to user memory. When all data is received, the upper layer is informed through a callback with the `kStatus_UART_RxIdle`.

If the receive ring buffer is full, the upper layer is informed through a callback with the `kStatus_UART_RxRingBufferOverflow`. In the callback function, the upper layer reads data out from the ring buffer. If not, existing data is overwritten by the new data.

The ring buffer size is specified when creating the handle. Note that one byte is reserved for the ring buffer maintenance. When creating handle using the following code.

```
UART_TransferCreateHandle(UART0, &handle, UART_UserCallback, NULL);
```

In this example, the buffer size is 32, but only 31 bytes are used for saving data.

30.2.2 Typical use case

30.2.2.1 UART Send/receive using a polling method

```
uint8_t ch;
```

```

UART_GetDefaultConfig(&user_config);
user_config.baudRate_Bps = 115200U;
user_config.enableTx = true;
user_config.enableRx = true;

UART_Init(UART1, &user_config, 120000000U);

while(1)
{
    UART_ReadBlocking(UART1, &ch, 1);
    UART_WriteBlocking(UART1, &ch, 1);
}

```

30.2.2.2 UART Send/receive using an interrupt method

```

uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_TransferCreateHandle(UART1, &g_uartHandle, UART_UserCallback, NULL);

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

    // Send out.
    UART_TransferSendNonBlocking(&g_uartHandle, &g_uartHandle, &sendXfer);

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

    // Prepare to receive.

```

UART Driver

```
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
rxFinished = false;

// Receive.
UART_TransferReceiveNonBlocking(&g_uartHandle, &g_uartHandle, &
    receiveXfer);

// Wait receive finished.
while (!rxFinished)
{
}

// ...
}
```

30.2.2.3 UART Receive using the ringbuffer feature

```
#define RING_BUFFER_SIZE 64
#define RX_DATA_SIZE 32

uart_handle_t g_uartHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t receiveData[RX_DATA_SIZE];
uint8_t ringBuffer[RING_BUFFER_SIZE];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    size_t bytesRead;
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);
    UART_TransferCreateHandle(UART1, &g_uartHandle, UART_UserCallback, NULL);

    // Now the RX is working in background, receive in to ring buffer.

    // Prepare to receive.
    receiveXfer.data = receiveData;
    receiveXfer.dataSize = RX_DATA_SIZE;
    rxFinished = false;

    // Receive.
    UART_TransferReceiveNonBlocking(UART1, &g_uartHandle, &receiveXfer);

    if (bytesRead = RX_DATA_SIZE) /* Have read enough data. */
    {
```



```

    ;
}
else
{
    if (bytesRead) /* Received some data, process first. */
    {
        ;
    }

    // Wait receive finished.
    while (!rxFinished)
    {
    }
}

// ...
}

```

30.2.2.4 UART Send/Receive using the DMA method

```

uart_handle_t g_uartHandle;
dma_handle_t g_uartTxDmaHandle;
dma_handle_t g_uartRxDmaHandle;
uart_config_t user_config;
uart_transfer_t sendXfer;
uart_transfer_t receiveXfer;
volatile bool txFinished;
volatile bool rxFinished;
uint8_t sendData[] = {'H', 'e', 'l', 'l', 'o'};
uint8_t receiveData[32];

void UART_UserCallback(uart_handle_t *handle, status_t status, void *userData)
{
    userData = userData;

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

    if (kStatus_UART_RxIdle == status)
    {
        rxFinished = true;
    }
}

void main(void)
{
    //...

    UART_GetDefaultConfig(&user_config);
    user_config.baudRate_Bps = 115200U;
    user_config.enableTx = true;
    user_config.enableRx = true;

    UART_Init(UART1, &user_config, 120000000U);

    // Set up the DMA
    DMAMUX_Init(DMAMUX0);
    DMAMUX_SetSource(DMAMUX0, UART_TX_DMA_CHANNEL, UART_TX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_TX_DMA_CHANNEL);
    DMAMUX_SetSource(DMAMUX0, UART_RX_DMA_CHANNEL, UART_RX_DMA_REQUEST);
    DMAMUX_EnableChannel(DMAMUX0, UART_RX_DMA_CHANNEL);

    DMA_Init(DMA0);
}

```

UART Driver

```
/* Create DMA handle. */
DMA_CreateHandle(&g_uartTxDmaHandle, DMA0, UART_TX_DMA_CHANNEL);
DMA_CreateHandle(&g_uartRxDmaHandle, DMA0, UART_RX_DMA_CHANNEL);

UART_TransferCreateHandleDMA(UART1, &g_uartHandle, UART_UserCallback, NULL,
    &g_uartTxDmaHandle, &g_uartRxDmaHandle);

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

// Send out.
UART_TransferSendDMA(UART1, &g_uartHandle, &sendXfer);

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

// Prepare to receive.
receiveXfer.data = receiveData;
receiveXfer.dataSize = sizeof(receiveData)/sizeof(receiveData[0]);
rxFinished = false;

// Receive.
UART_TransferReceiveDMA(UART1, &g_uartHandle, &receiveXfer);

// Wait receive finished.
while (!rxFinished)
{
}

// ...
}
```

Data Structures

- struct `uart_config_t`
UART configuration structure. [More...](#)
- struct `uart_transfer_t`
UART transfer structure. [More...](#)
- struct `uart_handle_t`
UART handle structure. [More...](#)

Typedefs

- typedef void(* `uart_transfer_callback_t`)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)
UART transfer callback function.

Enumerations

- enum `_uart_status` {
 - `kStatus_UART_TxBusy` = MAKE_STATUS(kStatusGroup_UART, 0),
 - `kStatus_UART_RxBusy` = MAKE_STATUS(kStatusGroup_UART, 1),
 - `kStatus_UART_TxIdle` = MAKE_STATUS(kStatusGroup_UART, 2),
 - `kStatus_UART_RxIdle` = MAKE_STATUS(kStatusGroup_UART, 3),
 - `kStatus_UART_TxWatermarkTooLarge` = MAKE_STATUS(kStatusGroup_UART, 4),
 - `kStatus_UART_RxWatermarkTooLarge` = MAKE_STATUS(kStatusGroup_UART, 5),
 - `kStatus_UART_FlagCannotClearManually`,
 - `kStatus_UART_Error` = MAKE_STATUS(kStatusGroup_UART, 7),
 - `kStatus_UART_RxRingBufferOverflow` = MAKE_STATUS(kStatusGroup_UART, 8),
 - `kStatus_UART_RxHardwareOverflow` = MAKE_STATUS(kStatusGroup_UART, 9),
 - `kStatus_UART_NoiseError` = MAKE_STATUS(kStatusGroup_UART, 10),
 - `kStatus_UART_FramingError` = MAKE_STATUS(kStatusGroup_UART, 11),
 - `kStatus_UART_ParityError` = MAKE_STATUS(kStatusGroup_UART, 12),
 - `kStatus_UART_BaudrateNotSupport` }

Error codes for the UART driver.
- enum `uart_parity_mode_t` {
 - `kUART_ParityDisabled` = 0x0U,
 - `kUART_ParityEven` = 0x2U,
 - `kUART_ParityOdd` = 0x3U }

UART parity mode.
- enum `uart_stop_bit_count_t` {
 - `kUART_OneStopBit` = 0U,
 - `kUART_TwoStopBit` = 1U }

UART stop bit count.
- enum `_uart_interrupt_enable` {
 - `kUART_RxActiveEdgeInterruptEnable` = (UART_BDH_RXEDGIE_MASK),
 - `kUART_TxDataRegEmptyInterruptEnable` = (UART_C2_TIE_MASK << 8),
 - `kUART_TransmissionCompleteInterruptEnable` = (UART_C2_TCIE_MASK << 8),
 - `kUART_RxDataRegFullInterruptEnable` = (UART_C2_RIE_MASK << 8),
 - `kUART_IdleLineInterruptEnable` = (UART_C2_ILIE_MASK << 8),
 - `kUART_RxOverflowInterruptEnable` = (UART_C3_ORIE_MASK << 16),
 - `kUART_NoiseErrorInterruptEnable` = (UART_C3_NEIE_MASK << 16),
 - `kUART_FramingErrorInterruptEnable` = (UART_C3_FEIE_MASK << 16),
 - `kUART_ParityErrorInterruptEnable` = (UART_C3_PEIE_MASK << 16) }

UART interrupt configuration structure, default settings all disabled.
- enum `_uart_flags` {

UART Driver

```
kUART_TxDataRegEmptyFlag = (UART_S1_TDRE_MASK),  
kUART_TransmissionCompleteFlag = (UART_S1_TC_MASK),  
kUART_RxDataRegFullFlag = (UART_S1_RDRF_MASK),  
kUART_IdleLineFlag = (UART_S1_IDLE_MASK),  
kUART_RxOverrunFlag = (UART_S1_OR_MASK),  
kUART_NoiseErrorFlag = (UART_S1_NF_MASK),  
kUART_FramingErrorFlag = (UART_S1_FE_MASK),  
kUART_ParityErrorFlag = (UART_S1_PF_MASK),  
kUART_RxActiveEdgeFlag,  
kUART_RxActiveFlag }
```

UART status flags.

Driver version

- #define **FSL_UART_DRIVER_VERSION** (MAKE_VERSION(2, 1, 4))
UART driver version 2.1.4.

Initialization and deinitialization

- status_t **UART_Init** (UART_Type *base, const **uart_config_t** *config, uint32_t srcClock_Hz)
Initializes a UART instance with a user configuration structure and peripheral clock.
- void **UART_Deinit** (UART_Type *base)
Deinitializes a UART instance.
- void **UART_GetDefaultConfig** (**uart_config_t** *config)
Gets the default configuration structure.
- status_t **UART_SetBaudRate** (UART_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)
Sets the UART instance baud rate.

Status

- uint32_t **UART_GetStatusFlags** (UART_Type *base)
Gets UART status flags.
- status_t **UART_ClearStatusFlags** (UART_Type *base, uint32_t mask)
Clears status flags with the provided mask.

Interrupts

- void **UART_EnableInterrupts** (UART_Type *base, uint32_t mask)
Enables UART interrupts according to the provided mask.
- void **UART_DisableInterrupts** (UART_Type *base, uint32_t mask)
Disables the UART interrupts according to the provided mask.
- uint32_t **UART_GetEnabledInterrupts** (UART_Type *base)
Gets the enabled UART interrupts.

Bus Operations

- static void [UART_EnableTx](#) (UART_Type *base, bool enable)
Enables or disables the UART transmitter.
- static void [UART_EnableRx](#) (UART_Type *base, bool enable)
Enables or disables the UART receiver.
- static void [UART_WriteByte](#) (UART_Type *base, uint8_t data)
Writes to the TX register.
- static uint8_t [UART_ReadByte](#) (UART_Type *base)
Reads the RX register directly.
- void [UART_WriteBlocking](#) (UART_Type *base, const uint8_t *data, size_t length)
Writes to the TX register using a blocking method.
- status_t [UART_ReadBlocking](#) (UART_Type *base, uint8_t *data, size_t length)
Read RX data register using a blocking method.

Transactional

- void [UART_TransferCreateHandle](#) (UART_Type *base, uart_handle_t *handle, [uart_transfer_callback_t](#) callback, void *userData)
Initializes the UART handle.
- void [UART_TransferStartRingBuffer](#) (UART_Type *base, uart_handle_t *handle, uint8_t *ringBuffer, size_t ringBufferSize)
Sets up the RX ring buffer.
- void [UART_TransferStopRingBuffer](#) (UART_Type *base, uart_handle_t *handle)
Aborts the background transfer and uninstalls the ring buffer.
- status_t [UART_TransferSendNonBlocking](#) (UART_Type *base, uart_handle_t *handle, [uart_transfer_t](#) *xfer)
Transmits a buffer of data using the interrupt method.
- void [UART_TransferAbortSend](#) (UART_Type *base, uart_handle_t *handle)
Aborts the interrupt-driven data transmit.
- status_t [UART_TransferGetSendCount](#) (UART_Type *base, uart_handle_t *handle, uint32_t *count)
Gets the number of bytes written to the UART TX register.
- status_t [UART_TransferReceiveNonBlocking](#) (UART_Type *base, uart_handle_t *handle, [uart_transfer_t](#) *xfer, size_t *receivedBytes)
Receives a buffer of data using an interrupt method.
- void [UART_TransferAbortReceive](#) (UART_Type *base, uart_handle_t *handle)
Aborts the interrupt-driven data receiving.
- status_t [UART_TransferGetReceiveCount](#) (UART_Type *base, uart_handle_t *handle, uint32_t *count)
Gets the number of bytes that have been received.
- void [UART_TransferHandleIRQ](#) (UART_Type *base, uart_handle_t *handle)
UART IRQ handle function.
- void [UART_TransferHandleErrorIRQ](#) (UART_Type *base, uart_handle_t *handle)
UART Error IRQ handle function.

30.2.3 Data Structure Documentation

30.2.3.1 struct uart_config_t

Data Fields

- uint32_t [baudRate_Bps](#)
UART baud rate.
- [uart_parity_mode_t](#) [parityMode](#)
Parity mode, disabled (default), even, odd.
- bool [enableTx](#)
Enable TX.
- bool [enableRx](#)
Enable RX.

30.2.3.2 struct uart_transfer_t

Data Fields

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

30.2.3.2.0.63 Field Documentation

30.2.3.2.0.63.1 uint8_t* uart_transfer_t::data

30.2.3.2.0.63.2 size_t uart_transfer_t::dataSize

30.2.3.3 struct _uart_handle

Data Fields

- uint8_t *volatile [txData](#)
Address of remaining data to send.
- volatile size_t [txDataSize](#)
Size of the remaining data to send.
- size_t [txDataSizeAll](#)
Size of the data to send out.
- uint8_t *volatile [rxData](#)
Address of remaining data to receive.
- volatile size_t [rxDataSize](#)
Size of the remaining data to receive.
- size_t [rxDataSizeAll](#)
Size of the data to receive.
- uint8_t * [rxRingBuffer](#)
Start address of the receiver ring buffer.
- size_t [rxRingBufferSize](#)

- *Size of the ring buffer.*
volatile uint16_t [rxRingBufferHead](#)
- *Index for the driver to store received data into ring buffer.*
volatile uint16_t [rxRingBufferTail](#)
- *Index for the user to get data from the ring buffer.*
[uart_transfer_callback_t](#) [callback](#)
- *Callback function.*
void * [userData](#)
- *UART callback function parameter.*
volatile uint8_t [txState](#)
- *TX transfer state.*
volatile uint8_t [rxState](#)
- *RX transfer state.*

UART Driver

30.2.3.3.0.64 Field Documentation

- 30.2.3.3.0.64.1 `uint8_t* volatile uart_handle_t::txData`
- 30.2.3.3.0.64.2 `volatile size_t uart_handle_t::txDataSize`
- 30.2.3.3.0.64.3 `size_t uart_handle_t::txDataSizeAll`
- 30.2.3.3.0.64.4 `uint8_t* volatile uart_handle_t::rxData`
- 30.2.3.3.0.64.5 `volatile size_t uart_handle_t::rxDataSize`
- 30.2.3.3.0.64.6 `size_t uart_handle_t::rxDataSizeAll`
- 30.2.3.3.0.64.7 `uint8_t* uart_handle_t::rxRingBuffer`
- 30.2.3.3.0.64.8 `size_t uart_handle_t::rxRingBufferSize`
- 30.2.3.3.0.64.9 `volatile uint16_t uart_handle_t::rxRingBufferHead`
- 30.2.3.3.0.64.10 `volatile uint16_t uart_handle_t::rxRingBufferTail`
- 30.2.3.3.0.64.11 `uart_transfer_callback_t uart_handle_t::callback`
- 30.2.3.3.0.64.12 `void* uart_handle_t::userData`
- 30.2.3.3.0.64.13 `volatile uint8_t uart_handle_t::txState`

30.2.4 Macro Definition Documentation

- 30.2.4.1 `#define FSL_UART_DRIVER_VERSION (MAKE_VERSION(2, 1, 4))`

30.2.5 Typedef Documentation

- 30.2.5.1 `typedef void(* uart_transfer_callback_t)(UART_Type *base, uart_handle_t *handle, status_t status, void *userData)`

30.2.6 Enumeration Type Documentation

30.2.6.1 `enum _uart_status`

Enumerator

- kStatus_UART_TxBusy* Transmitter is busy.
- kStatus_UART_RxBusy* Receiver is busy.
- kStatus_UART_TxIdle* UART transmitter is idle.
- kStatus_UART_RxIdle* UART receiver is idle.
- kStatus_UART_TxWatermarkTooLarge* TX FIFO watermark too large.

kStatus_UART_RxWatermarkTooLarge RX FIFO watermark too large.
kStatus_UART_FlagCannotClearManually UART flag can't be manually cleared.
kStatus_UART_Error Error happens on UART.
kStatus_UART_RxRingBufferOverflow UART RX software ring buffer overrun.
kStatus_UART_RxHardwareOverflow UART RX receiver overrun.
kStatus_UART_NoiseError UART noise error.
kStatus_UART_FramingError UART framing error.
kStatus_UART_ParityError UART parity error.
kStatus_UART_BaudrateNotSupport Baudrate is not support in current clock source.

30.2.6.2 enum uart_parity_mode_t

Enumerator

kUART_ParityDisabled Parity disabled.
kUART_ParityEven Parity enabled, type even, bit setting: PE|PT = 10.
kUART_ParityOdd Parity enabled, type odd, bit setting: PE|PT = 11.

30.2.6.3 enum uart_stop_bit_count_t

Enumerator

kUART_OneStopBit One stop bit.
kUART_TwoStopBit Two stop bits.

30.2.6.4 enum _uart_interrupt_enable

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

Enumerator

kUART_RxActiveEdgeInterruptEnable RX active edge interrupt.
kUART_TxDataRegEmptyInterruptEnable Transmit data register empty interrupt.
kUART_TransmissionCompleteInterruptEnable Transmission complete interrupt.
kUART_RxDataRegFullInterruptEnable Receiver data register full interrupt.
kUART_IdleLineInterruptEnable Idle line interrupt.
kUART_RxOverflowInterruptEnable Receiver overrun interrupt.
kUART_NoiseErrorInterruptEnable Noise error flag interrupt.
kUART_FramingErrorInterruptEnable Framing error flag interrupt.
kUART_ParityErrorInterruptEnable Parity error flag interrupt.

UART Driver

30.2.6.5 enum _uart_flags

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

Enumerator

kUART_TxDataRegEmptyFlag TX data register empty flag.
kUART_TransmissionCompleteFlag Transmission complete flag.
kUART_RxDataRegFullFlag RX data register full flag.
kUART_IdleLineFlag Idle line detect flag.
kUART_RxOverrunFlag RX overrun flag.
kUART_NoiseErrorFlag RX takes 3 samples of each received bit. If any of these samples differ, noise flag sets
kUART_FramingErrorFlag Frame error flag, sets if logic 0 was detected where stop bit expected.
kUART_ParityErrorFlag If parity enabled, sets upon parity error detection.
kUART_RxActiveEdgeFlag RX pin active edge interrupt flag, sets when active edge detected.
kUART_RxActiveFlag Receiver Active Flag (RAF), sets at beginning of valid start bit.

30.2.7 Function Documentation

30.2.7.1 status_t UART_Init (UART_Type * *base*, const uart_config_t * *config*, uint32_t *srcClock_Hz*)

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

```
* uart_config_t uartConfig;  
* uartConfig.baudRate_Bps = 115200U;  
* uartConfig.parityMode = kUART_ParityDisabled;  
* uartConfig.stopBitCount = kUART_OneStopBit;  
* uartConfig.txFifoWatermark = 0;  
* uartConfig.rxFifoWatermark = 1;  
* UART_Init(UART1, &uartConfig, 200000000U);  
*
```

Parameters

<i>base</i>	UART peripheral base address.
<i>config</i>	Pointer to the user-defined configuration structure.

<i>srcClock_Hz</i>	UART clock source frequency in HZ.
--------------------	------------------------------------

Return values

<i>kStatus_UART_Baudrate-NotSupport</i>	Baudrate is not support in current clock source.
<i>kStatus_Success</i>	Status UART initialize succeed

30.2.7.2 void UART_Deinit (UART_Type * *base*)

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

Parameters

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

30.2.7.3 void UART_GetDefaultConfig (uart_config_t * *config*)

This function initializes the UART configuration structure to a default value. The default values are as follows. `uartConfig->baudRate_Bps = 115200U`; `uartConfig->bitCountPerChar = kUART_8BitsPerChar`; `uartConfig->parityMode = kUART_ParityDisabled`; `uartConfig->stopBitCount = kUART_OneStopBit`; `uartConfig->txFifoWatermark = 0`; `uartConfig->rxFifoWatermark = 1`; `uartConfig->enableTx = false`; `uartConfig->enableRx = false`;

Parameters

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

30.2.7.4 status_t UART_SetBaudRate (UART_Type * *base*, uint32_t *baudRate_Bps*, uint32_t *srcClock_Hz*)

This function configures the UART module baud rate. This function is used to update the UART module baud rate after the UART module is initialized by the `UART_Init`.

```
* UART_SetBaudRate(UART1, 115200U, 200000000U);
*
```

UART Driver

Parameters

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

Return values

<i>kStatus_UART_Baudrate-NotSupport</i>	Baudrate is not support in the current clock source.
<i>kStatus_Success</i>	Set baudrate succeeded.

30.2.7.5 uint32_t UART_GetStatusFlags (UART_Type * *base*)

This function gets all UART status flags. The flags are returned as the logical OR value of the enumerators [_uart_flags](#). To check a specific status, compare the return value with enumerators in [_uart_flags](#). For example, to check whether the TX is empty, do the following.

```
*      if (kUART_TxDataRegEmptyFlag & UART_GetStatusFlags(UART1))
*      {
*          ...
*      }
*
```

Parameters

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

Returns

UART status flags which are ORed by the enumerators in the [_uart_flags](#).

30.2.7.6 status_t UART_ClearStatusFlags (UART_Type * *base*, uint32_t *mask*)

This function clears UART status flags with a provided mask. An automatically cleared flag can't be cleared by this function. These flags can only be cleared or set by hardware. kUART_TxDataRegEmptyFlag, kUART_TransmissionCompleteFlag, kUART_RxDataRegFullFlag, kUART_RxActiveFlag, kUART_NoiseErrorInRxDataRegFlag, kUART_ParityErrorInRxDataRegFlag, kUART_TxFifoEmptyFlag, kUART_RxFifoEmptyFlag Note that this API should be called when the Tx/Rx is idle. Otherwise it has no effect.

Parameters

<i>base</i>	UART peripheral base address.
<i>mask</i>	The status flags to be cleared; it is logical OR value of _uart_flags .

Return values

<i>kStatus_UART_Flag- CannotClearManually</i>	The flag can't be cleared by this function but it is cleared automatically by hardware.
<i>kStatus_Success</i>	Status in the mask is cleared.

30.2.7.7 void UART_EnableInterrupts (UART_Type * *base*, uint32_t *mask*)

This function enables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [_uart_interrupt_enable](#). For example, to enable TX empty interrupt and RX full interrupt, do the following.

```
*  UART_EnableInterrupts(UART1,
    kUART_TxDataRegEmptyInterruptEnable |
    kUART_RxDataRegFullInterruptEnable);
*
```

Parameters

<i>base</i>	UART peripheral base address.
<i>mask</i>	The interrupts to enable. Logical OR of _uart_interrupt_enable .

30.2.7.8 void UART_DisableInterrupts (UART_Type * *base*, uint32_t *mask*)

This function disables the UART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See [_uart_interrupt_enable](#). For example, to disable TX empty interrupt and RX full interrupt do the following.

```
*  UART_DisableInterrupts(UART1,
    kUART_TxDataRegEmptyInterruptEnable |
    kUART_RxDataRegFullInterruptEnable);
*
```

UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>mask</i>	The interrupts to disable. Logical OR of _uart_interrupt_enable .

30.2.7.9 uint32_t UART_GetEnabledInterrupts (UART_Type * *base*)

This function gets the enabled UART interrupts. The enabled interrupts are returned as the logical OR value of the enumerators [_uart_interrupt_enable](#). To check a specific interrupts enable status, compare the return value with enumerators in [_uart_interrupt_enable](#). For example, to check whether TX empty interrupt is enabled, do the following.

```
*    uint32_t enabledInterrupts = UART_GetEnabledInterrupts(UART1);  
*  
*    if (kUART_TxDataRegEmptyInterruptEnable & enabledInterrupts)  
*    {  
*        ...  
*    }  
*
```

Parameters

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

Returns

UART interrupt flags which are logical OR of the enumerators in [_uart_interrupt_enable](#).

30.2.7.10 static void UART_EnableTx (UART_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the UART transmitter.

Parameters

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

30.2.7.11 static void UART_EnableRx (UART_Type * *base*, bool *enable*) [inline], [static]

This function enables or disables the UART receiver.

Parameters

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

30.2.7.12 static void UART_WriteByte (UART_Type * *base*, uint8_t *data*) [inline], [static]

This function writes data to the TX register directly. The upper layer must ensure that the TX register is empty or TX FIFO has empty room before calling this function.

Parameters

<i>base</i>	UART peripheral base address.
<i>data</i>	The byte to write.

30.2.7.13 static uint8_t UART_ReadByte (UART_Type * *base*) [inline], [static]

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

Parameters

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

Returns

The byte read from UART data register.

30.2.7.14 void UART_WriteBlocking (UART_Type * *base*, const uint8_t * *data*, size_t *length*)

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Note

This function does not check whether all data is sent out to the bus. Before disabling the TX, check kUART_TransmissionCompleteFlag to ensure that the TX is finished.

UART Driver

Parameters

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

30.2.7.15 **status_t** UART_ReadBlocking (**UART_Type** * *base*, **uint8_t** * *data*, **size_t** *length*)

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

Parameters

<i>base</i>	UART peripheral base address.
<i>data</i>	Start address of the buffer to store the received data.
<i>length</i>	Size of the buffer.

Return values

<i>kStatus_UART_Rx-HardwareOverrun</i>	Receiver overrun occurred while receiving data.
<i>kStatus_UART_Noise-Error</i>	A noise error occurred while receiving data.
<i>kStatus_UART_Framing-Error</i>	A framing error occurred while receiving data.
<i>kStatus_UART_Parity-Error</i>	A parity error occurred while receiving data.
<i>kStatus_Success</i>	Successfully received all data.

30.2.7.16 **void** UART_TransferCreateHandle (**UART_Type** * *base*, **uart_handle_t** * *handle*, **uart_transfer_callback_t** *callback*, **void** * *userData*)

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

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>callback</i>	The callback function.
<i>userData</i>	The parameter of the callback function.

30.2.7.17 void UART_TransferStartRingBuffer (UART_Type * *base*, uart_handle_t * *handle*, uint8_t * *ringBuffer*, size_t *ringBufferSize*)

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

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

Note

When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, only 31 bytes are used for saving data.

Parameters

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

30.2.7.18 void UART_TransferStopRingBuffer (UART_Type * *base*, uart_handle_t * *handle*)

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

UART Driver

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

30.2.7.19 **status_t UART_TransferSendNonBlocking (UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_t * *xfer*)**

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the ISR, the UART driver calls the callback function and passes the [kStatus_UART_TxIdle](#) as status parameter.

Note

The [kStatus_UART_TxIdle](#) is passed to the upper layer when all data is written to the TX register. However, it does not ensure that all data is sent out. Before disabling the TX, check the [kUART_TransmissionCompleteFlag](#) to ensure that the TX is finished.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART transfer structure. See uart_transfer_t .

Return values

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

30.2.7.20 **void UART_TransferAbortSend (UART_Type * *base*, uart_handle_t * *handle*)**

This function aborts the interrupt-driven data sending. The user can get the `remainBytes` to find out how many bytes are not sent out.

Parameters

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

30.2.7.21 **status_t UART_TransferGetSendCount (UART_Type * *base*, uart_handle_t * *handle*, uint32_t * *count*)**

This function gets the number of bytes written to the UART TX register by using the interrupt method.

Parameters

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

Return values

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

30.2.7.22 **status_t UART_TransferReceiveNonBlocking (UART_Type * *base*, uart_handle_t * *handle*, uart_transfer_t * *xfer*, size_t * *receivedBytes*)**

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter *receivedBytes* shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the UART driver. When the new data arrives, the receive request is serviced first. When all data is received, the UART driver notifies the upper layer through a callback function and passes the status parameter [kStatus_UART_RxIdle](#). For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the *xfer->data* and this function returns with the parameter *receivedBytes* set to 5. For the left 5 bytes, newly arrived data is saved from the *xfer->data[5]*. When 5 bytes are received, the UART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the *xfer->data*. When all data is received, the upper layer is notified.

UART Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART transfer structure, see uart_transfer_t .
<i>receivedBytes</i>	Bytes received from the ring buffer directly.

Return values

<i>kStatus_Success</i>	Successfully queue the transfer into transmit queue.
<i>kStatus_UART_RxBusy</i>	Previous receive request is not finished.
<i>kStatus_InvalidArgument</i>	Invalid argument.

30.2.7.23 void UART_TransferAbortReceive (UART_Type * *base*, uart_handle_t * *handle*)

This function aborts the interrupt-driven data receiving. The user can get the remainBytes to know how many bytes are not received yet.

Parameters

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

30.2.7.24 status_t UART_TransferGetReceiveCount (UART_Type * *base*, uart_handle_t * *handle*, uint32_t * *count*)

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

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Receive bytes count.

Return values

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

30.2.7.25 void UART_TransferHandleIRQ (UART_Type * *base*, uart_handle_t * *handle*)

This function handles the UART transmit and receive IRQ request.

Parameters

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

30.2.7.26 void UART_TransferHandleErrorIRQ (UART_Type * *base*, uart_handle_t * *handle*)

This function handles the UART error IRQ request.

Parameters

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

30.3 UART DMA Driver

30.3.1 Overview

Data Structures

- struct [uart_dma_handle_t](#)
UART DMA handle. [More...](#)

Typedefs

- typedef void(* [uart_dma_transfer_callback_t](#))(UART_Type *base, uart_dma_handle_t *handle, status_t status, void *userData)
UART transfer callback function.

eDMA transactional

- void [UART_TransferCreateHandleDMA](#) (UART_Type *base, uart_dma_handle_t *handle, [uart_dma_transfer_callback_t](#) callback, void *userData, dma_handle_t *txDmaHandle, dma_handle_t *rxDmaHandle)
Initializes the UART handle which is used in transactional functions and sets the callback.
- status_t [UART_TransferSendDMA](#) (UART_Type *base, uart_dma_handle_t *handle, [uart_transfer_t](#) *xfer)
Sends data using DMA.
- status_t [UART_TransferReceiveDMA](#) (UART_Type *base, uart_dma_handle_t *handle, [uart_transfer_t](#) *xfer)
Receives data using DMA.
- void [UART_TransferAbortSendDMA](#) (UART_Type *base, uart_dma_handle_t *handle)
Aborts the send data using DMA.
- void [UART_TransferAbortReceiveDMA](#) (UART_Type *base, uart_dma_handle_t *handle)
Aborts the received data using DMA.
- status_t [UART_TransferGetSendCountDMA](#) (UART_Type *base, uart_dma_handle_t *handle, uint32_t *count)
Gets the number of bytes written to UART TX register.
- status_t [UART_TransferGetReceiveCountDMA](#) (UART_Type *base, uart_dma_handle_t *handle, uint32_t *count)
Gets the number of bytes that have been received.

30.3.2 Data Structure Documentation

30.3.2.1 struct _uart_dma_handle

Data Fields

- UART_Type * [base](#)

- *UART peripheral base address.*
• `uart_dma_transfer_callback_t` `callback`
Callback function.
- `void * userData`
UART callback function parameter.
- `size_t rxDataSizeAll`
Size of the data to receive.
- `size_t txDataSizeAll`
Size of the data to send out.
- `dma_handle_t * txDmaHandle`
The DMA TX channel used.
- `dma_handle_t * rxDmaHandle`
The DMA RX channel used.
- `volatile uint8_t txState`
TX transfer state.
- `volatile uint8_t rxState`
RX transfer state.

30.3.2.1.0.65 Field Documentation

30.3.2.1.0.65.1 `UART_Type* uart_dma_handle_t::base`

30.3.2.1.0.65.2 `uart_dma_transfer_callback_t uart_dma_handle_t::callback`

30.3.2.1.0.65.3 `void* uart_dma_handle_t::userData`

30.3.2.1.0.65.4 `size_t uart_dma_handle_t::rxDataSizeAll`

30.3.2.1.0.65.5 `size_t uart_dma_handle_t::txDataSizeAll`

30.3.2.1.0.65.6 `dma_handle_t* uart_dma_handle_t::txDmaHandle`

30.3.2.1.0.65.7 `dma_handle_t* uart_dma_handle_t::rxDmaHandle`

30.3.2.1.0.65.8 `volatile uint8_t uart_dma_handle_t::txState`

30.3.3 Typedef Documentation

30.3.3.1 `typedef void(* uart_dma_transfer_callback_t)(UART_Type *base,
uart_dma_handle_t *handle, status_t status, void *userData)`

30.3.4 Function Documentation

30.3.4.1 `void UART_TransferCreateHandleDMA (UART_Type * base, uart_dma_handle_t
* handle, uart_dma_transfer_callback_t callback, void * userData,
dma_handle_t * txDmaHandle, dma_handle_t * rxDmaHandle)`

UART DMA Driver

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_dma_handle_t</code> structure.
<i>callback</i>	UART callback, NULL means no callback.
<i>userData</i>	User callback function data.
<i>rxDmaHandle</i>	User requested DMA handle for the RX DMA transfer.
<i>txDmaHandle</i>	User requested DMA handle for the TX DMA transfer.

30.3.4.2 **status_t** UART_TransferSendDMA (**UART_Type** * *base*, **uart_dma_handle_t** * *handle*, **uart_transfer_t** * *xfer*)

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

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART DMA transfer structure. See uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

30.3.4.3 **status_t** UART_TransferReceiveDMA (**UART_Type** * *base*, **uart_dma_handle_t** * *handle*, **uart_transfer_t** * *xfer*)

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

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_dma_handle_t</code> structure.
<i>xfer</i>	UART DMA transfer structure. See uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_RxBusy</i>	Previous transfer on going.
<i>kStatus_InvalidArgument</i>	Invalid argument.

30.3.4.4 void UART_TransferAbortSendDMA (UART_Type * *base*, `uart_dma_handle_t` * *handle*)

This function aborts the sent data using DMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to <code>uart_dma_handle_t</code> structure.

30.3.4.5 void UART_TransferAbortReceiveDMA (UART_Type * *base*, `uart_dma_handle_t` * *handle*)

This function abort receive data which using DMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to <code>uart_dma_handle_t</code> structure.

30.3.4.6 status_t UART_TransferGetSendCountDMA (UART_Type * *base*, `uart_dma_handle_t` * *handle*, `uint32_t` * *count*)

This function gets the number of bytes written to UART TX register by DMA.

UART DMA Driver

Parameters

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

Return values

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

30.3.4.7 **status_t** UART_TransferGetReceiveCountDMA (**UART_Type** * *base*, **uart_dma_handle_t** * *handle*, **uint32_t** * *count*)

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

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Receive bytes count.

Return values

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

30.4 UART eDMA Driver

30.4.1 Overview

Data Structures

- struct [uart_edma_handle_t](#)
UART eDMA handle. [More...](#)

Typedefs

- typedef void(* [uart_edma_transfer_callback_t](#))(UART_Type *base, uart_edma_handle_t *handle, status_t status, void *userData)
UART transfer callback function.

eDMA transactional

- void [UART_TransferCreateHandleEDMA](#) (UART_Type *base, uart_edma_handle_t *handle, [uart_edma_transfer_callback_t](#) callback, void *userData, [edma_handle_t](#) *txEdmaHandle, [edma_handle_t](#) *rxEdmaHandle)
Initializes the UART handle which is used in transactional functions.
- status_t [UART_SendEDMA](#) (UART_Type *base, uart_edma_handle_t *handle, [uart_transfer_t](#) *xfer)
Sends data using eDMA.
- status_t [UART_ReceiveEDMA](#) (UART_Type *base, uart_edma_handle_t *handle, [uart_transfer_t](#) *xfer)
Receives data using eDMA.
- void [UART_TransferAbortSendEDMA](#) (UART_Type *base, uart_edma_handle_t *handle)
Aborts the sent data using eDMA.
- void [UART_TransferAbortReceiveEDMA](#) (UART_Type *base, uart_edma_handle_t *handle)
Aborts the receive data using eDMA.
- status_t [UART_TransferGetSendCountEDMA](#) (UART_Type *base, uart_edma_handle_t *handle, uint32_t *count)
Gets the number of bytes that have been written to UART TX register.
- status_t [UART_TransferGetReceiveCountEDMA](#) (UART_Type *base, uart_edma_handle_t *handle, uint32_t *count)
Gets the number of received bytes.

30.4.2 Data Structure Documentation

30.4.2.1 struct _uart_edma_handle

Data Fields

- [uart_edma_transfer_callback_t](#) callback

UART eDMA Driver

- *Callback function.*
void * [userData](#)
- *UART callback function parameter.*
size_t [rxDataSizeAll](#)
- *Size of the data to receive.*
size_t [txDataSizeAll](#)
- *Size of the data to send out.*
[edma_handle_t](#) * [txEdmaHandle](#)
- *The eDMA TX channel used.*
[edma_handle_t](#) * [rxEdmaHandle](#)
- *The eDMA RX channel used.*
uint8_t [nbytes](#)
- *eDMA minor byte transfer count initially configured.*
volatile uint8_t [txState](#)
- *TX transfer state.*
volatile uint8_t [rxState](#)
- *RX transfer state.*

30.4.2.1.0.66 Field Documentation

30.4.2.1.0.66.1 [uart_edma_transfer_callback_t](#) [uart_edma_handle_t::callback](#)

30.4.2.1.0.66.2 void* [uart_edma_handle_t::userData](#)

30.4.2.1.0.66.3 size_t [uart_edma_handle_t::rxDataSizeAll](#)

30.4.2.1.0.66.4 size_t [uart_edma_handle_t::txDataSizeAll](#)

30.4.2.1.0.66.5 [edma_handle_t](#)* [uart_edma_handle_t::txEdmaHandle](#)

30.4.2.1.0.66.6 [edma_handle_t](#)* [uart_edma_handle_t::rxEdmaHandle](#)

30.4.2.1.0.66.7 uint8_t [uart_edma_handle_t::nbytes](#)

30.4.2.1.0.66.8 volatile uint8_t [uart_edma_handle_t::txState](#)

30.4.3 Typedef Documentation

30.4.3.1 typedef void(* [uart_edma_transfer_callback_t](#))(UART_Type **base*,
[uart_edma_handle_t](#) **handle*, status_t *status*, void **userData*)

30.4.4 Function Documentation

30.4.4.1 void UART_TransferCreateHandleEDMA (UART_Type * *base*,
[uart_edma_handle_t](#) * *handle*, [uart_edma_transfer_callback_t](#) *callback*, void *
userData, [edma_handle_t](#) * *txEdmaHandle*, [edma_handle_t](#) * *rxEdmaHandle*)

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.
<i>callback</i>	UART callback, NULL means no callback.
<i>userData</i>	User callback function data.
<i>rxEdmaHandle</i>	User-requested DMA handle for RX DMA transfer.
<i>txEdmaHandle</i>	User-requested DMA handle for TX DMA transfer.

30.4.4.2 **status_t UART_SendEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uart_transfer_t * *xfer*)**

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

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>xfer</i>	UART eDMA transfer structure. See uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_TxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

30.4.4.3 **status_t UART_ReceiveEDMA (UART_Type * *base*, uart_edma_handle_t * *handle*, uart_transfer_t * *xfer*)**

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

Parameters

UART eDMA Driver

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.
<i>xfer</i>	UART eDMA transfer structure. See uart_transfer_t .

Return values

<i>kStatus_Success</i>	if succeeded; otherwise failed.
<i>kStatus_UART_RxBusy</i>	Previous transfer ongoing.
<i>kStatus_InvalidArgument</i>	Invalid argument.

30.4.4.4 void UART_TransferAbortSendEDMA (UART_Type * *base*, `uart_edma_handle_t` * *handle*)

This function aborts sent data using eDMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.

30.4.4.5 void UART_TransferAbortReceiveEDMA (UART_Type * *base*, `uart_edma_handle_t` * *handle*)

This function aborts receive data using eDMA.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	Pointer to the <code>uart_edma_handle_t</code> structure.

30.4.4.6 status_t UART_TransferGetSendCountEDMA (UART_Type * *base*, `uart_edma_handle_t` * *handle*, `uint32_t` * *count*)

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

Parameters

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

Return values

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

30.4.4.7 **status_t** UART_TransferGetReceiveCountEDMA (**UART_Type** * *base*, **uart_edma_handle_t** * *handle*, **uint32_t** * *count*)

This function gets the number of received bytes.

Parameters

<i>base</i>	UART peripheral base address.
<i>handle</i>	UART handle pointer.
<i>count</i>	Receive bytes count.

Return values

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

30.5 UART FreeRTOS Driver

30.5.1 Overview

Data Structures

- struct `uart_rtos_config_t`
UART configuration structure. [More...](#)

UART RTOS Operation

- int `UART_RTOS_Init` (`uart_rtos_handle_t` *handle, `uart_handle_t` *t_handle, const `uart_rtos_config_t` *cfg)
Initializes a UART instance for operation in RTOS.
- int `UART_RTOS_Deinit` (`uart_rtos_handle_t` *handle)
Deinitializes a UART instance for operation.

UART transactional Operation

- int `UART_RTOS_Send` (`uart_rtos_handle_t` *handle, const `uint8_t` *buffer, `uint32_t` length)
Sends data in the background.
- int `UART_RTOS_Receive` (`uart_rtos_handle_t` *handle, `uint8_t` *buffer, `uint32_t` length, `size_t` *received)
Receives data.

30.5.2 Data Structure Documentation

30.5.2.1 struct `uart_rtos_config_t`

Data Fields

- `UART_Type` * `base`
UART base address.
- `uint32_t` `srcclk`
UART source clock in Hz.
- `uint32_t` `baudrate`
Desired communication speed.
- `uart_parity_mode_t` `parity`
Parity setting.
- `uart_stop_bit_count_t` `stopbits`
Number of stop bits to use.
- `uint8_t` * `buffer`
Buffer for background reception.
- `uint32_t` `buffer_size`
Size of buffer for background reception.

30.5.3 Function Documentation

30.5.3.1 `int UART_RTOS_Init (uart_rtos_handle_t * handle, uart_handle_t * t_handle,
const uart_rtos_config_t * cfg)`

UART FreeRTOS Driver

Parameters

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

Returns

0 succeed; otherwise fail.

30.5.3.2 int UART_RTOS_Deinit (uart_rtos_handle_t * *handle*)

This function deinitializes the UART module, sets all register values to reset value, and frees the resources.

Parameters

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

30.5.3.3 int UART_RTOS_Send (uart_rtos_handle_t * *handle*, const uint8_t * *buffer*, uint32_t *length*)

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

Parameters

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

30.5.3.4 int UART_RTOS_Receive (uart_rtos_handle_t * *handle*, uint8_t * *buffer*, uint32_t *length*, size_t * *received*)

This function receives data from UART. It is a synchronous API. If data is immediately available, it is returned immediately and the number of bytes received.

Parameters

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

Chapter 31 Debug Console

31.1 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.

31.2 Function groups

31.2.1 Initialization

To initialize the debug console, call the `DbgConsole_Init()` function with these parameters. This function automatically enables the module and the clock.

```
/*
 * @brief Initializes the the peripheral used to debug messages.
 *
 * @param baseAddr      Indicates which address of the peripheral is used to send debug messages.
 * @param baudRate      The desired baud rate in bits per second.
 * @param device        Low level device type for the debug console, can be one of:
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_UART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPUART,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_LPSCI,
 *                      @arg DEBUG_CONSOLE_DEVICE_TYPE_USBCDC.
 * @param clkSrcFreq    Frequency of peripheral source clock.
 *
 * @return              Whether initialization was successful or not.
 */
status_t DbgConsole_Init(uint32_t baseAddr, uint32_t baudRate, uint8_t device, uint32_t clkSrcFreq)
```

Selects the supported debug console hardware device type, such as

```
DEBUG_CONSOLE_DEVICE_TYPE_NONE
DEBUG_CONSOLE_DEVICE_TYPE_LPSCI
DEBUG_CONSOLE_DEVICE_TYPE_UART
DEBUG_CONSOLE_DEVICE_TYPE_LPUART
DEBUG_CONSOLE_DEVICE_TYPE_USBCDC
```

After the initialization is successful, `stdout` and `stdin` are connected to the selected peripheral. The debug console state is stored in the `debug_console_state_t` structure, such as shown here.

```
typedef struct DebugConsoleState
{
    uint8_t          type;
    void*            base;
    debug_console_ops_t ops;
} debug_console_state_t;
```

Function groups

This example shows how to call the DbgConsole_Init() given the user configuration structure.

```
uint32_t uartClkSrcFreq = CLOCK_GetFreq (BOARD_DEBUG_UART_CLKSRC);  
  
DbgConsole_Init (BOARD_DEBUG_UART_BASEADDR, BOARD_DEBUG_UART_BAUDRATE, DEBUG_CONSOLE_DEVICE_TYPE_UART,  
                uartClkSrcFreq);
```

31.2.2 Advanced Feature

The debug console provides input and output functions to scan and print formatted data.

- Support a format specifier for PRINTF following this prototype " %[flags][width][.precision][length]specifier", which is explained below

flags	Description
-	Left-justified within the given field width. Right-justified is the default.
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is written, a blank space is inserted before the value.
#	Used with o, x, or X specifiers the value is preceded with 0, 0x, or 0X respectively for values other than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Width	Description
(number)	A minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E, and f specifiers this is the number of digits to be printed after the decimal point. For g and G specifiers This is the maximum number of significant digits to be printed. For s this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
Do not support	

specifier	Description
d or i	Signed decimal integer
f	Decimal floating point
F	Decimal floating point capital letters
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer capital letters
o	Signed octal
b	Binary value
p	Pointer address
u	Unsigned decimal integer
c	Character
s	String of characters
n	Nothing printed

Function groups

- Support a format specifier for SCANF following this prototype " %[*][width][length]specifier", which is explained below

*	Description
An optional starting asterisk indicates that the data is to be read from the stream but ignored. In other words, it is not stored in the corresponding argument.	

width	Description
This specifies the maximum number of characters to be read in the current reading operation.	

length	Description
hh	The argument is interpreted as a signed character or unsigned character (only applies to integer specifiers: i, d, o, u, x, and X).
h	The argument is interpreted as a short integer or unsigned short integer (only applies to integer specifiers: i, d, o, u, x, and X).
l	The argument is interpreted as a long integer or unsigned long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
ll	The argument is interpreted as a long long integer or unsigned long long integer for integer specifiers (i, d, o, u, x, and X) and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g, and G).
j or z or t	Not supported

specifier	Qualifying Input	Type of argument
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char *

specifier	Qualifying Input	Type of argument
i	Integer: : Number optionally preceded with a + or - sign	int *
d	Decimal integer: Number optionally preceded with a + or - sign	int *
a, A, e, E, f, F, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This reads subsequent characters until a white space is found (white space characters are considered to be blank, newline, and tab).	char *
u	Unsigned decimal integer.	unsigned int *

The debug console has its own printf/scanf/putchar/getchar functions which are defined in the header file.

```
int DbgConsole_Printf(const char *fmt_s, ...);
int DbgConsole_Putchar(int ch);
int DbgConsole_Scanf(const char *fmt_ptr, ...);
int DbgConsole_Getchar(void);
```

This utility supports selecting toolchain's printf/scanf or the MCUXpresso SDK printf/scanf.

```
#if SDK_DEBUGCONSOLE    /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF           DbgConsole_Printf
#define SCANF            DbgConsole_Scanf
#define PUTCHAR          DbgConsole_Putchar
#define GETCHAR          DbgConsole_Getchar
#else                   /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF           printf
#define SCANF            scanf
#define PUTCHAR          putchar
#define GETCHAR          getchar
#endif /* SDK_DEBUGCONSOLE */
```

31.3 Typical use case

Some examples use the PUTCHAR & GETCHAR function

```
ch = GETCHAR();
PUTCHAR(ch);
```

Typical use case

Some examples use the PRINTF function

Statement prints the string format.

```
PRINTF("%s %s\r\n", "Hello", "world!");
```

Statement prints the hexadecimal format/

```
PRINTF("0x%02X hexadecimal number equivalent 255", 255);
```

Statement prints the decimal floating point and unsigned decimal.

```
PRINTF("Execution timer: %s\n\rTime: %u ticks %2.5f milliseconds\n\rDONE\n\r", "1 day", 86400, 86.4);
```

Some examples use the SCANF function

```
PRINTF("Enter a decimal number: ");
SCANF("%d", &i);
PRINTF("\r\nYou have entered %d.\r\n", i, i);
PRINTF("Enter a hexadecimal number: ");
SCANF("%x", &i);
PRINTF("\r\nYou have entered 0x%X (%d).\r\n", i, i);
```

Print out failure messages using KSDK __assert_func:

```
void __assert_func(const char *file, int line, const char *func, const char *failedExpr)
{
    PRINTF("ASSERT ERROR \" %s \": file \"%s\" Line \"%d\" function name \"%s\" \n", failedExpr, file ,
        line, func);
    for (;;)
    {}
}
```

Note:

To use 'printf' and 'scanf' for GNUC Base, add file 'fsl_sbrk.c' in path: ..\{package}\devices\{subset}\utilities\fsl-_sbrk.c to your project.

Modules

- [Semihosting](#)

31.4 Semihosting

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism can be used, for example, to enable functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard of the host rather than having a screen and keyboard on the target system.

31.4.1 Guide Semihosting for IAR

NOTE: After the setting both "printf" and "scanf" are available for debugging.

Step 1: Setting up the environment

1. To set debugger options, choose Project>Options. In the Debugger category, click the Setup tab.
2. Select Run to main and click OK. This ensures that the debug session starts by running the main function.
3. The project is now ready to be built.

Step 2: Building the project

1. Compile and link the project by choosing Project>Make or F7.
2. Alternatively, click the Make button on the tool bar. The Make command compiles and links those files that have been modified.

Step 3: Starting semihosting

1. Choose "Semihosting_IAR" project -> "Options" -> "Debugger" -> "J-Link/J-Trace".
2. Choose tab "J-Link/J-Trace" -> "Connection" tab -> "SWD".
3. Start the project by choosing Project>Download and Debug.
4. Choose View>Terminal I/O to display the output from the I/O operations.

31.4.2 Guide Semihosting for Keil μ Vision

NOTE: Keil supports Semihosting only for Cortex-M3/Cortex-M4 cores.

Step 1: Prepare code

Remove function `fputc` and `fgetc` is used to support KEIL in "fsl_debug_console.c" and add the following code to project.

```
#pragma import(__use_no_semihosting_swi)

volatile int ITM_RxBuffer = ITM_RXBUFFER_EMPTY;    /* used for Debug Input */
```

Semihosting

```
struct __FILE
{
    int handle;
};
FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f)
{
    return (ITM_SendChar(ch));
}

int fgetc(FILE *f)
{
    /* blocking */
    while (ITM_CheckChar() != 1)
        ;
    return (ITM_ReceiveChar());
}

int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}

void _ttywrch(int ch)
{
    ITM_SendChar(ch);
}

void _sys_exit(int return_code)
{
label:
    goto label; /* endless loop */
}
```

Step 2: Setting up the environment

1. In menu bar, choose Project>Options for target or using Alt+F7 or click.
2. Select "Target" tab and not select "Use MicroLIB".
3. Select "Debug" tab, select "J-Link/J-Trace Cortex" and click "Setting button".
4. Select "Debug" tab and choose Port:SW, then select "Trace" tab, choose "Enable" and click OK.

Step 3: Building the project

1. Compile and link the project by choosing Project>Build Target or using F7.

Step 4: Building the project

1. Choose "Debug" on menu bar or Ctrl F5.
2. In menu bar, choose "Serial Window" and click to "Debug (printf) Viewer".
3. Run line by line to see result in Console Window.

31.4.3 Guide Semihosting for KDS

NOTE: After the setting use "printf" for debugging.

Step 1: Setting up the environment

1. In menu bar, choose Project>Properties>C/C++ Build>Settings>Tool Settings.
2. Select "Libraries" on "Cross ARM C Linker" and delete "nosys".
3. Select "Miscellaneous" on "Cross ARM C Linker", add "-specs=rdimon.specs" to "Other link flages" and tick "Use newlib-nano", and click OK.

Step 2: Building the project

1. In menu bar, choose Project>Build Project.

Step 3: Starting semihosting

1. In Debug configurations, choose "Startup" tab, tick "Enable semihosting and Telnet". Press "Apply" and "Debug".
2. After clicking Debug, the Window is displayed same as below. Run line by line to see the result in the Console Window.

31.4.4 Guide Semihosting for ATL

NOTE: J-Link has to be used to enable semihosting.

Step 1: Prepare code

Add the following code to the project.

```
int _write(int file, char *ptr, int len)
{
    /* Implement your write code here. This is used by puts and printf. */
    int i=0;
    for(i=0 ; i<len ; i++)
        ITM_SendChar((*ptr++));
    return len;
}
```

Step 2: Setting up the environment

1. In menu bar, choose Debug Configurations. In tab "Embedded C/C++ Application" choose "- Semihosting_ATL_xxx debug J-Link".
2. In tab "Debugger" set up as follows.
 - JTAG mode must be selected

Semihosting

- SWV tracing must be enabled
 - Enter the Core Clock frequency, which is hardware board-specific.
 - Enter the desired SWO Clock frequency. The latter depends on the JTAG Probe and must be a multiple of the Core Clock value.
3. Click "Apply" and "Debug".

Step 3: Starting semihosting

1. In the Views menu, expand the submenu SWV and open the docking view "SWV Console". 2. Open the SWV settings panel by clicking the "Configure Serial Wire Viewer" button in the SWV Console view toolbar. 3. Configure the data ports to be traced by enabling the ITM channel 0 check-box in the ITM stimulus ports group: Choose "EXETRC: Trace Exceptions" and In tab "ITM Stimulus Ports" choose "Enable Port" 0. Then click "OK".
2. It is recommended not to enable other SWV trace functionalities at the same time because this may over use the SWO pin causing packet loss due to a limited bandwidth (certain other SWV tracing capabilities can send a lot of data at very high-speed). Save the SWV configuration by clicking the OK button. The configuration is saved with other debug configurations and remains effective until changed.
3. Press the red Start/Stop Trace button to send the SWV configuration to the target board to enable SWV trace recoding. The board does not send any SWV packages until it is properly configured. The SWV Configuration must be present, if the configuration registers on the target board are reset. Also, tracing does not start until the target starts to execute.
4. Start the target execution again by pressing the green Resume Debug button.
5. The SWV console now shows the printf() output.

31.4.5 Guide Semihosting for ARMGCC

Step 1: Setting up the environment

1. Turn on "J-LINK GDB Server" -> Select suitable "Target device" -> "OK".
2. Turn on "PuTTY". Set up as follows.
 - "Host Name (or IP address)" : localhost
 - "Port" :2333
 - "Connection type" : Telet.
 - Click "Open".
3. Increase "Heap/Stack" for GCC to 0x2000:

Add to "CMakeLists.txt"

```
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}
--defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --
defsym=__stack_size__=0x2000")

SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG} --
```

```
defsym=__heap_size__=0x2000")
SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "${CMAKE_EXE_LINKER_FLAGS_RELEASE}
--defsym=__heap_size__=0x2000")
```

Step 2: Building the project

1. Change "CMakeLists.txt":

Change "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "\${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=nano.specs")"

to "SET(CMAKE_EXE_LINKER_FLAGS_RELEASE "\${CMAKE_EXE_LINKER_FLAGS_RELEASE} -specs=rdimon.specs")"

Replace paragraph

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-common")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffunction-sections")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fdata-sections")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-ffreestanding")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-fno-builtin")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mthumb")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-mapcs")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
--gc-sections")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-static")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-z")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
-Xlinker")
```

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
muldefs")
```

To

```
SET(CMAKE_EXE_LINKER_FLAGS_DEBUG "${CMAKE_EXE_LINKER_FLAGS_DEBUG}
```

Semihosting

```
G} --specs=rdimon.specs ")
```

Remove

```
target_link_libraries(semihosting_ARMGCC.elf debug nosys)
```

2. Run "build_debug.bat" to build project

Step 3: Starting semihosting

- (a) Download the image and set as follows.

```
cd D:\mcu-sdk-2.0-origin\boards\twrk64f120m\driver_examples\semihosting\armgcc\debug
d:
C:\PROGRA~2\GNUTOO~1\4BD65~1.920\bin\arm-none-eabi-gdb.exe
target remote localhost:2331
monitor reset
monitor semihosting enable
monitor semihosting thumbSWI 0xAB
monitor semihosting IOClient 1
monitor flash device = MK64FN1M0xxx12
load semihosting_ARMGCC.elf
monitor reg pc = (0x00000004)
monitor reg sp = (0x00000000)
continue
```

- (b) After the setting, press "enter". The PuTTY window now shows the printf() output.

Chapter 32

Notification Framework

32.1 Overview

This section describes the programming interface of the Notifier driver.

32.2 Notifier Overview

The Notifier provides a configuration dynamic change service. Based on this service, applications can switch between pre-defined configurations. The Notifier enables drivers and applications to register callback functions to this framework. Each time that the configuration is changed, drivers and applications receive a notification and change their settings. To simplify, the Notifier only supports the static callback registration. This means that, for applications, all callback functions are collected into a static table and passed to the Notifier.

These are the steps for the configuration transition.

1. Before configuration transition, the Notifier sends a "BEFORE" message to the callback table. When this message is received, IP drivers should check whether any current processes can be stopped and stop them. If the processes cannot be stopped, the callback function returns an error.
The Notifier supports two types of transition policies, a graceful policy and a forceful policy. When the graceful policy is used, if some callbacks return an error while sending a "BEFORE" message, the configuration transition stops and the Notifier sends a "RECOVER" message to all drivers that have stopped. Then, these drivers can recover the previous status and continue to work. When the forceful policy is used, drivers are stopped forcefully.
2. After the "BEFORE" message is processed successfully, the system switches to the new configuration.
3. After the configuration changes, the Notifier sends an "AFTER" message to the callback table to notify drivers that the configuration transition is finished.

This example shows how to use the Notifier in the Power Manager application.

```
#include "fsl_notifier.h"

// Definition of the Power Manager callback.
status_t callback0(notifier_notification_block_t *notify, void *data)
{
    status_t ret = kStatus_Success;

    ...
    ...
    ...

    return ret;
}

// Definition of the Power Manager user function.
status_t APP_PowerModeSwitch(notifier_user_config_t *targetConfig, void *userData)
{

```

Notifier Overview

```
...
...
...
}
...
...
...
...
...
// Main function.
int main(void)
{
    // Define a notifier handle.
    notifier_handle_t powerModeHandle;

    // Callback configuration.
    user_callback_data_t callbackData0;

    notifier_callback_config_t callbackCfg0 = {callback0,
        kNOTIFIER_CallbackBeforeAfter,
        (void *)&callbackData0};

    notifier_callback_config_t callbacks[] = {callbackCfg0};

    // Power mode configurations.
    power_user_config_t vlprConfig;
    power_user_config_t stopConfig;

    notifier_user_config_t *powerConfigs[] = {&vlprConfig, &stopConfig};

    // Definition of a transition to and out the power modes.
    vlprConfig.mode = kAPP_PowerModeVlpr;
    vlprConfig.enableLowPowerWakeUpOnInterrupt = false;

    stopConfig = vlprConfig;
    stopConfig.mode = kAPP_PowerModeStop;

    // Create Notifier handle.
    NOTIFIER_CreateHandle(&powerModeHandle, powerConfigs, 2U, callbacks, 1U,
        APP_PowerModeSwitch, NULL);
    ...
    ...
    // Power mode switch.
    NOTIFIER_switchConfig(&powerModeHandle, targetConfigIndex,
        kNOTIFIER_PolicyAgreement);
}
```

Data Structures

- struct `notifier_notification_block_t`
notification block passed to the registered callback function. [More...](#)
- struct `notifier_callback_config_t`
Callback configuration structure. [More...](#)
- struct `notifier_handle_t`
Notifier handle structure. [More...](#)

Typedefs

- typedef void `notifier_user_config_t`
Notifier user configuration type.
- typedef status_t(* `notifier_user_function_t`)(`notifier_user_config_t` *targetConfig, void *userData)
Notifier user function prototype Use this function to execute specific operations in configuration switch.

- typedef status_t(* [notifier_callback_t](#))([notifier_notification_block_t](#) *notify, void *data)
Callback prototype.

Enumerations

- enum [_notifier_status](#) {
 [kStatus_NOTIFIER_ErrorNotificationBefore](#),
 [kStatus_NOTIFIER_ErrorNotificationAfter](#) }
Notifier error codes.
- enum [notifier_policy_t](#) {
 [kNOTIFIER_PolicyAgreement](#),
 [kNOTIFIER_PolicyForcible](#) }
Notifier policies.
- enum [notifier_notification_type_t](#) {
 [kNOTIFIER_NotifyRecover](#) = 0x00U,
 [kNOTIFIER_NotifyBefore](#) = 0x01U,
 [kNOTIFIER_NotifyAfter](#) = 0x02U }
Notification type.
- enum [notifier_callback_type_t](#) {
 [kNOTIFIER_CallbackBefore](#) = 0x01U,
 [kNOTIFIER_CallbackAfter](#) = 0x02U,
 [kNOTIFIER_CallbackBeforeAfter](#) = 0x03U }
The callback type, which indicates kinds of notification the callback handles.

Functions

- status_t [NOTIFIER_CreateHandle](#) ([notifier_handle_t](#) *notifierHandle, [notifier_user_config_t](#) **configs, uint8_t configsNumber, [notifier_callback_config_t](#) *callbacks, uint8_t callbacksNumber, [notifier_user_function_t](#) userFunction, void *userData)
Creates a Notifier handle.
- status_t [NOTIFIER_SwitchConfig](#) ([notifier_handle_t](#) *notifierHandle, uint8_t configIndex, [notifier-_policy_t](#) policy)
Switches the configuration according to a pre-defined structure.
- uint8_t [NOTIFIER_GetErrorCallbackIndex](#) ([notifier_handle_t](#) *notifierHandle)
This function returns the last failed notification callback.

32.3 Data Structure Documentation

32.3.1 struct [notifier_notification_block_t](#)

Data Fields

- [notifier_user_config_t](#) * [targetConfig](#)
Pointer to target configuration.
- [notifier_policy_t](#) [policy](#)
Configure transition policy.
- [notifier_notification_type_t](#) [notifyType](#)
Configure notification type.

Data Structure Documentation

32.3.1.0.0.67 Field Documentation

32.3.1.0.0.67.1 `notifier_user_config_t* notifier_notification_block_t::targetConfig`

32.3.1.0.0.67.2 `notifier_policy_t notifier_notification_block_t::policy`

32.3.1.0.0.67.3 `notifier_notification_type_t notifier_notification_block_t::notifyType`

32.3.2 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.

32.3.2.0.0.68 Field Documentation

32.3.2.0.0.68.1 `notifier_callback_t notifier_callback_config_t::callback`

32.3.2.0.0.68.2 `notifier_callback_type_t notifier_callback_config_t::callbackType`

32.3.2.0.0.68.3 `void* notifier_callback_config_t::callbackData`

32.3.3 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.

32.3.3.0.0.69 Field Documentation

32.3.3.0.0.69.1 `notifier_user_config_t** notifier_handle_t::configsTable`

32.3.3.0.0.69.2 `uint8_t notifier_handle_t::configsNumber`

32.3.3.0.0.69.3 `notifier_callback_config_t* notifier_handle_t::callbacksTable`

32.3.3.0.0.69.4 `uint8_t notifier_handle_t::callbacksNumber`

32.3.3.0.0.69.5 `uint8_t notifier_handle_t::errorCallbackIndex`

32.3.3.0.0.69.6 `uint8_t notifier_handle_t::currentConfigIndex`

32.3.3.0.0.69.7 `notifier_user_function_t notifier_handle_t::userFunction`

32.3.3.0.0.69.8 `void* notifier_handle_t::userData`

32.4 Typedef Documentation

32.4.1 `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.

32.4.2 `typedef status_t(* notifier_user_function_t)(notifier_user_config_t *targetConfig, void *userData)`

Before and after this function execution, different notification is sent to registered callbacks. If this function returns any error code, `NOTIFIER_SwitchConfig()` exits.

Parameters

Enumeration Type Documentation

<i>targetConfig</i>	target Configuration.
<i>userData</i>	Refers to other specific data passed to user function.

Returns

An error code or `kStatus_Success`.

32.4.3 `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`.

32.5 Enumeration Type Documentation

32.5.1 `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.

32.5.2 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.

32.5.3 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.

32.5.4 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.

32.6 Function Documentation

32.6.1 `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.

32.6.2 **status_t NOTIFIER_SwitchConfig (notifier_handle_t * *notifierHandle*, uint8_t *configIndex*, notifier_policy_t *policy*)**

This function sets the system to the target configuration. Before transition, the Notifier sends notifications to all callbacks registered to the callback table. Callbacks are invoked in the following order: All registered callbacks are notified ordered by index in the callbacks array. The same order is used for before and after switch notifications. The notifications before the configuration switch can be used to obtain confirmation about the change from registered callbacks. If any registered callback denies the configuration change, further execution of this function depends on the notifier policy: the configuration change is either forced (kNOTIFIER_PolicyForcible) or exited (kNOTIFIER_PolicyAgreement). When configuration change is forced, the result of the before switch notifications are ignored. If an agreement is required, if any callback returns an error code, further notifications before switch notifications are cancelled and all already notified callbacks are re-invoked. The index of the callback which returned error code during pre-switch notifications is stored (any error codes during callbacks re-invocation are ignored) and NOTIFIER_GetErrorCallback() can be used to get it. Regardless of the policies, if any callback returns an error code, an error code indicating in which phase the error occurred is returned when [NOTIFIER_SwitchConfig\(\)](#) exits.

Parameters

Function Documentation

<i>notifierHandle</i>	pointer to notifier handle
<i>configIndex</i>	Index of the target configuration.
<i>policy</i>	Transaction policy, kNOTIFIER_PolicyAgreement or kNOTIFIER_PolicyForcible.

Returns

An error code or kStatus_Success.

32.6.3 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 33 Shell

33.1 Overview

This part describes the programming interface of the Shell middleware. Shell controls MCUs by commands via the specified communication peripheral based on the debug console driver.

33.2 Function groups

33.2.1 Initialization

To initialize the Shell middleware, call the [SHELL_Init\(\)](#) function with these parameters. This function automatically enables the middleware.

```
void SHELL_Init(p_shell_context_t context, send_data_cb_t send_cb,  
               recv_data_cb_t recv_cb, char *prompt);
```

Then, after the initialization was successful, call a command to control MCUs.

This example shows how to call the [SHELL_Init\(\)](#) given the user configuration structure.

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");
```

33.2.2 Advanced Feature

- Support to get a character from standard input devices.

```
static uint8_t GetChar(p_shell_context_t context);
```

Commands	Description
Help	Lists all commands which are supported by Shell.
Exit	Exits the Shell program.
strCompare	Compares the two input strings.

Input character	Description
A	Gets the latest command in the history.
B	Gets the first command in the history.
C	Replaces one character at the right of the pointer.

Function groups

Input character	Description
D	Replaces one character at the left of the pointer.
	Run AutoComplete function
	Run cmdProcess function
	Clears a command.

33.2.3 Shell Operation

```
SHELL_Init(&user_context, SHELL_SendDataCallback, SHELL_ReceiveDataCallback, "SHELL>> ");  
SHELL_Main(&user_context);
```

Data Structures

- struct [p_shell_context_t](#)
Data structure for Shell environment. [More...](#)
- struct [shell_command_context_t](#)
User command data structure. [More...](#)
- struct [shell_command_context_list_t](#)
Structure list command. [More...](#)

Macros

- #define [SHELL_USE_HISTORY](#) (0U)
Macro to set on/off history feature.
- #define [SHELL_SEARCH_IN_HIST](#) (1U)
Macro to set on/off history feature.
- #define [SHELL_USE_FILE_STREAM](#) (0U)
Macro to select method stream.
- #define [SHELL_AUTO_COMPLETE](#) (1U)
Macro to set on/off auto-complete feature.
- #define [SHELL_BUFFER_SIZE](#) (64U)
Macro to set console buffer size.
- #define [SHELL_MAX_ARGS](#) (8U)
Macro to set maximum arguments in command.
- #define [SHELL_HIST_MAX](#) (3U)
Macro to set maximum count of history commands.
- #define [SHELL_MAX_CMD](#) (20U)
Macro to set maximum count of commands.
- #define [SHELL_OPTIONAL_PARAMS](#) (0xFF)
Macro to bypass arguments check.

Typedefs

- typedef void(* [send_data_cb_t](#))(uint8_t *buf, uint32_t len)
Shell user send data callback prototype.
- typedef void(* [recv_data_cb_t](#))(uint8_t *buf, uint32_t len)

- *Shell user receiver data callback prototype.*
typedef int(* [printf_data_t](#))(const char *format,...)
- *Shell user printf data prototype.*
typedef int32_t(* [cmd_function_t](#))(p_shell_context_t context, int32_t argc, char **argv)
- *User command function prototype.*

Enumerations

- enum [fun_key_status_t](#) {
 [kSHELL_Normal](#) = 0U,
 [kSHELL_Special](#) = 1U,
 [kSHELL_Function](#) = 2U }
 A type for the handle special key.

Shell functional operation

- void [SHELL_Init](#) (p_shell_context_t context, [send_data_cb_t](#) send_cb, [recv_data_cb_t](#) recv_cb, [printf_data_t](#) shell_printf, char *prompt)
 Enables the clock gate and configures the Shell module according to the configuration structure.
- int32_t [SHELL_RegisterCommand](#) (const [shell_command_context_t](#) *command_context)
 Shell register command.
- int32_t [SHELL_Main](#) (p_shell_context_t context)
 Main loop for Shell.

33.3 Data Structure Documentation

33.3.1 struct shell_context_struct

Data Fields

- char * [prompt](#)
 Prompt string.
- enum [_fun_key_status](#) [stat](#)
 Special key status.
- char [line](#) [[SHELL_BUFFER_SIZE](#)]
 Consult buffer.
- uint8_t [cmd_num](#)
 Number of user commands.
- uint8_t [l_pos](#)
 Total line position.
- uint8_t [c_pos](#)
 Current line position.
- [send_data_cb_t](#) [send_data_func](#)
 Send data interface operation.
- [recv_data_cb_t](#) [recv_data_func](#)
 Receive data interface operation.
- uint16_t [hist_current](#)
 Current history command in hist buff.
- uint16_t [hist_count](#)

Data Structure Documentation

- *Total history command in hist buff.*
char [hist_buf](#) [SHELL_HIST_MAX][SHELL_BUFFER_SIZE]
- *History buffer.*
bool [exit](#)
Exit Flag.

33.3.2 struct shell_command_context_t

Data Fields

- const char * [pcCommand](#)
The command that is executed.
- char * [pcHelpString](#)
String that describes how to use the command.
- const [cmd_function_t](#) [pFuncCallBack](#)
A pointer to the callback function that returns the output generated by the command.
- uint8_t [cExpectedNumberOfParameters](#)
Commands expect a fixed number of parameters, which may be zero.

33.3.2.0.0.70 Field Documentation

33.3.2.0.0.70.1 const char* shell_command_context_t::pcCommand

For example "help". It must be all lower case.

33.3.2.0.0.70.2 char* shell_command_context_t::pcHelpString

It should start with the command itself, and end with "\r\n". For example "help: Returns a list of all the commands\r\n".

33.3.2.0.0.70.3 const cmd_function_t shell_command_context_t::pFuncCallBack

33.3.2.0.0.70.4 uint8_t shell_command_context_t::cExpectedNumberOfParameters

33.3.3 struct shell_command_context_list_t

Data Fields

- const [shell_command_context_t](#) * [CommandList](#) [SHELL_MAX_CMD]
The command table list.
- uint8_t [numberOfCommandInList](#)
The total command in list.

33.4 Macro Definition Documentation

33.4.1 `#define SHELL_USE_HISTORY (0U)`

33.4.2 `#define SHELL_SEARCH_IN_HIST (1U)`

33.4.3 `#define SHELL_USE_FILE_STREAM (0U)`

33.4.4 `#define SHELL_AUTO_COMPLETE (1U)`

33.4.5 `#define SHELL_BUFFER_SIZE (64U)`

33.4.6 `#define SHELL_MAX_ARGS (8U)`

33.4.7 `#define SHELL_HIST_MAX (3U)`

33.4.8 `#define SHELL_MAX_CMD (20U)`

33.5 Typedef Documentation

33.5.1 `typedef void(* send_data_cb_t)(uint8_t *buf, uint32_t len)`

33.5.2 `typedef void(* recv_data_cb_t)(uint8_t *buf, uint32_t len)`

33.5.3 `typedef int(* printf_data_t)(const char *format,...)`

33.5.4 `typedef int32_t(* cmd_function_t)(p_shell_context_t context, int32_t argc, char **argv)`

33.6 Enumeration Type Documentation

33.6.1 `enum fun_key_status_t`

Enumerator

kSHELL_Normal Normal key.

kSHELL_Special Special key.

kSHELL_Function Function key.

33.7 Function Documentation

33.7.1 void SHELL_Init (p_shell_context_t *context*, send_data_cb_t *send_cb*, recv_data_cb_t *recv_cb*, printf_data_t *shell_printf*, char * *prompt*)

This function must be called before calling all other Shell functions. Call operation the Shell commands with user-defined settings. The example below shows how to set up the middleware Shell and how to call the SHELL_Init function by passing in these parameters. This is an example.

```
*  shell_context_struct user_context;
*  SHELL_Init(&user_context, SendDataFunc, ReceiveDataFunc, "SHELL>> ");
*
```

Parameters

<i>context</i>	The pointer to the Shell environment and runtime states.
<i>send_cb</i>	The pointer to call back send data function.
<i>recv_cb</i>	The pointer to call back receive data function.
<i>prompt</i>	The string prompt of Shell

33.7.2 int32_t SHELL_RegisterCommand (const shell_command_context_t * *command_context*)

Parameters

<i>command_ - context</i>	The pointer to the command data structure.
---------------------------	--

Returns

-1 if error or 0 if success

33.7.3 int32_t SHELL_Main (p_shell_context_t *context*)

Main loop for Shell; After this function is called, Shell begins to initialize the basic variables and starts to work.

Parameters

<i>context</i>	The pointer to the Shell environment and runtime states.
----------------	--

Returns

This function does not return until Shell command exit was called.

Chapter 34 DMA Manager

34.1 Overview

DMA Manager provides a series of functions to manage the DMAMUX instances and channels.

34.2 Function groups

34.2.1 DMAMGR Initialization and De-initialization

This function group initializes and deinitializes the DMA Manager.

34.2.2 DMAMGR Operation

This function group requests/releases the DMAMUX channel and configures the channel request source.

34.3 Typical use case

34.3.1 DMAMGR static channel allocation

```
uint8_t channel;
dmamanager_handle_t dmamanager_handle;

/* Initialize DMAMGR */
DMAMGR_Init(&dmamanager_handle, EXAMPLE_DMA_BASEADDR, DMA_CHANNEL_NUMBER, startChannel);
/* Request a DMAMUX channel by static allocate mechanism */
channel = kDMAMGR_STATIC_ALLOCATE;
DMAMGR_RequestChannel(&dmamanager_handle, kDmaRequestMux0AlwaysOn63, channel, &handle)
;
```

34.3.2 DMAMGR dynamic channel allocation

```
uint8_t channel;
dmamanager_handle_t dmamanager_handle;

/* Initialize DMAMGR */
DMAMGR_Init(&dmamanager_handle, EXAMPLE_DMA_BASEADDR, DMA_CHANNEL_NUMBER, startChannel);
/* Request a DMAMUX channel by Dynamic allocate mechanism */
channel = DMAMGR_DYNAMIC_ALLOCATE;
DMAMGR_RequestChannel(&dmamanager_handle, kDmaRequestMux0AlwaysOn63, channel, &handle)
;
```

Data Structures

- struct [dmamanager_handle_t](#)
dmamanager handle typedef. [More...](#)

Data Structure Documentation

Macros

- #define [DMAMGR_DYNAMIC_ALLOCATE](#) 0xFFU
Dynamic channel allocation mechanism.

Enumerations

- enum [_dma_manager_status](#) {
 [kStatus_DMAMGR_ChannelOccupied](#) = MAKE_STATUS(kStatusGroup_DMAMGR, 0),
 [kStatus_DMAMGR_ChannelNotUsed](#) = MAKE_STATUS(kStatusGroup_DMAMGR, 1),
 [kStatus_DMAMGR_NoFreeChannel](#) = MAKE_STATUS(kStatusGroup_DMAMGR, 2) }
DMA manager status.

DMAMGR Initialization and De-initialization

- void [DMAMGR_Init](#) ([dmamanager_handle_t](#) *dmamanager_handle, DMA_Type *dma_base, uint32_t channelNum, uint32_t startChannel)
Initializes the DMA manager.
- void [DMAMGR_Deinit](#) ([dmamanager_handle_t](#) *dmamanager_handle)
Deinitializes the DMA manager.

DMAMGR Operation

- status_t [DMAMGR_RequestChannel](#) ([dmamanager_handle_t](#) *dmamanager_handle, uint32_t requestSource, uint32_t channel, void *handle)
Requests a DMA channel.
- status_t [DMAMGR_ReleaseChannel](#) ([dmamanager_handle_t](#) *dmamanager_handle, void *handle)
Releases a DMA channel.
- bool [DMAMGR_IsChannelOccupied](#) ([dmamanager_handle_t](#) *dmamanager_handle, uint32_t channel)
Get a DMA channel status.

34.4 Data Structure Documentation

34.4.1 struct dmamanager_handle_t

Note

The contents of this structure are private and subject to change.

This dma manager handle structure is used to store the parameters transferred by users. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.

Data Fields

- void * [dma_base](#)
Peripheral DMA instance.
- uint32_t [channelNum](#)

- *Channel numbers for the DMA instance which need to be managed by dma manager.*
uint32_t [startChannel](#)
The start channel that can be managed by dma manager,users need to transfer it with a certain number or NULL.
- bool [s_DMAMGR_Channels](#) [64]
The s_DMAMGR_Channels is used to store dma manager state.
- uint32_t [DmamuxInstanceStart](#)
The DmamuxInstance is used to calculate the DMAMUX Instance according to the DMA Instance.
- uint32_t [multiple](#)
The multiple is used to calculate the multiple between DMAMUX count and DMA count.

34.4.1.0.0.71 Field Documentation

34.4.1.0.0.71.1 void* dmamanager_handle_t::dma_base

34.4.1.0.0.71.2 uint32_t dmamanager_handle_t::channelNum

34.4.1.0.0.71.3 uint32_t dmamanager_handle_t::startChannel

34.4.1.0.0.71.4 bool dmamanager_handle_t::s_DMAMGR_Channels[64]

34.4.1.0.0.71.5 uint32_t dmamanager_handle_t::DmamuxInstanceStart

34.4.1.0.0.71.6 uint32_t dmamanager_handle_t::multiple

34.5 Macro Definition Documentation

34.5.1 #define DMAMGR_DYNAMIC_ALLOCATE 0xFFU

34.6 Enumeration Type Documentation

34.6.1 enum _dma_manager_status

Enumerator

kStatus_DMAMGR_ChannelOccupied Channel has been occupied.

kStatus_DMAMGR_ChannelNotUsed Channel has not been used.

kStatus_DMAMGR_NoFreeChannel All channels have been occupied.

34.7 Function Documentation

34.7.1 void DMAMGR_Init (dmamanager_handle_t * *dmamanager_handle*,
DMA_Type * *dma_base*, uint32_t *channelNum*, uint32_t *startChannel*)

This function initializes the DMA manager, ungates the DMAMUX clocks, and initializes the eDMA or DMA peripherals.

Function Documentation

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal, users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>dma_base</i>	Peripheral DMA instance base pointer.
<i>dmamux_base</i>	Peripheral DMAMUX instance base pointer.
<i>channelNum</i>	Channel numbers for the DMA instance which need to be managed by dma manager.
<i>startChannel</i>	The start channel that can be managed by dma manager.

34.7.2 void DMAMGR_Deinit (dmamanager_handle_t * *dmamanager_handle*)

This function deinitializes the DMA manager, disables the DMAMUX channels, gates the DMAMUX clocks, and deinitializes the eDMA or DMA peripherals.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal, users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
--------------------------	--

34.7.3 status_t DMAMGR_RequestChannel (dmamanager_handle_t * *dmamanager_handle*, uint32_t *requestSource*, uint32_t *channel*, void * *handle*)

This function requests a DMA channel which is not occupied. The two channels to allocate the mechanism are dynamic and static channels. For the dynamic allocation mechanism (channe = DMAMGR_DYNAMIC_ALLOCATE), DMAMGR allocates a DMA channel according to the given request source and start-channel and then configures it. For static allocation mechanism, DMAMGR configures the given channel according to the given request source and channel number.

Parameters

<i>dmamanager_-handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal, users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>requestSource</i>	DMA channel request source number. See the soc.h, see the enum dma_request_source_t
<i>channel</i>	The channel number users want to occupy. If using the dynamic channel allocate mechanism, set the channel equal to DMAMGR_DYNAMIC_ALLOCATE.
<i>handle</i>	DMA or eDMA handle pointer.

Return values

<i>kStatus_Success</i>	In a dynamic/static channel allocation mechanism, allocate the DMAMUX channel successfully.
<i>kStatus_DMAMGR_NoFreeChannel</i>	In a dynamic channel allocation mechanism, all DMAMUX channels are occupied.
<i>kStatus_DMAMGR_ChannelOccupied</i>	In a static channel allocation mechanism, the given channel is occupied.

34.7.4 status_t DMAMGR_ReleaseChannel (dmamanager_handle_t * dmamanager_handle, void * handle)

This function releases an occupied DMA channel.

Parameters

<i>dmamanager_-handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal, users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>handle</i>	DMA or eDMA handle pointer.

Return values

Function Documentation

<i>kStatus_Success</i>	Releases the given channel successfully.
<i>kStatus_DMAMGR_ChannelNotUsed</i>	The given channel to be released had not been used before.

34.7.5 bool DMAMGR_IsChannelOccupied (dmamanager_handle_t * dmamanager_handle, uint32_t channel)

This function get a DMA channel status. Return 0 indicates the channel has not been used, return 1 indicates the channel has been occupied.

Parameters

<i>dmamanager_handle</i>	DMA manager handle pointer, this structure is maintained by dma manager internal, users only need to transfer the structure to the function. And users shall not free the memory before calling DMAMGR_Deinit, also shall not modify the contents of the memory.
<i>channel</i>	The channel number that users want get its status.

Chapter 35

Memory-Mapped Cryptographic Acceleration Unit (MMCAU)

35.1 Overview

The mmCAU software library uses the mmCAU co-processor that is connected to the ARM Cortex-M4/-M0+ Private Peripheral Bus (PPB). In this chapter, CAU refers to both CAU and mmCAU unless explicitly noted.

35.2 Purpose

The following chapter describes how to use the mmCAU software library in any application to integrate a cryptographic algorithm or hashing function supported by the software library. NXP products supported by the software library are MCU/MPUs. Check the specific Freescale product for CAU availability.

35.3 Library Features

The library is as compact and generic as possible to simplify the integration with existing cryptographic software. The library has a standard header file with ANSI C prototypes for all functions: "cau_api.h". This software library is thread safe only if CAU registers are saved on a context switch. The mmCAU software library is also compatible to ARM C compiler conventions (EABI). All pointers passed to mmCAU API functions (input and output data blocks, keys, key schedules, and so on) are aligned to 0-modulo-4 addresses.

For applications that don't need to deal with the aligned addresses, a simple wrapper layer is provided. The wrapper layer consists of the "fsl_mmcau.h" header file and "fsl_mmcau.c" source code file. The only function of the wrapper layer is that it supports unaligned addresses

. The CAU library supports the following encryption/decryption algorithms and hashing functions:

- AES128
- AES192
- AES256
- DES
- MD5
- SHA1
- SHA256

Note: 3DES crypto algorithms are supported by calling the corresponding DES crypto function three times. Hardware support for SHA256 is only present in the CAU version 2. See the appropriate MCU/MPU reference manual for details about availability. Additionally, the [cau_sha256_initialize_output\(\)](#) function checks the hardware revision and returns a (-1) value if the CAU lacks SHA256 support.

35.4 CAU and mmCAU software library overview

Table 1 shows the crypto algorithms and hashing functions included in the software library:

mmCAU software library usage

Crypto Algorithms	AES128 AES192 AES256	cau_aes_set_key
		cau_aes_encrypt
		cau_aes_decrypt
	DES/3DES	cau_des_chk_parity
		cau_des_encrypt
		cau_des_decrypt
Hashing Functions	MD5	cau_md5_initialize_output
		cau_md5_hash_n
		cau_md5_update
		cau_md5_hash
	SHA1	cau_sha1_initialize_output
		cau_sha1_hash_n
		cau_sha1_update
		cau_sha1_hash
	SHA256	cau_sha256_initialize_output
		cau_sha256_hash_n
		cau_sha256_update
		cau_sha256_hash

Table 1: Library Overview

35.5 mmCAU software library usage

The software library contains the following files:

File	Description
cau_api.h	CAU and mmCAU header file
lib_mmcau.a	mmCAU library

Table 2: File Description

The header file and lib_mmcau.a must always be included in the project.

Functions

- void [cau_aes_set_key](#) (const unsigned char *key, const int key_size, unsigned char *key_sch)
AES: Performs an AES key expansion.
- void [cau_aes_encrypt](#) (const unsigned char *in, const unsigned char *key_sch, const int nr, unsigned char *out)

- AES: Encrypts a single 16 byte block.*

 - void `cau_aes_decrypt` (const unsigned char *in, const unsigned char *key_sch, const int nr, unsigned char *out)
- AES: Decrypts a single 16-byte block.*

 - int `cau_des_chk_parity` (const unsigned char *key)
- DES: Checks key parity.*

 - void `cau_des_encrypt` (const unsigned char *in, const unsigned char *key, unsigned char *out)
- DES: Encrypts a single 8-byte block.*

 - void `cau_des_decrypt` (const unsigned char *in, const unsigned char *key, unsigned char *out)
- DES: Decrypts a single 8-byte block.*

 - void `cau_md5_initialize_output` (const unsigned char *md5_state)
- MD5: Initializes the MD5 state variables.*

 - void `cau_md5_hash_n` (const unsigned char *msg_data, const int num_blks, unsigned char *md5_state)
- MD5: Updates MD5 state variables with n message blocks.*

 - void `cau_md5_update` (const unsigned char *msg_data, const int num_blks, unsigned char *md5_state)
- MD5: Updates MD5 state variables.*

 - void `cau_md5_hash` (const unsigned char *msg_data, unsigned char *md5_state)
- MD5: Updates MD5 state variables with one message block.*

 - void `cau_sha1_initialize_output` (const unsigned int *sha1_state)
- SHA1: Initializes the SHA1 state variables.*

 - void `cau_sha1_hash_n` (const unsigned char *msg_data, const int num_blks, unsigned int *sha1_state)
- SHA1: Updates SHA1 state variables with n message blocks.*

 - void `cau_sha1_update` (const unsigned char *msg_data, const int num_blks, unsigned int *sha1_state)
- SHA1: Updates SHA1 state variables.*

 - void `cau_sha1_hash` (const unsigned char *msg_data, unsigned int *sha1_state)
- SHA1: Updates SHA1 state variables with one message block.*

 - int `cau_sha256_initialize_output` (const unsigned int *output)
- SHA256: Initializes the SHA256 state variables.*

 - void `cau_sha256_hash_n` (const unsigned char *input, const int num_blks, unsigned int *output)
- SHA256: Updates SHA256 state variables with n message blocks.*

 - void `cau_sha256_update` (const unsigned char *input, const int num_blks, unsigned int *output)
- SHA256: Updates SHA256 state variables.*

 - void `cau_sha256_hash` (const unsigned char *input, unsigned int *output)
- SHA256: Updates SHA256 state variables with one message block.*

 - status_t `MMCAU_AES_SetKey` (const uint8_t *key, const size_t keySize, uint8_t *keySch)
- AES: Performs an AES key expansion.*

 - status_t `MMCAU_AES_EncryptEcb` (const uint8_t *in, const uint8_t *keySch, uint32_t aesRounds, uint8_t *out)
- AES: Encrypts a single 16 byte block.*

 - status_t `MMCAU_AES_DecryptEcb` (const uint8_t *in, const uint8_t *keySch, uint32_t aesRounds, uint8_t *out)
- AES: Decrypts a single 16-byte block.*

 - status_t `MMCAU_DES_ChkParity` (const uint8_t *key)
- DES: Checks the key parity.*

 - status_t `MMCAU_DES_EncryptEcb` (const uint8_t *in, const uint8_t *key, uint8_t *out)
- DES: Encrypts a single 8-byte block.*

Function Documentation

- status_t [MMCAU_DES_DecryptEcb](#) (const uint8_t *in, const uint8_t *key, uint8_t *out)
DES: Decrypts a single 8-byte block.
- status_t [MMCAU_MD5_InitializeOutput](#) (uint32_t *md5State)
MD5: Initializes the MD5 state variables.
- status_t [MMCAU_MD5_HashN](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *md5State)
MD5: Updates the MD5 state variables with n message blocks.
- status_t [MMCAU_MD5_Update](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *md5State)
MD5: Updates the MD5 state variables.
- status_t [MMCAU_SHA1_InitializeOutput](#) (uint32_t *sha1State)
SHA1: Initializes the SHA1 state variables.
- status_t [MMCAU_SHA1_HashN](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *sha1State)
SHA1: Updates the SHA1 state variables with n message blocks.
- status_t [MMCAU_SHA1_Update](#) (const uint8_t *msgData, uint32_t numBlocks, uint32_t *sha1State)
SHA1: Updates the SHA1 state variables.
- status_t [MMCAU_SHA256_InitializeOutput](#) (uint32_t *sha256State)
SHA256: Initializes the SHA256 state variables.
- status_t [MMCAU_SHA256_HashN](#) (const uint8_t *input, uint32_t numBlocks, uint32_t *sha256State)
SHA256: Updates the SHA256 state variables with n message blocks.
- status_t [MMCAU_SHA256_Update](#) (const uint8_t *input, uint32_t numBlocks, uint32_t *sha256State)
SHA256: Updates SHA256 state variables.

35.6 Function Documentation

35.6.1 void cau_aes_set_key (const unsigned char * key, const int key_size, unsigned char * key_sch)

This function performs an AES key expansion

Parameters

	<i>key</i>	Pointer to input key (128, 192, 256 bits in length).
	<i>key_size</i>	Key size in bits (128, 192, 256)
out	<i>key_sch</i>	Pointer to key schedule output (44, 52, 60 longwords)

Note

All pointers must have word (4 bytes) alignment

Table below shows the requirements for the [cau_aes_set_key\(\)](#) function when using AES128, AES192 or AES256.

[in] Key Size (bits) [out] Key Schedule Size (32 bit data values)
:-----: :-----:

128 44
192 52
256 60

35.6.2 void cau_aes_encrypt (const unsigned char * *in*, const unsigned char * *key_sch*, const int *nr*, unsigned char * *out*)

This function encrypts a single 16-byte block for AES128, AES192 and AES256

Parameters

	<i>in</i>	Pointer to 16-byte block of input plaintext
	<i>key_sch</i>	Pointer to key schedule (44, 52, 60 longwords)
	<i>nr</i>	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	<i>out</i>	Pointer to 16-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

Table below shows the requirements for the [cau_aes_encrypt\(\)/cau_aes_decrypt\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
:-----:	:-----:	:-----:
AES128 44 10		
AES192 52 12		
AES256 60 14		

35.6.3 void cau_aes_decrypt (const unsigned char * *in*, const unsigned char * *key_sch*, const int *nr*, unsigned char * *out*)

This function decrypts a single 16-byte block for AES128, AES192 and AES256

Parameters

Function Documentation

	<i>in</i>	Pointer to 16-byte block of input ciphertext
	<i>key_sch</i>	Pointer to key schedule (44, 52, 60 longwords)
	<i>nr</i>	Number of AES rounds (10, 12, 14 = f(key_schedule))
out	<i>out</i>	Pointer to 16-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment

Input and output blocks may overlap.

Table below shows the requirements for the [cau_aes_encrypt\(\)](#)/[cau_aes_decrypt\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
:-----:	:-----:	:-----:
AES128	44	10
AES192	52	12
AES256	60	14

35.6.4 int cau_des_chk_parity (const unsigned char * key)

This function checks the parity of a DES key

Parameters

<i>key</i>	64-bit DES key with parity bits. Must have word (4 bytes) alignment.
------------	--

Returns

0 no error

-1 parity error

35.6.5 void cau_des_encrypt (const unsigned char * in, const unsigned char * key, unsigned char * out)

This function encrypts a single 8-byte block with DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input plaintext
	<i>key</i>	Pointer to 64-bit DES key with parity bits
out	<i>out</i>	Pointer to 8-byte block of output ciphertext

Note

All pointers must have word (4 bytes) alignment
Input and output blocks may overlap.

35.6.6 void cau_des_decrypt (const unsigned char * *in*, const unsigned char * *key*, unsigned char * *out*)

This function decrypts a single 8-byte block with DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input ciphertext
	<i>key</i>	Pointer to 64-bit DES key with parity bits
out	<i>out</i>	Pointer to 8-byte block of output plaintext

Note

All pointers must have word (4 bytes) alignment
Input and output blocks may overlap.

35.6.7 void cau_md5_initialize_output (const unsigned char * *md5_state*)

This function initializes the MD5 state variables. The output can be used as input to [cau_md5_hash\(\)](#) and [cau_md5_hash_n\(\)](#).

Parameters

out	<i>md5_state</i>	Pointer to 128-bit block of md5 state variables: a,b,c,d
-----	------------------	--

Note

All pointers must have word (4 bytes) alignment

Function Documentation

35.6.8 void cau_md5_hash_n (const unsigned char * *msg_data*, const int *num_blks*, unsigned char * *md5_state*)

This function updates MD5 state variables for one or more input message blocks

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blks</i>	Number of 512-bit blocks to process
in, out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_md5_initialize_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

35.6.9 void cau_md5_update (const unsigned char * *msg_data*, const int *num_blks*, unsigned char * *md5_state*)

This function updates MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_md5_initialize_output\(\)](#) first.

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blks</i>	Number of 512-bit blocks to process
out	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_md5_initialize_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

35.6.10 void cau_md5_hash (const unsigned char * *msg_data*, unsigned char * *md5_state*)

This function updates MD5 state variables for one input message block

Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
<i>in, out</i>	<i>md5_state</i>	Pointer to 128-bit block of MD5 state variables: a,b,c,d

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_md5_initialize_output\(\)](#) function must be called when starting a new hash.

35.6.11 void cau_sha1_initialize_output (const unsigned int * *sha1_state*)

This function initializes the SHA1 state variables. The output can be used as input to [cau_sha1_hash\(\)](#) and [cau_sha1_hash_n\(\)](#).

Parameters

<i>out</i>	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e
------------	-------------------	---

Note

All pointers must have word (4 bytes) alignment

35.6.12 void cau_sha1_hash_n (const unsigned char * *msg_data*, const int *num_blks*, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one or more input message blocks

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blks</i>	Number of 512-bit blocks to process
<i>in, out</i>	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha1_initialize_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

35.6.13 void `cau_sha1_update` (const unsigned char * *msg_data*, const int *num_blks*, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_sha1_initialize_output\(\)](#) first.

Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blks</i>	Number of 512-bit blocks to process
out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha1_initialize_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

35.6.14 void cau_sha1_hash (const unsigned char * *msg_data*, unsigned int * *sha1_state*)

This function updates SHA1 state variables for one input message block

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
in, out	<i>sha1_state</i>	Pointer to 160-bit block of SHA1 state variables: a,b,c,d,e

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha1_initialize_output\(\)](#) function must be called when starting a new hash.

35.6.15 int cau_sha256_initialize_output (const unsigned int * *output*)

This function initializes the SHA256 state variables. The output can be used as input to [cau_sha256_hash\(\)](#) and [cau_sha256_hash_n\(\)](#).

Parameters

out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables a,b,c,d,e,f,g,h
-----	---------------------	--

Note

All pointers must have word (4 bytes) alignment

Returns

- 0 No error. CAU hardware support for SHA256 is present.
- 1 Error. CAU hardware support for SHA256 is not present.

35.6.16 void cau_sha256_hash_n (const unsigned char * *input*, const int *num_blks*, unsigned int * *output*)

This function updates SHA256 state variables for one or more input message blocks

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blks</i>	Number of 512-bit blocks to process
in, out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha256_initialize_output\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

35.6.17 void cau_sha256_update (const unsigned char * *input*, const int *num_blks*, unsigned int * *output*)

This function updates SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_sha256_initialize_output\(\)](#) first.

Function Documentation

Parameters

	<i>msg_data</i>	Pointer to start of input message data
	<i>num_blks</i>	Number of 512-bit blocks to process
out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha256_initialize_output\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

35.6.18 void cau_sha256_hash (const unsigned char * *input*, unsigned int * *output*)

This function updates SHA256 state variables for one input message block

Parameters

	<i>msg_data</i>	Pointer to start of 512-bits of input message data
in, out	<i>sha256_state</i>	Pointer to 256-bit block of SHA2 state variables: a,b,c,d,e,f,g,h

Note

All pointers must have word (4 bytes) alignment

Input message and digest output blocks must not overlap. The [cau_sha256_initialize_output\(\)](#) function must be called when starting a new hash.

35.6.19 status_t MMCAU_AES_SetKey (const uint8_t * *key*, const size_t *keySize*, uint8_t * *keySch*)

This function performs an AES key expansion.

Parameters

	<i>key</i>	Pointer to input key (128, 192, 256 bits in length).
	<i>keySize</i>	Key size in bytes (16, 24, 32)
out	<i>keySch</i>	Pointer to key schedule output (44, 52, 60 longwords)

Note

Table below shows the requirements for the [MMCAU_AES_SetKey\(\)](#) function when using AES128, AES192, or AES256.

[in] Key Size (bits) [out] Key Schedule Size (32 bit data values)	
:-----: :-----:	
128 44	
192 52	
256 60	

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

35.6.20 **status_t MMCAU_AES_EncryptEcb (const uint8_t * *in*, const uint8_t * *keySch*, uint32_t *aesRounds*, uint8_t * *out*)**

This function encrypts a single 16-byte block for AES128, AES192, and AES256.

Parameters

	<i>in</i>	Pointer to 16-byte block of input plaintext.
	<i>keySch</i>	Pointer to key schedule (44, 52, 60 longwords).
	<i>aesRounds</i>	Number of AES rounds (10, 12, 14 = f(key_schedule)).
out	<i>out</i>	Pointer to 16-byte block of output ciphertext.

Note

Input and output blocks may overlap.

Table below shows the requirements for the [MMCAU_AES_EncryptEcb\(\)/MMCAU_AES_DecryptEcb\(\)](#) function when using AES128, AES192 or AES256.

Block Cipher [in] Key Schedule Size (longwords) [in] Number of AES rounds	
:-----: :-----: :-----:	
AES128 44 10	
AES192 52 12	
AES256 60 14	

Function Documentation

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

35.6.21 `status_t MMCAU_AES_DecryptEcb (const uint8_t * in, const uint8_t * keySch, uint32_t aesRounds, uint8_t * out)`

This function decrypts a single 16-byte block for AES128, AES192, and AES256.

Parameters

	<i>in</i>	Pointer to 16-byte block of input ciphertext.
	<i>keySch</i>	Pointer to key schedule (44, 52, 60 longwords).
	<i>aesRounds</i>	Number of AES rounds (10, 12, 14 = f(key_schedule)).
<i>out</i>	<i>out</i>	Pointer to 16-byte block of output plaintext.

Note

Input and output blocks may overlap.

Table below shows the requirements for the `cau_aes_encrypt()/cau_aes_decrypt()` function when using AES128, AES192 or AES256.

Block Cipher	[in] Key Schedule Size (longwords)	[in] Number of AES rounds
:-----:	:-----:	:-----:
AES128	44	10
AES192	52	12
AES256	60	14

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

35.6.22 `status_t MMCAU_DES_ChkParity (const uint8_t * key)`

This function checks the parity of a DES key.

Parameters

<i>key</i>	64-bit DES key with parity bits.
------------	----------------------------------

Returns

kStatus_Success No error.
kStatus_Fail Parity error.
kStatus_InvalidArgument Key argument is NULL.

35.6.23 **status_t MMCAU_DES_EncryptEcb (const uint8_t * *in*, const uint8_t * *key*, uint8_t * *out*)**

This function encrypts a single 8-byte block with the DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input plaintext.
	<i>key</i>	Pointer to 64-bit DES key with parity bits.
<i>out</i>	<i>out</i>	Pointer to 8-byte block of output ciphertext.

Note

Input and output blocks may overlap.

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

35.6.24 **status_t MMCAU_DES_DecryptEcb (const uint8_t * *in*, const uint8_t * *key*, uint8_t * *out*)**

This function decrypts a single 8-byte block with the DES algorithm.

Parameters

	<i>in</i>	Pointer to 8-byte block of input ciphertext.
--	-----------	--

Function Documentation

	<i>key</i>	Pointer to 64-bit DES key with parity bits.
out	<i>out</i>	Pointer to 8-byte block of output plaintext.

Note

Input and output blocks may overlap.

Returns

Status of the operation. (kStatus_Success, kStatus_InvalidArgument, kStatus_Fail)

35.6.25 status_t MMCAU_MD5_InitializeOutput (uint32_t * *md5State*)

This function initializes the MD5 state variables. The output can be used as input to [MMCAU_MD5_HashN\(\)](#).

Parameters

out	<i>md5State</i>	Pointer to 128-bit block of md5 state variables: a,b,c,d
-----	-----------------	--

35.6.26 status_t MMCAU_MD5_HashN (const uint8_t * *msgData*, uint32_t *numBlocks*, uint32_t * *md5State*)

This function updates the MD5 state variables for one or more input message blocks.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
in, out	<i>md5State</i>	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

Note

Input message and digest output blocks must not overlap. The [MMCAU_MD5_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

35.6.27 **status_t** MMCAU_MD5_Update (**const uint8_t** * *msgData*, **uint32_t** *numBlocks*, **uint32_t** * *md5State*)

This function updates the MD5 state variables for one or more input message blocks. It starts a new hash as it internally calls [MMCAU_MD5_InitializeOutput\(\)](#) first.

Function Documentation

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
out	<i>md5State</i>	Pointer to 128-bit block of MD5 state variables: a, b, c, d.

Note

Input message and digest output blocks must not overlap. The [MMCAU_MD5_InitializeOutput\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

35.6.28 status_t MMCAU_SHA1_InitializeOutput (uint32_t * *sha1State*)

This function initializes the SHA1 state variables. The output can be used as input to [MMCAU_SHA1_HashN\(\)](#).

Parameters

out	<i>sha1State</i>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.
-----	------------------	--

35.6.29 status_t MMCAU_SHA1_HashN (const uint8_t * *msgData*, uint32_t *numBlocks*, uint32_t * *sha1State*)

This function updates the SHA1 state variables for one or more input message blocks.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
in, out	<i>sha1State</i>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA1_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

35.6.30 **status_t** MMCAU_SHA1_Update (**const uint8_t** * *msgData*, **uint32_t** *numBlocks*, **uint32_t** * *sha1State*)

This function updates the SHA1 state variables for one or more input message blocks. It starts a new hash as it internally calls [MMCAU_SHA1_InitializeOutput\(\)](#) first.

Function Documentation

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
out	<i>sha1State</i>	Pointer to 160-bit block of SHA1 state variables: a, b, c, d, e.

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA1_InitializeOutput\(\)](#) function is not required to be called as it is called internally to start a new hash. All input message blocks must be contiguous.

35.6.31 **status_t** MMCAU_SHA256_InitializeOutput (uint32_t * *sha256State*)

This function initializes the SHA256 state variables. The output can be used as input to [MMCAU_SHA256_HashN\(\)](#).

Parameters

out	<i>sha256State</i>	Pointer to 256-bit block of SHA2 state variables a, b, c, d, e, f, g, h.
-----	--------------------	--

Returns

kStatus_Success No error. CAU hardware support for SHA256 is present.

kStatus_Fail Error. CAU hardware support for SHA256 is not present.

kStatus_InvalidArgument Error. sha256State is NULL.

35.6.32 **status_t** MMCAU_SHA256_HashN (const uint8_t * *input*, uint32_t *numBlocks*, uint32_t * *sha256State*)

This function updates SHA256 state variables for one or more input message blocks.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
--	----------------	---

	<i>numBlocks</i>	Number of 512-bit blocks to process.
<i>in, out</i>	<i>sha256State</i>	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA256_InitializeOutput\(\)](#) function must be called when starting a new hash. Useful when handling non-contiguous input message blocks.

35.6.33 **status_t MMCAU_SHA256_Update (const uint8_t * *input*, uint32_t *numBlocks*, uint32_t * *sha256State*)**

This function updates the SHA256 state variables for one or more input message blocks. It starts a new hash as it internally calls [cau_sha256_initialize_output\(\)](#) first.

Parameters

	<i>msgData</i>	Pointer to start of input message data.
	<i>numBlocks</i>	Number of 512-bit blocks to process.
<i>out</i>	<i>sha256State</i>	Pointer to 256-bit block of SHA2 state variables: a, b, c, d, e, f, g, h.

Note

Input message and digest output blocks must not overlap. The [MMCAU_SHA256_InitializeOutput\(\)](#) function is not required to be called, as it is called internally to start a new hash. All input message blocks must be contiguous.

Chapter 36

Secure Digital Card/Embedded MultiMedia Card (CARD)

36.1 Overview

The MCUXpresso SDK provides a driver to access the Secure Digital Card and Embedded MultiMedia Card based on the SDHC driver.

Function groups

This function group implements the SD card functional API.

This function group implements the MMC card functional API.

Typical use case

```
/* Initialize SDHC. */
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Save host information. */
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->host.transfer = SDHC_TransferFunction;

/* Init card. */
if (SD_Init(card))
{
    PRINTF("\r\nSD card init failed.\r\n");
}

while (true)
{
    if (kStatus_Success != SD_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != SD_ReadBlocks(card, g_dataRead, DATA_BLOCK_START, DATA_BLOCK_COUNT)
    )
    {
        PRINTF("Read multiple data blocks failed.\r\n");
    }

    if (kStatus_Success != SD_EraseBlocks(card, DATA_BLOCK_START, DATA_BLOCK_COUNT))
    {
        PRINTF("Erase multiple data blocks failed.\r\n");
    }
}

SD_Deinit(card);

/* Initialize SDHC. */
```

Overview

```
sdhcConfig->cardDetectDat3 = false;
sdhcConfig->endianMode = kSDHC_EndianModeLittle;
sdhcConfig->dmaMode = kSDHC_DmaModeAdma2;
sdhcConfig->readWatermarkLevel = 0x80U;
sdhcConfig->writeWatermarkLevel = 0x80U;
SDHC_Init(BOARD_SDHC_BASEADDR, sdhcConfig);

/* Save host information. */
card->host.base = BOARD_SDHC_BASEADDR;
card->host.sourceClock_Hz = CLOCK_GetFreq(BOARD_SDHC_CLKSRC);
card->host.transfer = SDHC_TransferFunction;

/* Init card. */
if (MMC_Init(card))
{
    PRINTF("\n MMC card init failed \n");
}

while (true)
{
    if (kStatus_Success != MMC_WriteBlocks(card, g_dataWrite, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF("Write multiple data blocks failed.\r\n");
    }
    if (kStatus_Success != MMC_ReadBlocks(card, g_dataRead, DATA_BLOCK_START,
        DATA_BLOCK_COUNT))
    {
        PRINTF("Read multiple data blocks failed.\r\n");
    }
}

MMC_Deinit(card);
```

Data Structures

- struct [sd_card_t](#)
SD card state. [More...](#)
- struct [sdio_card_t](#)
SDIO card state. [More...](#)
- struct [mmc_card_t](#)
SD card state. [More...](#)
- struct [mmc_boot_config_t](#)
MMC card boot configuration definition. [More...](#)

Macros

- #define [FSL_SDMMC_DRIVER_VERSION](#) (MAKE_VERSION(2U, 1U, 2U)) /*2.1.2*/
Driver version.
- #define [FSL_SDMMC_DEFAULT_BLOCK_SIZE](#) (512U)
Default block size.
- #define [HOST_NOT_SUPPORT](#) 0U
use this define to indicate the host not support feature
- #define [HOST_SUPPORT](#) 1U
use this define to indicate the host support feature
- #define [kHOST_DATABUSWIDTH1BIT](#) kSDHC_DataBusWidth1Bit
1-bit mode
- #define [kHOST_DATABUSWIDTH4BIT](#) kSDHC_DataBusWidth4Bit
4-bit mode

- #define `kHOST_DATABUSWIDTH8BIT` `kSDHC_DataBusWidth8Bit`
8-bit mode
- #define `HOST_STANDARD_TUNING_START` (0U)
standard tuning start point
- #define `HOST_TUINIG_STEP` (1U)
standard tuning step
- #define `HOST_RETUNING_TIMER_COUNT` (4U)
Re-tuning timer.

Enumerations

- enum `_sdmmc_status` {
`kStatus_SDMMC_NotSupportYet` = MAKE_STATUS(kStatusGroup_SDMMC, 0U),
`kStatus_SDMMC_TransferFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 1U),
`kStatus_SDMMC_SetCardBlockSizeFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 2U),
`kStatus_SDMMC_HostNotSupport` = MAKE_STATUS(kStatusGroup_SDMMC, 3U),
`kStatus_SDMMC_CardNotSupport` = MAKE_STATUS(kStatusGroup_SDMMC, 4U),
`kStatus_SDMMC_AllSendCidFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 5U),
`kStatus_SDMMC_SendRelativeAddressFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 6U),
`kStatus_SDMMC_SendCsdFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 7U),
`kStatus_SDMMC_SelectCardFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 8U),
`kStatus_SDMMC_SendScrFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 9U),
`kStatus_SDMMC_SetDataBusWidthFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 10U),
`kStatus_SDMMC_GoIdleFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 11U),
`kStatus_SDMMC_HandShakeOperationConditionFailed`,
`kStatus_SDMMC_SendApplicationCommandFailed`,
`kStatus_SDMMC_SwitchFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 14U),
`kStatus_SDMMC_StopTransmissionFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 15U),
`kStatus_SDMMC_WaitWriteCompleteFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 16U),
`kStatus_SDMMC_SetBlockCountFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 17U),
`kStatus_SDMMC_SetRelativeAddressFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 18U),
`kStatus_SDMMC_SwitchBusTimingFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 19U),
`kStatus_SDMMC_SendExtendedCsdFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 20U),
`kStatus_SDMMC_ConfigureBootFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 21U),
`kStatus_SDMMC_ConfigureExtendedCsdFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 22-
U),
`kStatus_SDMMC_EnableHighCapacityEraseFailed`,
`kStatus_SDMMC_SendTestPatternFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 24U),
`kStatus_SDMMC_ReceiveTestPatternFailed` = MAKE_STATUS(kStatusGroup_SDMMC, 25U),
`kStatus_SDMMC_SDIO_ResponseError` = MAKE_STATUS(kStatusGroup_SDMMC, 26U),
`kStatus_SDMMC_SDIO_InvalidArgument`,
`kStatus_SDMMC_SDIO_SendOperationConditionFail`,
`kStatus_SDMMC_InvalidVoltage` = MAKE_STATUS(kStatusGroup_SDMMC, 29U),
`kStatus_SDMMC_SDIO_SwitchHighSpeedFail` = MAKE_STATUS(kStatusGroup_SDMMC, 30-
U),

Overview

- U),
kStatus_SDMMC_SDIO_ReadCISFail = MAKE_STATUS(kStatusGroup_SDMMC, 31U),
kStatus_SDMMC_SDIO_InvalidCard = MAKE_STATUS(kStatusGroup_SDMMC, 32U),
kStatus_SDMMC_TuningFail = MAKE_STATUS(kStatusGroup_SDMMC, 33U),
kStatus_SDMMC_SwitchVoltageFail = MAKE_STATUS(kStatusGroup_SDMMC, 34U),
kStatus_SDMMC_ReTuningRequest = MAKE_STATUS(kStatusGroup_SDMMC, 35U),
kStatus_SDMMC_SetDriverStrengthFail = MAKE_STATUS(kStatusGroup_SDMMC, 36U),
kStatus_SDMMC_SetPowerClassFail = MAKE_STATUS(kStatusGroup_SDMMC, 37U) }
- SD/MMC card API's running status.*
- enum _sd_card_flag {
kSD_SupportHighCapacityFlag = (1U << 1U),
kSD_Support4BitWidthFlag = (1U << 2U),
kSD_SupportSdhcFlag = (1U << 3U),
kSD_SupportSdxcFlag = (1U << 4U),
kSD_SupportVoltage180v = (1U << 5U),
kSD_SupportSetBlockCountCmd = (1U << 6U),
kSD_SupportSpeedClassControlCmd = (1U << 7U) }
- SD card flags.*
- enum _mmc_card_flag {
kMMC_SupportHighSpeed26MHZFlag = (1U << 0U),
kMMC_SupportHighSpeed52MHZFlag = (1U << 1U),
kMMC_SupportHighSpeedDDR52MHZ180V300VFlag = (1 << 2U),
kMMC_SupportHighSpeedDDR52MHZ120VFlag = (1 << 3U),
kMMC_SupportHS200200MHZ180VFlag = (1 << 4U),
kMMC_SupportHS200200MHZ120VFlag = (1 << 5U),
kMMC_SupportHS400DDR200MHZ180VFlag = (1 << 6U),
kMMC_SupportHS400DDR200MHZ120VFlag = (1 << 7U),
kMMC_SupportHighCapacityFlag = (1U << 8U),
kMMC_SupportAlternateBootFlag = (1U << 9U),
kMMC_SupportDDRBootFlag = (1U << 10U),
kMMC_SupportHighSpeedBootFlag = (1U << 11U),
kMMC_DataBusWidth4BitFlag = (1U << 12U),
kMMC_DataBusWidth8BitFlag = (1U << 13U),
kMMC_DataBusWidth1BitFlag = (1U << 14U) }
- MMC card flags.*
- enum card_operation_voltage_t {
kCARD_OperationVoltageNone = 0U,
kCARD_OperationVoltage330V = 1U,
kCARD_OperationVoltage300V = 2U,
kCARD_OperationVoltage180V = 3U }
- card operation voltage*
- enum _host_capability
- SDHC host capability.*
- enum _host_endian_mode {
kHOST_EndianModeBig = 0U,
kHOST_EndianModeHalfWordBig = 1U,

`kHOST_EndianModeLittle = 2U }`
host Endian mode corresponding to driver define

SDCARD Function

- `status_t SD_Init (sd_card_t *card)`
Initializes the card on a specific host controller.
- `void SD_Deinit (sd_card_t *card)`
Deinitializes the card.
- `bool SD_CheckReadOnly (sd_card_t *card)`
Checks whether the card is write-protected.
- `status_t SD_ReadBlocks (sd_card_t *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`
Reads blocks from the specific card.
- `status_t SD_WriteBlocks (sd_card_t *card, const uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`
Writes blocks of data to the specific card.
- `status_t SD_EraseBlocks (sd_card_t *card, uint32_t startBlock, uint32_t blockCount)`
Erases blocks of the specific card.

MMCCARD Function

- `status_t MMC_Init (mmc_card_t *card)`
Initializes the MMC card.
- `void MMC_Deinit (mmc_card_t *card)`
Deinitializes the card.
- `bool MMC_CheckReadOnly (mmc_card_t *card)`
Checks if the card is read-only.
- `status_t MMC_ReadBlocks (mmc_card_t *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`
Reads data blocks from the card.
- `status_t MMC_WriteBlocks (mmc_card_t *card, const uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)`
Writes data blocks to the card.
- `status_t MMC_EraseGroups (mmc_card_t *card, uint32_t startGroup, uint32_t endGroup)`
Erases groups of the card.
- `status_t MMC_SelectPartition (mmc_card_t *card, mmc_access_partition_t partitionNumber)`
Selects the partition to access.
- `status_t MMC_SetBootConfig (mmc_card_t *card, const mmc_boot_config_t *config)`
Configures the boot activity of the card.
- `status_t SDIO_CardInactive (sdio_card_t *card)`
set SDIO card to inactive state
- `status_t SDIO_IO_Write_Direct (sdio_card_t *card, sdio_func_num_t func, uint32_t regAddr, uint8_t *data, bool raw)`
IO direct write transfer function.
- `status_t SDIO_IO_Read_Direct (sdio_card_t *card, sdio_func_num_t func, uint32_t regAddr, uint8_t *data)`
IO direct read transfer function.
- `status_t SDIO_IO_Write_Extended (sdio_card_t *card, sdio_func_num_t func, uint32_t regAddr, uint8_t *buffer, uint32_t count, uint32_t flags)`

Data Structure Documentation

- IO extended write transfer function.*
 - status_t **SDIO_IO_Read_Extended** (sdio_card_t *card, sdio_func_num_t func, uint32_t regAddr, uint8_t *buffer, uint32_t count, uint32_t flags)
- IO extended read transfer function.*
 - status_t **SDIO_GetCardCapability** (sdio_card_t *card, sdio_func_num_t func)
- get SDIO card capability*
 - status_t **SDIO_SetBlockSize** (sdio_card_t *card, sdio_func_num_t func, uint32_t blockSize)
- set SDIO card block size*
 - status_t **SDIO_CardReset** (sdio_card_t *card)
- set SDIO card reset*
 - status_t **SDIO_SetDataBusWidth** (sdio_card_t *card, sdio_bus_width_t busWidth)
- set SDIO card data bus width*
 - status_t **SDIO_SwitchToHighSpeed** (sdio_card_t *card)
- switch the card to high speed*
 - status_t **SDIO_ReadCIS** (sdio_card_t *card, sdio_func_num_t func, const uint32_t *tupleList, uint32_t tupleNum)
- read SDIO card CIS for each function*
 - status_t **SDIO_Init** (sdio_card_t *card)
- SDIO card init function.*
 - status_t **SDIO_EnableIOInterrupt** (sdio_card_t *card, sdio_func_num_t func, bool enable)
- enable IO interrupt*
 - status_t **SDIO_EnableIO** (sdio_card_t *card, sdio_func_num_t func, bool enable)
- enable IO and wait IO ready*
 - status_t **SDIO_SelectIO** (sdio_card_t *card, sdio_func_num_t func)
- select IO*
 - status_t **SDIO_AbortIO** (sdio_card_t *card, sdio_func_num_t func)
- Abort IO transfer.*
 - void **SDIO_DeInit** (sdio_card_t *card)
- SDIO card deinit.*

adaptor function

- static status_t **HOST_NotSupport** (void *parameter)
 - host not support function, this function is used for host not support feature*
- status_t **CardInsertDetect** (HOST_TYPE *hostBase)
 - Detect card insert, only need for SD cases.*
- status_t **HOST_Init** (void *host)
 - Init host controller.*
- void **HOST_Deinit** (void *host)
 - Deinit host controller.*

36.2 Data Structure Documentation

36.2.1 struct sd_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- HOST_CONFIG [host](#)
Host information.
- bool [isHostReady](#)
use this flag to indicate if need host re-init or not
- uint32_t [busClock_Hz](#)
SD bus clock frequency united in Hz.
- uint32_t [relativeAddress](#)
Relative address of the card.
- uint32_t [version](#)
Card version.
- uint32_t [flags](#)
Flags in _sd_card_flag.
- uint32_t [rawCid](#) [4U]
Raw CID content.
- uint32_t [rawCsd](#) [4U]
Raw CSD content.
- uint32_t [rawScr](#) [2U]
Raw CSD content.
- uint32_t [ocr](#)
Raw OCR content.
- sd_cid_t [cid](#)
CID.
- sd_csd_t [csd](#)
CSD.
- sd_scr_t [scr](#)
SCR.
- uint32_t [blockCount](#)
Card total block number.
- uint32_t [blockSize](#)
Card block size.
- sd_timing_mode_t [currentTiming](#)
current timing mode
- sd_driver_strength_t [driverStrength](#)
driver strength
- sd_max_current_t [maxCurrent](#)
card current limit
- [card_operation_voltage_t](#) [operationVoltage](#)
card operation voltage

36.2.2 struct sdio_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- HOST_CONFIG [host](#)

Data Structure Documentation

- Host information.*
- bool [isHostReady](#)
use this flag to indicate if need host re-init or not
- bool [memPresentFlag](#)
indicate if memory present
- uint32_t [busClock_Hz](#)
SD bus clock frequency united in Hz.
- uint32_t [relativeAddress](#)
Relative address of the card.
- uint8_t [sdVersion](#)
SD version.
- uint8_t [sdioVersion](#)
SDIO version.
- uint8_t [cccrVersion](#)
CCCR version.
- uint8_t [ioTotalNumber](#)
total number of IO function
- uint32_t [cccrflags](#)
Flags in _sd_card_flag.
- uint32_t [io0blockSize](#)
record the io0 block size
- uint32_t [ocr](#)
Raw OCR content, only 24bit available for SDIO card.
- uint32_t [commonCISPointer](#)
point to common CIS
- sdio_fbr_t [ioFBR](#) [7U]
FBR table.
- sdio_common_cis_t [commonCIS](#)
CIS table.
- sdio_func_cis_t [funcCIS](#) [7U]
function CIS table

36.2.3 struct mmc_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- HOST_CONFIG [host](#)
Host information.
- bool [isHostReady](#)
use this flag to indicate if need host re-init or not
- uint32_t [busClock_Hz](#)
MMC bus clock united in Hz.
- uint32_t [relativeAddress](#)
Relative address of the card.
- bool [enablePreDefinedBlockCount](#)
Enable PRE-DEFINED block count when read/write.

- uint32_t [flags](#)
Capability flag in _mmc_card_flag.
- uint32_t [rawCid](#) [4U]
Raw CID content.
- uint32_t [rawCsd](#) [4U]
Raw CSD content.
- uint32_t [rawExtendedCsd](#) [MMC_EXTENDED_CSD_BYTES/4U]
Raw MMC Extended CSD content.
- uint32_t [ocr](#)
Raw OCR content.
- mmc_cid_t [cid](#)
CID.
- mmc_csd_t [csd](#)
CSD.
- mmc_extended_csd_t [extendedCsd](#)
Extended CSD.
- uint32_t [blockSize](#)
Card block size.
- uint32_t [userPartitionBlocks](#)
Card total block number in user partition.
- uint32_t [bootPartitionBlocks](#)
Boot partition size united as block size.
- uint32_t [eraseGroupBlocks](#)
Erase group size united as block size.
- mmc_access_partition_t [currentPartition](#)
Current access partition.
- mmc_voltage_window_t [hostVoltageWindow](#)
Host voltage window.
- mmc_high_speed_timing_t [currentTiming](#)
indicate the current host timing mode

36.2.4 struct mmc_boot_config_t

Data Fields

- bool [enableBootAck](#)
Enable boot ACK.
- mmc_boot_partition_enable_t [bootPartition](#)
Boot partition.
- bool [retainBootBusWidth](#)
If retain boot bus width.
- mmc_data_bus_width_t [bootDataBusWidth](#)
Boot data bus width.

36.3 Macro Definition Documentation

36.3.1 #define FSL_SDMMC_DRIVER_VERSION (MAKE_VERSION(2U, 1U, 2U)) /*2.1.2*/

36.4 Enumeration Type Documentation

36.4.1 enum_sdmmc_status

Enumerator

kStatus_SDMMC_NotSupportYet Haven't supported.
kStatus_SDMMC_TransferFailed Send command failed.
kStatus_SDMMC_SetCardBlockSizeFailed Set block size failed.
kStatus_SDMMC_HostNotSupport Host doesn't support.
kStatus_SDMMC_CardNotSupport Card doesn't support.
kStatus_SDMMC_AllSendCidFailed Send CID failed.
kStatus_SDMMC_SendRelativeAddressFailed Send relative address failed.
kStatus_SDMMC_SendCsdFailed Send CSD failed.
kStatus_SDMMC_SelectCardFailed Select card failed.
kStatus_SDMMC_SendScrFailed Send SCR failed.
kStatus_SDMMC_SetDataBusWidthFailed Set bus width failed.
kStatus_SDMMC_GoIdleFailed Go idle failed.
kStatus_SDMMC_HandShakeOperationConditionFailed Send Operation Condition failed.
kStatus_SDMMC_SendApplicationCommandFailed Send application command failed.
kStatus_SDMMC_SwitchFailed Switch command failed.
kStatus_SDMMC_StopTransmissionFailed Stop transmission failed.
kStatus_SDMMC_WaitWriteCompleteFailed Wait write complete failed.
kStatus_SDMMC_SetBlockCountFailed Set block count failed.
kStatus_SDMMC_SetRelativeAddressFailed Set relative address failed.
kStatus_SDMMC_SwitchBusTimingFailed Switch high speed failed.
kStatus_SDMMC_SendExtendedCsdFailed Send EXT_CSD failed.
kStatus_SDMMC_ConfigureBootFailed Configure boot failed.
kStatus_SDMMC_ConfigureExtendedCsdFailed Configure EXT_CSD failed.
kStatus_SDMMC_EnableHighCapacityEraseFailed Enable high capacity erase failed.
kStatus_SDMMC_SendTestPatternFailed Send test pattern failed.
kStatus_SDMMC_ReceiveTestPatternFailed Receive test pattern failed.
kStatus_SDMMC_SDIO_ResponseError sdio response error
kStatus_SDMMC_SDIO_InvalidArgument sdio invalid argument response error
kStatus_SDMMC_SDIO_SendOperationConditionFail sdio send operation condition fail
kStatus_SDMMC_InvalidVoltage invalid voltage
kStatus_SDMMC_SDIO_SwitchHighSpeedFail switch to high speed fail
kStatus_SDMMC_SDIO_ReadCISFail read CIS fail
kStatus_SDMMC_SDIO_InvalidCard invalid SDIO card
kStatus_SDMMC_TuningFail tuning fail
kStatus_SDMMC_SwitchVoltageFail switch voltage fail
kStatus_SDMMC_ReTuningRequest retuning request
kStatus_SDMMC_SetDriverStrengthFail set driver strength fail
kStatus_SDMMC_SetPowerClassFail set power class fail

36.4.2 enum_sd_card_flag

Enumerator

kSD_SupportHighCapacityFlag Support high capacity.
kSD_Support4BitWidthFlag Support 4-bit data width.
kSD_SupportSdhcFlag Card is SDHC.
kSD_SupportSdxcFlag Card is SDXC.
kSD_SupportVoltage180v card support 1.8v voltage
kSD_SupportSetBlockCountCmd card support cmd23 flag
kSD_SupportSpeedClassControlCmd card support speed class control flag

36.4.3 enum_mmc_card_flag

Enumerator

kMMC_SupportHighSpeed26MHZFlag Support high speed 26MHZ.
kMMC_SupportHighSpeed52MHZFlag Support high speed 52MHZ.
kMMC_SupportHighSpeedDDR52MHZ180V300VFlag ddr 52MHZ 1.8V or 3.0V
kMMC_SupportHighSpeedDDR52MHZ120VFlag DDR 52MHZ 1.2V.
kMMC_SupportHS200200MHZ180VFlag HS200 ,200MHZ,1.8V.
kMMC_SupportHS200200MHZ120VFlag HS200, 200MHZ, 1.2V.
kMMC_SupportHS400DDR200MHZ180VFlag HS400, DDR, 200MHZ,1.8V.
kMMC_SupportHS400DDR200MHZ120VFlag HS400, DDR, 200MHZ,1.2V.
kMMC_SupportHighCapacityFlag Support high capacity.
kMMC_SupportAlternateBootFlag Support alternate boot.
kMMC_SupportDDRBootFlag support DDR boot flag
kMMC_SupportHighSpeedBootFlag support high speed boot flag
kMMC_DataBusWidth4BitFlag current data bus is 4 bit mode
kMMC_DataBusWidth8BitFlag current data bus is 8 bit mode
kMMC_DataBusWidth1BitFlag current data bus is 1 bit mode

36.4.4 enum_card_operation_voltage_t

Enumerator

kCARD_OperationVoltageNone indicate current voltage setting is not setting bu suser
kCARD_OperationVoltage330V card operation voltage around 3.3v
kCARD_OperationVoltage300V card operation voltage around 3.0v
kCARD_OperationVoltage180V card operation voltage around 31.8v

Function Documentation

36.4.5 enum _host_endian_mode

Enumerator

kHOST_EndianModeBig Big endian mode.

kHOST_EndianModeHalfWordBig Half word big endian mode.

kHOST_EndianModeLittle Little endian mode.

36.5 Function Documentation

36.5.1 status_t SD_Init (sd_card_t * *card*)

This function initializes the card on a specific host controller.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_Not-SupportYet</i>	Card not support.
<i>kStatus_SDMMC_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_SDMMC_All-SendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_Send-RelativeAddressFailed</i>	Send relative address failed.
<i>kStatus_SDMMC_Send-CsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_Select-CardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_Send-ScrFailed</i>	Send SCR failed.

<i>kStatus_SDMMC_SetBus-WidthFailed</i>	Set bus width failed.
<i>kStatus_SDMMC_Switch-HighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_Set-CardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.2 void SD_Deinit (sd_card_t * *card*)

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

36.5.3 bool SD_CheckReadOnly (sd_card_t * *card*)

This function checks if the card is write-protected via the CSD register.

Parameters

<i>card</i>	The specific card.
-------------	--------------------

Return values

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

36.5.4 status_t SD_ReadBlocks (sd_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

This function reads blocks from the specific card with default block size defined by the SDHC_CARD_DEFAULT_BLOCK_SIZE.

Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save the data read from card.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to read.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.5 **status_t SD_WriteBlocks (sd_card_t * *card*, const uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)**

This function writes blocks to the specific card with default block size 512 bytes.

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer holding the data to be written to the card.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to write.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.6 **status_t SD_EraseBlocks (sd_card_t * card, uint32_t startBlock, uint32_t blockCount)**

This function erases blocks of the specific card with default block size 512 bytes.

Parameters

<i>card</i>	Card descriptor.
<i>startBlock</i>	The start block index.
<i>blockCount</i>	The number of blocks to erase.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.

Function Documentation

<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.7 status_t MMC_Init (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Go-IdleFailed</i>	Go idle failed.
<i>kStatus_SDMMC_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_SDMMC_All-SendCidFailed</i>	Send CID failed.
<i>kStatus_SDMMC_Set-RelativeAddressFailed</i>	Set relative address failed.
<i>kStatus_SDMMC_Send-CsdFailed</i>	Send CSD failed.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Select-CardFailed</i>	Send SELECT_CARD command failed.
<i>kStatus_SDMMC_Send-ExtendedCsdFailed</i>	Send EXT_CSD failed.
<i>kStatus_SDMMC_SetBus-WidthFailed</i>	Set bus width failed.

<i>kStatus_SDMMC_Switch-HighSpeedFailed</i>	Switch high speed failed.
<i>kStatus_SDMMC_Set-CardBlockSizeFailed</i>	Set card block size failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.8 void MMC_Deinit (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

36.5.9 bool MMC_CheckReadOnly (mmc_card_t * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

36.5.10 status_t MMC_ReadBlocks (mmc_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save data.
<i>startBlock</i>	The start block index.

Function Documentation

<i>blockCount</i>	The number of blocks to read.
-------------------	-------------------------------

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Card-NotSupport</i>	Card not support.
<i>kStatus_SDMMC_Set-BlockCountFailed</i>	Set block count failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.11 **status_t MMC_WriteBlocks (mmc_card_t * *card*, const uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)**

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	The buffer to save data blocks.
<i>startBlock</i>	Start block number to write.
<i>blockCount</i>	Block count.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Not-SupportYet</i>	Not support now.
<i>kStatus_SDMMC_Set-BlockCountFailed</i>	Set block count failed.

<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_SDMMC_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.12 **status_t MMC_EraseGroups (mmc_card_t * *card*, uint32_t *startGroup*, uint32_t *endGroup*)**

Erase group is the smallest erase unit in MMC card. The erase range is [*startGroup*, *endGroup*].

Parameters

<i>card</i>	Card descriptor.
<i>startGroup</i>	Start group number.
<i>endGroup</i>	End group number.

Return values

<i>kStatus_InvalidArgument</i>	Invalid argument.
<i>kStatus_SDMMC_Wait-WriteCompleteFailed</i>	Send status failed.
<i>kStatus_SDMMC_-TransferFailed</i>	Transfer failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.13 **status_t MMC_SelectPartition (mmc_card_t * *card*, mmc_access_partition_t *partitionNumber*)**

Parameters

Function Documentation

<i>card</i>	Card descriptor.
<i>partition- Number</i>	The partition number.

Return values

<i>kStatus_SDMMC_- ConfigureExtendedCsd- Failed</i>	Configure EXT_CSD failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.14 **status_t MMC_SetBootConfig (mmc_card_t * *card*, const mmc_boot_config_t * *config*)**

Parameters

<i>card</i>	Card descriptor.
<i>config</i>	Boot configuration structure.

Return values

<i>kStatus_SDMMC_Not- SupportYet</i>	Not support now.
<i>kStatus_SDMMC_- ConfigureExtendedCsd- Failed</i>	Configure EXT_CSD failed.
<i>kStatus_SDMMC_- ConfigureBootFailed</i>	Configure boot failed.
<i>kStatus_Success</i>	Operate successfully.

36.5.15 **status_t SDIO_CardInActive (sdio_card_t * *card*)**

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.16 **status_t SDIO_IO_Write_Direct (sdio_card_t * *card*, sdio_func_num_t *func*, uint32_t *regAddr*, uint8_t * *data*, bool *raw*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO numner
<i>register</i>	address
<i>the</i>	data pinter to write
<i>raw</i>	flag, indicate read after write or write only

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.17 **status_t SDIO_IO_Read_Direct (sdio_card_t * *card*, sdio_func_num_t *func*, uint32_t *regAddr*, uint8_t * *data*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Function Documentation

<i>register</i>	address
<i>data</i>	pointer to read

Return values

<i>kStatus_SDMMC_- TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.18 **status_t SDIO_IO_Write_Extended (sdio_card_t * *card*, sdio_func_num_t *func*, uint32_t *regAddr*, uint8_t * *buffer*, uint32_t *count*, uint32_t *flags*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>register</i>	address
<i>data</i>	buffer to write
<i>data</i>	count
<i>write</i>	flags

Return values

<i>kStatus_SDMMC_- TransferFailed</i>	
<i>kStatus_SDMMC_SDIO- _InvalidArgument</i>	
<i>kStatus_Success</i>	

36.5.19 **status_t SDIO_IO_Read_Extended (sdio_card_t * *card*, sdio_func_num_t *func*, uint32_t *regAddr*, uint8_t * *buffer*, uint32_t *count*, uint32_t *flags*)**

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>register</i>	address
<i>data</i>	buffer to read
<i>data</i>	count
<i>write</i>	flags

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_SDMMC_SDIO-_InvalidArgument</i>	
<i>kStatus_Success</i>	

36.5.20 **status_t** SDIO_GetCardCapability (**sdio_card_t** * *card*, **sdio_func_num_t** *func*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.21 **status_t** SDIO_SetBlockSize (**sdio_card_t** * *card*, **sdio_func_num_t** *func*, **uint32_t** *blockSize*)

Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	io number
<i>block</i>	size

Return values

<i>kStatus_SDMMC_Set-CardBlockSizeFailed</i>	
<i>kStatus_SDMMC_SDIO-InvalidArgument</i>	
<i>kStatus_Success</i>	

36.5.22 **status_t** SDIO_CardReset (**sdio_card_t** * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.23 **status_t** SDIO_SetDataBusWidth (**sdio_card_t** * *card*, **sdio_bus_width_t** *busWidth*)

Parameters

<i>card</i>	Card descriptor.
<i>data</i>	bus width

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.24 **status_t** SDIO_SwitchToHighSpeed (**sdio_card_t** * *card*)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_SDMMC_SDIO-_SwitchHighSpeedFail</i>	
<i>kStatus_Success</i>	

36.5.25 **status_t** SDIO_ReadCIS (**sdio_card_t** * *card*, **sdio_func_num_t** *func*, **const uint32_t** * *tupleList*, **uint32_t** *tupleNum*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	io number
<i>tuple</i>	code list
<i>tuple</i>	code number

Return values

<i>kStatus_SDMMC_SDIO-_ReadCISFail</i>	
--	--

Function Documentation

<i>kStatus_SDMMC_- TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.26 status_t SDIO_Init (sdio_card_t * card)

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>kStatus_SDMMC_Go- IdleFailed</i>	
<i>kStatus_SDMMC_Hand- ShakeOperation- ConditionFailed</i>	
<i>kStatus_SDMMC_SDIO- _InvalidCard</i>	
<i>kStatus_SDMMC_SDIO- _InvalidVoltage</i>	
<i>kStatus_SDMMC_Send- RelativeAddressFailed</i>	
<i>kStatus_SDMMC_Select- CardFailed</i>	
<i>kStatus_SDMMC_SDIO- _SwitchHighSpeedFail</i>	
<i>kStatus_SDMMC_SDIO- _ReadCISFail</i>	
<i>kStatus_SDMMC_- TransferFailed</i>	

<i>kStatus_Success</i>	
------------------------	--

36.5.27 **status_t** SDIO_EnableIOInterrupt (**sdio_card_t** * *card*, **sdio_func_num_t** *func*, **bool** *enable*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>enable/disable</i>	flag

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.28 **status_t** SDIO_EnableIO (**sdio_card_t** * *card*, **sdio_func_num_t** *func*, **bool** *enable*)

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number
<i>enable/disable</i>	flag

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.29 **status_t** SDIO_SelectIO (**sdio_card_t** * *card*, **sdio_func_num_t** *func*)

Function Documentation

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.30 `status_t SDIO_AbortIO (sdio_card_t * card, sdio_func_num_t func)`

Parameters

<i>card</i>	Card descriptor.
<i>function</i>	IO number

Return values

<i>kStatus_SDMMC_-TransferFailed</i>	
<i>kStatus_Success</i>	

36.5.31 `void SDIO_Delnit (sdio_card_t * card)`

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

36.5.32 `static status_t HOST_NotSupport (void * parameter) [inline], [static]`

Parameters

<i>void</i>	parameter ,used to avoid build warning
-------------	--

Return values

<i>kStatus_Fail,host</i>	do not support
--------------------------	----------------

36.5.33 status_t CardInsertDetect (HOST_TYPE * *hostBase*)

Parameters

<i>hostBase</i>	the pointer to host base address
-----------------	----------------------------------

Return values

<i>kStatus_Success</i>	detect card insert
<i>kStatus_Fail</i>	card insert event fail

36.5.34 status_t HOST_Init (void * *host*)

Parameters

<i>host</i>	the pointer to host structure in card structure.
-------------	--

Return values

<i>kStatus_Success</i>	host init success
<i>kStatus_Fail</i>	event fail

36.5.35 void HOST_Deinit (void * *host*)

Parameters

Function Documentation

<i>host</i>	the pointer to host structure in card structure.
-------------	--

Chapter 37

SPI based Secure Digital Card (SDSPI)

37.1 Overview

The MCUXpresso SDK provides a driver to access the Secure Digital Card based on the SPI driver.

Function groups

This function group implements the SD card functional API in the SPI mode.

Typical use case

```
/* SPI_Init(). */

/* Register the SDSPI driver callback. */

/* Initializes card. */
if (kStatus_Success != SDSPI_Init(card))
{
    SDSPI_Deinit(card)
    return;
}

/* Read/Write card */
memset(g_testWriteBuffer, 0x17U, sizeof(g_testWriteBuffer));

while (true)
{
    memset(g_testReadBuffer, 0U, sizeof(g_testReadBuffer));

    SDSPI_WriteBlocks(card, g_testWriteBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    SDSPI_ReadBlocks(card, g_testReadBuffer, TEST_START_BLOCK, TEST_BLOCK_COUNT);

    if (memcmp(g_testReadBuffer, g_testReadBuffer, sizeof(g_testWriteBuffer)))
    {
        break;
    }
}
```

Data Structures

- struct [sdspi_command_t](#)
SDSPI command. [More...](#)
- struct [sdspi_host_t](#)
SDSPI host state. [More...](#)
- struct [sdspi_card_t](#)
SD Card Structure. [More...](#)

Enumerations

- enum `_sdspi_status` {
`kStatus_SDSPI_SetFrequencyFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 0U),
`kStatus_SDSPI_ExchangeFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 1U),
`kStatus_SDSPI_WaitReadyFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 2U),
`kStatus_SDSPI_ResponseError` = MAKE_STATUS(kStatusGroup_SDSPI, 3U),
`kStatus_SDSPI_WriteProtected` = MAKE_STATUS(kStatusGroup_SDSPI, 4U),
`kStatus_SDSPI_GoIdleFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 5U),
`kStatus_SDSPI_SendCommandFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 6U),
`kStatus_SDSPI_ReadFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 7U),
`kStatus_SDSPI_WriteFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 8U),
`kStatus_SDSPI_SendInterfaceConditionFailed`,
`kStatus_SDSPI_SendOperationConditionFailed`,
`kStatus_SDSPI_ReadOcrFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 11U),
`kStatus_SDSPI_SetBlockSizeFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 12U),
`kStatus_SDSPI_SendCsdFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 13U),
`kStatus_SDSPI_SendCidFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 14U),
`kStatus_SDSPI_StopTransmissionFailed` = MAKE_STATUS(kStatusGroup_SDSPI, 15U),
`kStatus_SDSPI_SendApplicationCommandFailed` }
SDSPI API status.
- enum `_sdspi_card_flag` {
`kSDSPI_SupportHighCapacityFlag` = (1U << 0U),
`kSDSPI_SupportSdhcFlag` = (1U << 1U),
`kSDSPI_SupportSdxcFlag` = (1U << 2U),
`kSDSPI_SupportSdscFlag` = (1U << 3U) }
SDSPI card flag.
- enum `sdspi_response_type_t` {
`kSDSPI_ResponseTypeR1` = 0U,
`kSDSPI_ResponseTypeR1b` = 1U,
`kSDSPI_ResponseTypeR2` = 2U,
`kSDSPI_ResponseTypeR3` = 3U,
`kSDSPI_ResponseTypeR7` = 4U }
SDSPI response type.

SDSPI Function

- status_t `SDSPI_Init` (`sdspi_card_t` *card)
Initializes the card on a specific SPI instance.
- void `SDSPI_Deinit` (`sdspi_card_t` *card)
Deinitializes the card.
- bool `SDSPI_CheckReadOnly` (`sdspi_card_t` *card)
Checks whether the card is write-protected.
- status_t `SDSPI_ReadBlocks` (`sdspi_card_t` *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)
Reads blocks from the specific card.
- status_t `SDSPI_WriteBlocks` (`sdspi_card_t` *card, uint8_t *buffer, uint32_t startBlock, uint32_t blockCount)

Writes blocks of data to the specific card.

37.2 Data Structure Documentation

37.2.1 struct sdsapi_command_t

Data Fields

- uint8_t [index](#)
Command index.
- uint32_t [argument](#)
Command argument.
- uint8_t [responseType](#)
Response type.
- uint8_t [response](#) [5U]
Response content.

37.2.2 struct sdsapi_host_t

Data Fields

- uint32_t [busBaudRate](#)
Bus baud rate.
- status_t(* [setFrequency](#))(uint32_t frequency)
Set frequency of SPI.
- status_t(* [exchange](#))(uint8_t *in, uint8_t *out, uint32_t size)
Exchange data over SPI.
- uint32_t(* [getCurrentMilliseconds](#))(void)
Get current time in milliseconds.

37.2.3 struct sdsapi_card_t

Define the card structure including the necessary fields to identify and describe the card.

Data Fields

- [sdsapi_host_t](#) * [host](#)
Host state information.
- uint32_t [relativeAddress](#)
Relative address of the card.
- uint32_t [flags](#)
Flags defined in `_sdsapi_card_flag`.
- uint8_t [rawCid](#) [16U]
Raw CID content.
- uint8_t [rawCsd](#) [16U]

Enumeration Type Documentation

- *Raw CSD content.*
uint8_t [rawScr](#) [8U]
- *Raw SCR content.*
uint32_t [ocr](#)
- *Raw OCR content.*
sd_cid_t [cid](#)
- *CID.*
sd_csd_t [csd](#)
- *CSD.*
sd_scr_t [scr](#)
- *SCR.*
uint32_t [blockCount](#)
- *Card total block number.*
uint32_t [blockSize](#)
- *Card block size.*

37.2.3.0.0.72 Field Documentation

37.2.3.0.0.72.1 uint32_t sdspi_card_t::flags

37.3 Enumeration Type Documentation

37.3.1 enum _sdspi_status

Enumerator

kStatus_SDSPI_SetFrequencyFailed Set frequency failed.
kStatus_SDSPI_ExchangeFailed Exchange data on SPI bus failed.
kStatus_SDSPI_WaitReadyFailed Wait card ready failed.
kStatus_SDSPI_ResponseError Response is error.
kStatus_SDSPI_WriteProtected Write protected.
kStatus_SDSPI_GoIdleFailed Go idle failed.
kStatus_SDSPI_SendCommandFailed Send command failed.
kStatus_SDSPI_ReadFailed Read data failed.
kStatus_SDSPI_WriteFailed Write data failed.
kStatus_SDSPI_SendInterfaceConditionFailed Send interface condition failed.
kStatus_SDSPI_SendOperationConditionFailed Send operation condition failed.
kStatus_SDSPI_ReadOcrFailed Read OCR failed.
kStatus_SDSPI_SetBlockSizeFailed Set block size failed.
kStatus_SDSPI_SendCsdFailed Send CSD failed.
kStatus_SDSPI_SendCidFailed Send CID failed.
kStatus_SDSPI_StopTransmissionFailed Stop transmission failed.
kStatus_SDSPI_SendApplicationCommandFailed Send application command failed.

37.3.2 enum _sdspi_card_flag

Enumerator

kSDSPI_SupportHighCapacityFlag Card is high capacity.

kSDSPI_SupportSdhcFlag Card is SDHC.

kSDSPI_SupportSdxcFlag Card is SDXC.

kSDSPI_SupportSdscFlag Card is SDSC.

37.3.3 enum sdspi_response_type_t

Enumerator

kSDSPI_ResponseTypeR1 Response 1.

kSDSPI_ResponseTypeR1b Response 1 with busy.

kSDSPI_ResponseTypeR2 Response 2.

kSDSPI_ResponseTypeR3 Response 3.

kSDSPI_ResponseTypeR7 Response 7.

37.4 Function Documentation

37.4.1 status_t SDSPI_Init (sdspi_card_t * *card*)

This function initializes the card on a specific SPI instance.

Parameters

<i>card</i>	Card descriptor
-------------	-----------------

Return values

<i>kStatus_SDSPI_Set-FrequencyFailed</i>	Set frequency failed.
<i>kStatus_SDSPI_GoIdle-Failed</i>	Go idle failed.
<i>kStatus_SDSPI_Send-InterfaceConditionFailed</i>	Send interface condition failed.

Function Documentation

<i>kStatus_SDSPI_Send-OperationCondition-Failed</i>	Send operation condition failed.
<i>kStatus_Timeout</i>	Send command timeout.
<i>kStatus_SDSPI_Not-SupportYet</i>	Not support yet.
<i>kStatus_SDSPI_ReadOcr-Failed</i>	Read OCR failed.
<i>kStatus_SDSPI_SetBlock-SizeFailed</i>	Set block size failed.
<i>kStatus_SDSPI_SendCsd-Failed</i>	Send CSD failed.
<i>kStatus_SDSPI_SendCid-Failed</i>	Send CID failed.
<i>kStatus_Success</i>	Operate successfully.

37.4.2 void SDSPI_Deinit (sdspi_card_t * *card*)

This function deinitializes the specific card.

Parameters

<i>card</i>	Card descriptor
-------------	-----------------

37.4.3 bool SDSPI_CheckReadOnly (sdspi_card_t * *card*)

This function checks if the card is write-protected via CSD register.

Parameters

<i>card</i>	Card descriptor.
-------------	------------------

Return values

<i>true</i>	Card is read only.
<i>false</i>	Card isn't read only.

37.4.4 **status_t SDSPI_ReadBlocks (sdspi_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)**

This function reads blocks from specific card.

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	the buffer to hold the data read from card
<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to read

Return values

<i>kStatus_SDSPI_Send-CommandFailed</i>	Send command failed.
<i>kStatus_SDSPI_Read-Failed</i>	Read data failed.
<i>kStatus_SDSPI_Stop-TransmissionFailed</i>	Stop transmission failed.
<i>kStatus_Success</i>	Operate successfully.

37.4.5 **status_t SDSPI_WriteBlocks (sdspi_card_t * *card*, uint8_t * *buffer*, uint32_t *startBlock*, uint32_t *blockCount*)**

This function writes blocks to specific card

Parameters

<i>card</i>	Card descriptor.
<i>buffer</i>	the buffer holding the data to be written to the card

Function Documentation

<i>startBlock</i>	the start block index
<i>blockCount</i>	the number of blocks to write

Return values

<i>kStatus_SDSPI_Write-Protected</i>	Card is write protected.
<i>kStatus_SDSPI_Send-CommandFailed</i>	Send command failed.
<i>kStatus_SDSPI_-ResponseError</i>	Response is error.
<i>kStatus_SDSPI_Write-Failed</i>	Write data failed.
<i>kStatus_SDSPI_-ExchangeFailed</i>	Exchange data over SPI failed.
<i>kStatus_SDSPI_Wait-ReadyFailed</i>	Wait card to be ready status failed.
<i>kStatus_Success</i>	Operate successfully.

How to Reach Us:**Home Page:**nxp.com**Web Support:**nxp.com/support

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

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

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

© 2017 NXP B.V.

Document Number: MCUXSDKKW24APIRM

Rev. 0

Mar 2017

